

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

<b>Document Status</b>	Final
<b>Part of AUTOSAR Standard</b>	Adaptive Platform
<b>Part of Standard Release</b>	17-03

<b>Document Change History</b>			
<b>Date</b>	<b>Release</b>	<b>Changed by</b>	<b>Description</b>
2017-03-31	17-03	AUTOSAR Release Management	<ul style="list-style-type: none"><li>• Initial release</li></ul>

**Disclaimer**

This work (specification and/or software implementation) and the material contained in it, as released by AUTOSAR, is for the purpose of information only. AUTOSAR and the companies that have contributed to it shall not be liable for any use of the work.

The material contained in this work is protected by copyright and other types of intellectual property rights. The commercial exploitation of the material contained in this work requires a license to such intellectual property rights.

This work may be utilized or reproduced without any modification, in any form or by any means, for informational purposes only. For any other purpose, no part of the work may be utilized or reproduced, in any form or by any means, without permission in writing from the publisher.

The work has been developed for automotive applications only. It has neither been developed, nor tested for non-automotive applications.

The word AUTOSAR and the AUTOSAR logo are registered trademarks.

## Table of Contents

1	Background	7
2	The vision	8
2.1	Rationale for the production of AUTOSAR C++14	8
2.2	Objectives of AUTOSAR C++14	8
3	Scope	10
3.1	Allowed features of C++ language	10
3.2	Limitations	13
4	Using AUTOSAR C++14	14
5	Introduction to the rules	15
5.1	Rule classification	15
5.1.1	Rule classification according to compatibility with MISRA	15
5.1.2	Rule classification according to obligation level	15
5.1.3	Rule classification according to enforcement by static analysis	15
5.1.4	Rule classification according to allocated target	16
5.2	Organization of rules	16
5.3	Exceptions to the rules	16
5.4	Redundancy in the rules	16
5.5	Presentation of rules	17
5.6	Understanding the issue references	17
5.7	Scope of rules	17
6	AUTOSAR C++14 coding rules	18
6.0	Language independent issues	18
6.0.1	Unnecessary constructs	18
6.0.2	Storage	23
6.0.3	Runtime failures	23
6.0.4	Arithmetic	24
6.1	General	26
6.1.1	Scope	26
6.1.2	Normative references	28
6.1.4	Implementation compliance	29
6.2	Lexical conventions	30
6.2.3	Character sets	30
6.2.5	Trigraph sequences	31
6.2.6	Alternative tokens	31
6.2.8	Comments	32
6.2.9	Header names	36
6.2.11	Identifiers	36
6.2.14	Literals	41
6.3	Basic concepts	44

6.3.1	Declarations and definitions	44
6.3.2	One Definition Rule	46
6.3.3	Scope	47
6.3.4	Name lookup	51
6.3.9	Types	51
6.4	Standard conversions	52
6.4.5	Integral promotions	52
6.4.7	Integral conversion	55
6.4.10	Pointer conversions	57
6.5	Expressions	58
6.5.0	General	58
6.5.1	Primary expression	67
6.5.2	Postfix expressions	75
6.5.3	Unary expressions	83
6.5.6	Multiplicative operators	84
6.5.8	Shift operators	85
6.5.10	Equality operators	86
6.5.14	Logical AND operator	86
6.5.16	Conditional operator	87
6.5.18	Assignment and compound assignment operation	87
6.5.19	Comma operator	87
6.5.20	Constant expression	88
6.6	Statements	88
6.6.2	Expression statement	88
6.6.3	Compound statement or block	89
6.6.4	Selection statements	89
6.6.5	Iteration statements	91
6.6.6	Jump statements	94
6.7	Declaration	96
6.7.1	Specifiers	96
6.7.2	Enumeration declaration	104
6.7.3	Namespaces	109
6.7.4	The asm declaration	109
6.7.5	Linkage specification	111
6.8	Declarators	115
6.8.0	General	115
6.8.2	Ambiguity resolution	115
6.8.3	Meaning of declarators	116
6.8.4	Function definitions	116
6.8.5	Initializers	118
6.9	Classes	125
6.9.3	Member function	125
6.9.5	Unions	128
6.9.6	Bit-fields	128
6.10	Derived Classes	130
6.10.1	Multiple base Classes	130

6.10.2	Member name lookup	132
6.10.3	Virtual functions	133
6.11	Member access control	139
6.11.0	General	139
6.11.3	Friends	142
6.12	Special member functions	142
6.12.0	General	142
6.12.1	Constructors	143
6.12.4	Destructors	149
6.12.6	Initialization	151
6.12.7	Construction and destructions	153
6.12.8	Copying and moving class objects	155
6.13	Overloading	167
6.13.1	Overloadable declarations	167
6.13.2	Declaration matching	170
6.13.3	Overload resolution	173
6.13.5	Overloaded operators	174
6.13.6	Build-in operators	175
6.14	Templates	176
6.14.0	General	176
6.14.1	Template parameters	176
6.14.5	Template declarations	179
6.14.6	Name resolution	179
6.14.7	Template instantiation and specialization	179
6.14.8	Function template specializations	181
6.15	Exception handling	183
6.15.0	General	186
6.15.1	Throwing an exception	201
6.15.2	Constructors and destructors	211
6.15.3	Handling an exception	215
6.15.4	Exception specifications	225
6.15.5	Special functions	234
6.16	Preprocessing directives	241
6.16.0	General	241
6.16.1	Conditional inclusion	244
6.16.2	Source file inclusion	245
6.16.3	Macro replacement	247
6.16.6	Error directive	248
6.16.7	Pragma directive	249
6.17	Library introduction - partial	249
6.17.1	General	249
6.17.2	The C standard library	251
6.17.3	Definitions	252
6.18	Language support library - partial	253
6.18.0	General	253
6.18.1	Types	254

6.18.2	Implementation properties	258
6.18.5	Dynamic memory management	259
6.18.9	Other runtime support	268
6.19	Diagnostics library - partial	272
6.19.4	Error numbers	272
6.23	Containers library - partial	272
6.23.1	General	272
6.27	Input/output library - partial	274
6.27.1	General	274
7	References	277
A	Traceability to existing standards	278
A.1	Traceability to MISRA C++:2008	278
A.2	Traceability to HIC++ v4.0	297
A.3	Traceability to JSF	309
A.4	Traceability to SEI CERT C++	325
A.5	Traceability to C++ Core Guidelines	334
B	Glossary	366

# 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 [6]. 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 [A.1](#) for the comparison. The adopted MISRA rules are only referenced by ID and title, without providing the full contents.

The standard specifies 342 rules, from which:

1. 154 rules are adopted without modifications from MISRA C++:2008 (this means 67% of MISRA is adopted without modifications).
2. 131 rules are derived/based on existing C++ standards
3. 57 rules are based on research or other literature or other resources.

The MISRA C++:2008 rules are referenced by ID and title. The inclusion of ID and of the rule title for the adopted rules is considered not to 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 [7]
2. High Integrity C++ Coding Standard Version 4.0 [8]
3. CERT C++ Coding Standard [9]
4. C++ Core Guidelines [10]
5. Google C++ Style Guide [11]

### 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
<b>6.8 Declarators</b>				
	Default arguments	-	X	
	Variadic arguments	-		X
	List initialization	C++11	X	
<b>6.9 Classes</b>				
	Unions	-		X
	Bit-fields	-	X	
<b>6.10 Derived Classes</b>				
	Inheritance	-	X	
	Multiple inheritance	-		X
	Virtual functions	-	X	
	override specifier	C++11	X	
	final specifier	C++11	X	
<b>6.11 Member Access Control</b>				
	friend declaration	-		X
<b>6.12 Special Member Functions</b>				
	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	
<b>6.13 Overloading</b>				
	User-defined literals	C++11		X
	Digit sequences separators '	C++14	X	
<b>6.14 Templates</b>				
	Variadic templates	C++11	X	
	Variable templates	C++14	X	
<b>6.15 Exception Handling</b>				
	Exceptions	-	X	
	Function-try-blocks	-		X
	Dynamic exception specification	-		X
	noexcept specifier	C++11	X	
<b>6.16 Preprocessing Directives</b>				
	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 C++ standard libraries is only initial (incomplete)
3. The rule set for security (as long as it is not common to critical software or safety-related software) is not provided
4. The traceability to JSF, ISO CPP contains some non-analyzed rules
5. The traceability to ISO 26262 is not provided

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.

## 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 [6]

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

See MISRA C++ 2008 [6]

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 [6]

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

See MISRA C++ 2008 [6]

**Rule M0-1-5 (required, implementation, automated)**  
**A project shall not contain unused type declarations.**

See MISRA C++ 2008 [6]

**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

```
1  //% $Id: A0-1-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2  #include <array>
3  #include <cstdint>
4  std::uint8_t fn1(std::uint8_t param) noexcept
5  {
6      std::int32_t x{
7          0}; // Non-compliant - DU data flow anomaly; Variable defined,
8             // but not used
9      if (param > 0)
10     {
11         return 1;
12     }
13     else
14     {
15         return 0;
16     }
17 }
18 std::int32_t fn2() noexcept
19 {
20     std::int8_t x{10U}; // Compliant - variable defined and will be used
21     std::int8_t y{20U}; // Compliant - variable defined and will be used
22     std::int16_t result = x + y; // x and y variables used
23
24     x = 0; // Non-compliant - DU data flow anomaly; Variable defined, but x is
25            // not subsequently used and goes out of scope
26     y = 0; // Non-compliant - DU data flow anomaly; Variable defined, but y is
27            // not subsequently used and goes out of scope
28     return result;
29 }
30 std::int32_t fn3(std::int32_t param) noexcept
31 {
32     std::int32_t x{param +
33                   1}; // Compliant - variable defined, and will be used in
34                       // one of the branches
35                       // However, scope of x variable could be reduced
36     if (param > 20)
37     {
38         return x;
39     }
40     return 0;
41 }
42 std::int32_t fn4(std::int32_t param) noexcept
```

```
43 {
44     std::int32_t x{param +
45                 1}; // Compliant - variable defined, and will be used in
46                 // some of the branches
47     if (param > 20)
48     {
49         return x + 1;
50     }
51     else if (param > 10)
52     {
53         return x;
54     }
55     else
56     {
57         return 0;
58     }
59 }
60 void fn5() noexcept
61 {
62     std::array<std::int32_t, 100> arr{};
63     arr.fill(1);
64
65     constexpr std::uint8_t limit{100U};
66     std::int8_t x{0};
67     for (std::uint8_t i{0U}; i < limit; ++i) // Compliant by exception - on the
68     // final loop, value of i defined will
69     // not be used
70     {
71         arr[i] = arr[x];
72         ++x; // Non-compliant - DU data flow anomaly on the final loop, value
73             // defined and not used
74     }
75 }
```

## 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

```
1 // $Id: A0-1-2.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <algorithm>
3 #include <cstdint>
4 #include <vector>
5 std::uint8_t fn1() noexcept
6 {
7     return 0U;
8 }
9 void fn2() noexcept
10 {
11     std::uint8_t x = fn1(); // Compliant
12     fn1(); // Non-compliant
13     static_cast<void>(fn1()); // Compliant by exception
14 }
15 void fn3()
16 {
17     std::vector<std::int8_t> v{0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5};
18     std::unique(v.begin(), v.end()); // Non-compliant
19     v.erase(std::unique(v.begin(), v.end()), v.end()); // Compliant
20 }
```

## See also

- MISRA C++ 2008 [6]: 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 [8]: 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 [6]

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

See MISRA C++ 2008 [6]

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

See MISRA C++ 2008 [6]

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 static function or private method of a class shall be used.**

## Rationale

Static functions or private class's methods that are not used may be symptomatic of a serious problems, such as poor software design or missing paths in flow control.

This rule applies to all defined static and non-static methods declared in class's private section and all defined static functions.

This rule enforces developers to statically and explicitly use every static function or private method of a class 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 the statement applies to private struct's methods as well.

## Example

```

1  //% $Id: A0-1-3.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
2  #include <stdint>
3  static void f1() // Compliant
4  {
5  }
6  static void f2() // Non-compliant, defined static function never used
7  {
8  }
9  class C
10 {
11     public:
12         C() : x(0) {}
13         void m1(std::int32_t i) // Compliant, method m1 is used
14         {
15             x = i;
16         }
17         void m2(std::int32_t i,
18                 std::int32_t j) // Compliant, never used but declared
19                                 // as public
20         {
21             x = (i > j) ? i : j;
22         }
23
24     protected:
25         void m1ProtectedImpl(std::int32_t j) // Compliant, never used but declared
26                                               // as protected
27         {
28             x = j;
29         }

```

```
30
31 private:
32     std::int32_t x;
33     void m1PrivateImpl(
34         std::int32_t j) // Non-compliant, private method never used
35     {
36         x = j;
37     }
38 };
39 int main(int, char**)
40 {
41     f1();
42     C c;
43     c.m1(1);
44     return 0;
45 }
```

### See also

- MISRA C++ 2008 [6]: Rule 0-1-10 required Every defined function shall be called at least once.
- HIC++ v4.0 [8]: 1.2.2 Ensure that no expression or sub-expression is redundant.

#### **Rule M0-1-11 (required, implementation, automated)**

**There shall be no unused parameters (named or unnamed) in non-virtual functions.**

See MISRA C++ 2008 [6]

#### **Rule M0-1-12 (required, implementation, automated)**

**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.**

See MISRA C++ 2008 [6]

## 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 [6]

## 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 [6]

**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 [6]

Note: Rule M0-3-2 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 [6]

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

See MISRA C++ 2008 [6]

**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(
7      std::numeric_limits<float>::digits == 24,
8      "Type float does not comply with IEEE 754 single precision format");
9
10 static_assert(
11     std::numeric_limits<double>::is_iec559,
12     "type double does not comply with IEEE 754 double precision format");
13 static_assert(
14     std::numeric_limits<double>::digits == 53,
15     "Type double does not comply with IEEE 754 double precision format");
```

### See also

- MISRA C++ 2008 [6]: Rule 0-4-3 Floating-point implementations shall comply with a defined floating-point standard.
- JSF December 2005 [7]: 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

```
1  //% $Id: A0-4-2.cpp 270021 2017-03-09 15:30:37Z piotr.tanski $
2  void fn() noexcept
3  {
4      float f1{0.1F};           // Compliant
5      double f2{0.1};          // Compliant
6      long double f3{0.1L};    // Non-compliant
7  }
```

### 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 ISO/IEC 14882:2014 C++ Language Standard the integer division and modulo operator results are no longer implementation-defined. “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 modulo operator as it is defined in terms of integer division.

Deducing a type of an auto variable initialized using `={} with a single element can be implemented in non-standard way on some compilers. The deduced type of such initialization should be of the variable type according to C++14 Language Standard, while C++17 Language Standard defines that it should be of an std::initializer_list.`

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

### See also

- MISRA C++ 2008 [6]: Rule 1-0-3 The implementation of integer division in the chosen compiler shall be determined and documented.

## 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 [5].

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 271687 2017-03-23 08:57:35Z piotr.tanski $
2  #include <cstdlib>
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 [6]: 1-0-1 All code shall conform to ISO/IEC 14882:2003 “The C++ Standard Incorporating Technical Corrigendum 1”
- JSF December 2005 [7]: 4.4.1 All code shall conform to ISO/IEC 14882:2002(E) standard C++.
- HIC++ v4.0 [8]: 1.1.1 Ensure that code complies with the 2011 ISO C++ Language Standard.
- HIC++ v4.0 [8]: 1.3.4 Do not use deprecated STL library features.

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

See MISRA C++ 2008 [6]

**Rule A1-1-2 (required, 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 [7]: AV Rule 218 Compiler warning levels will be set in compliance with project policies.

**Rule A1-1-3 (required, implementation, 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, implementation, 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 [5].**

**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 [5]: 11 Confidence in the use of software tools.

### 6.1.4 Implementation compliance

**Rule A1-4-1 (required, implementation, non-automated)**  
**Code metrics and their valid boundaries shall be defined.**

## Rationale

Code metrics that concern i.a. 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.

## See also

- HIC++ v4.0 [8]: 8.3.1 Do not write functions with an excessive McCabe Cyclomatic Complexity.
- HIC++ v4.0 [8]: 8.3.2 Do not write functions with a high static program path count.
- HIC++ v4.0 [8]: 8.2.2 Do not declare functions with an excessive number of parameters.

**Rule A1-4-2 (required, implementation, non-automated)**  
**All code shall comply with defined boundaries of code metrics.**

### Rationale

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

- none

## 6.2 Lexical conventions

### 6.2.3 Character sets

**Rule A2-2-1 (required, 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.

### Example

```
1 // $Id: A2-2-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <stdint>
3 void fn() noexcept
4 {
5     std::int32_t sum = 0; // Compliant
6     // std::int32_t Â£_value = 10; // Non-compliant
7     // sum += Â£_value; // Non-compliant
8     // Variable sum stores Â£ pounds // Non-compliant
9 }
```

### See also

- JSF December 2005 [7]: AV Rule 9: Only those characters specified in the C++ basic source character set will be used.

## 6.2.5 Trigraph sequences

**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

```

1 //% $Id: A2-5-1.cpp 270728 2017-03-16 10:38:20Z piotr.tanski $
2 #include <iostream>
3 void fn1()
4 {
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 [7]: AV Rule 11 Trigraphs will not be used.
- HIC++ v4.0 [8]: 2.2.1 Do not use digraphs or trigraphs.

## 6.2.6 Alternative tokens

**Rule A2-6-1 (required, implementation, automated)**  
**Digraphs shall not be used.**

### Rationale

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

The use of digraphs may not meet developer expectations.

### Example

```
1  // % $Id: A2-6-1.cpp 266557 2017-02-07 13:08:19Z piotr.tanski $
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 [6]: advisory 2-5-1 Digraphs should not be used.
- JSF December 2005 [7]: 4.4.1 AV Rule 12 The following digraphs will not be used.
- HIC++ v4.0 [8]: 2.2.1 Do not use digraphs or trigraphs.

## 6.2.8 Comments

**Rule A2-8-1 (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: A2-8-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2  #include <stdint>
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 \
8             // character occurrence in the end of C++ comment
9
10     constexpr std::int8_t limit = 10;
11     for (; idx <= limit; ++idx)
12     {
13         // ...
14     }
```



```
14 }
```

## See also

- none

**Rule A2-8-2 (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: [A2-8-4](#).

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

```
1 // $Id: A2-8-2.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <cstdint>
3 void fn1() noexcept
4 {
5     std::int32_t i = 0;
6     // /*
7     // * ++i; /* incrementing the variable i */
8     // */ // Non-compliant - C-style comments nesting is not supported,
9     // compilation error
10    for (; i < 10; ++i)
11    {
12        // ...
13    }
14 }
15 void fn2() noexcept
16 {
17     std::int32_t i = 0;
18     // ++i; // Incrementing the variable i // Non-compliant - code should not
19     // be commented-out
20     for (; i < 10; ++i)
21     {
22         // ...
23     }
24 }
25 void fn3() noexcept
26 {
27     std::int32_t i = 0;
```

```

28     ++i; // Incrementing the variable i using ++i syntax // Compliant - code
29         // is not commented-out, but ++i occurs in a
30         // comment too
31     for (; i < 10; ++i)
32     {
33         // ...
34     }
35 }
  
```

## See also

- MISRA C++ 2008 [6]: Rule 2-7-2 Sections of code shall not be “commented out” using C-style comments.
- MISRA C++ 2008 [6]: Rule 2-7-3 Sections of code should not be “commented out” using C++ comments.

**Rule A2-8-3 (required, implementation, automated)**  
**All declarations of “user-defined” types, static and non-static data members, functions and methods shall be preceded by documentation using “///” comments and “@tag” tags.**

## 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.

Note that the documentation style is also supported by external tools, e.g. doxygen.

## Example

```

1  ///< $Id: A2-8-3.hpp 271696 2017-03-23 09:23:09Z piotr.tanski $
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 description
33         std::int const* getX() const noexcept { return &x; }
34
35     private:
36         /// @brief Data member description
37         std::int32_t x;
38         /// @brief Data member description
39         float y;
40 };

```

## See also

- none

**Rule A2-8-4 (required, implementation, automated)**  
**C-style comments shall not be used.**

## Rationale

C-style comment delimiters `/* ... */` are not supposed to be used as they make the source code less readable and introduce errors when nesting a C-style comment in the C-style comment.

## Example

```

1 // $Id: A2-8-4.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <stdint>
3 void fn(bool b1, bool b2) noexcept
4 {
5     // Introduces x of type std::int16_t // Compliant
6     std::int32_t x = 0;
7     // std::int16_t x = 0; // commented out temporarily, type too small //
8     // Compliant
9     if (b1)
10    {
11        if (b2)
12        {

```

```
13         /*
14         * Do something special here // Non-compliant
15         */
16     }
17 }
18
19 // /* disable this code temporarily
20 // if (b1)
21 // {
22 // /* TODO: should we do something? */ // Non-compliant - compilation
23 // error
24 // }
25 // */
26 }
```

### See also

- MISRA C++ 2008 [6]: 2-7-1 The character sequence `/*` shall not be used within a C-style comment.
- HIC++ v4.0 [8]: 2.3.1 Do not use the C comment delimiters `/* ... */`.

### 6.2.9 Header names

**Rule A2-9-1 (required, implementation, automated)**  
**A header file name shall be identical to a type name declared in it if it declares a type.**

#### Rationale

Naming a header file with a name of a type (e.g. a struct, a class, etc.) declared in it makes include-directives and project view more readable.

#### See also

- none

### 6.2.11 Identifiers

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

See MISRA C++ 2008 [6]

**Rule A2-11-1 (required, 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: A2-11-1.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
2 #include <stdint>
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 }
16 std::int32_t idx;
17 void f1(std::int32_t);
18 void f2()
19 {
20     std::int32_t max = 5;
21
22     for (std::int32_t idx = 0; idx < max;
```

```

23         ++idx) // Non-compliant, hides idx in outer scope
24     {
25         for (std::int32_t idx = 0; idx < max;
26             ++idx) // Non-compliant, hides idx in outer scope
27             {
28             }
29     }
30 }
31 void f3()
32 {
33     std::int32_t i = 0;
34     std::int32_t j = 0;
35     auto lambda = [i]() {
36         std::int32_t j =
37             10; // Compliant - j was not captured, so it does not hide
38                // j in outer scope
39         return i + j;
40     };
41 }

```

## See also

- MISRA C++ 2008 [6]: required 2-10-2 Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
- JSF December 2005 [7]: 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 [8]: 3.1.1 Do not hide declarations.

**Rule M2-10-3 (required, implementation, automated)**  
**A typedef name (including qualification, if any) shall be a unique identifier.**

See MISRA C++ 2008 [6]

**Rule A2-11-2 (required, implementation, automated)**  
**A “using” name shall be a unique identifier within a namespace.**

## Rationale

Reusing a using name either as another using name or for any other purpose may lead to developer confusion. The same using shall not be duplicated anywhere within a namespace.

## Example

```

1 //% $Id: A2-11-2.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
2 #include <cstdlib>
3 namespace n1
4 {

```

```
5 using func = void (*) (std::int32_t, std::int32_t);
6 void f1 ()
7 {
8     using func =
9         void (*) (void); // Non-compliant, reuses func identifier declared
10                        // in the same namespace
11 }
12 }
13 namespace n2
14 {
15 using func = void (*) (std::int32_t,
16                        std::int32_t); // Compliant, reuses func identifier but
17                                    // in another namespace
18 }
```

### See also

- MISRA C++ 2008 [6]: Rule 2-10-3 (Required) A typedef name (including qualification, if any) shall be a unique identifier.
- JSF December 2005 [7]: 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 [8]: 2.4.1 Ensure that each identifier is distinct from any other visible identifier

#### **Rule A2-11-3 (required, implementation, automated)**

**A “user-defined” type name shall be a unique identifier within a namespace.**

### Rationale

Reusing a user-defined type name, either as another type or for any other purpose, may lead to developer confusion. The user-defined type name shall not be duplicated anywhere in the project, even if the declaration is identical. The term user-defined type is defined as follows: - class - struct - union - enumeration - typedef / using

### Example

```
1 // % $Id: A2-11-3.cpp 272338 2017-03-28 08:15:01Z piotr.tanski $
2 #include <stdint>
3 class Type
4 {
5 };
6 // struct Type { }; // Non-compliant, Type name reused
7 // enum class Type : std::int8_t { }; // Non-compliant, Type name reused
```

### See also

- MISRA C++ 2008 [6]: required 2-10-4 A class, union or enum name (including qualification, if any) shall be a unique identifier.

- JSF December 2005 [7]: 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 [8]: 2.4.1 Ensure that each identifier is distinct from any other visible identifier

**Rule A2-11-4 (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: A2-11-4.cpp 272338 2017-03-28 08:15:01Z piotr.tanski $
2 #include <cstdint>
3 // f1.cpp
4 namespace ns1
5 {
6     static std::int32_t globalvariable = 0;
7 }
8
9 // f2.cpp
10 namespace ns1
11 {
12     // static std::int32_t globalvariable = 0; // Non-compliant - identifier reused
13     // in ns1 namespace in f1.cpp
14 }
15 namespace ns2
16 {
17     static std::int32_t globalvariable =
18         0; // Compliant - identifier reused, but in another namespace
19 }
20
21 // f3.cpp
22 static std::int32_t globalvariable =
23     0; // Compliant - identifier reused, but in another namespace
```

## See also

- MISRA C++ 2008 [6]: advisory 2-10-5 The identifier name of a non-member object or function with static storage duration should not be reused.



**Rule A2-11-5 (advisory, implementation, automated)****An identifier name of a non-member object or function with static storage duration 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.

**Example**

```
1 // % $Id: A2-11-5.cpp 271927 2017-03-24 12:01:35Z piotr.tanski $
2 #include <stdint>
3 // f1.cpp
4 namespace n_s1
5 {
6     static std::int32_t globalvariable = 0;
7 }
8 static std::int32_t filevariable = 5; // Compliant - identifier not reused
9 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 [6]: advisory 2-10-5 The identifier name of a non-member object or function with static storage duration should not be reused.

**Rule M2-10-6 (required, implementation, automated)****If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.**

See MISRA C++ 2008 [6]

**6.2.14 Literals**

**Rule A2-14-1 (required, 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>'`, `'\U<Unicode Character Name>'`

### Example

```
1 //% $Id: A2-14-1.cpp 266557 2017-02-07 13:08:19Z piotr.tanski $
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 [6]: required 2-13-1 Only those escape sequences that are defined in ISO/IEC14882:2003 shall be used.

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

See MISRA C++ 2008 [6]

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

See MISRA C++ 2008 [6]

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

See MISRA C++ 2008 [6]

**Rule A2-14-2 (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: A2-14-2.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
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 A2-14-3
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 [6]: required 2-13-5 Narrow and wide string literals shall not be concatenated.
- HIC++ v4.0 [8]: 2.5.1 Do not concatenate strings with different encoding prefixes

**Rule A2-14-3 (required, 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: A2-14-3.cpp 266557 2017-02-07 13:08:19Z piotr.tanski $
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

## 6.3 Basic concepts

### 6.3.1 Declarations and definitions

**Rule A3-1-1 (required, 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

```
1 //% $Id: A3-1-1.hpp 271715 2017-03-23 10:13:51Z piotr.tanski $
2 #include <cstdint>
3 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 [6]: 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, 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 [7]: 4.9.2 AV Rule 53 Header files will always have a file name extension of ".h".

**Rule A3-1-3 (advisory, 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 [7]: 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 [6]

**Rule A3-1-4 (required, 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 <stdint>
3 extern std::int32_t array1[]; // Non-compliant
4 extern std::int32_t array2[42]; // Compliant
```

## See also

- MISRA C++ 2008 [6]: Rule 3-1-3 When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.

### 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 [6]

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

See MISRA C++ 2008 [6]

**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 [6]

**Rule M3-2-4 (required, implementation, automated)**

**An identifier with external linkage shall have exactly one definition.**

See MISRA C++ 2008 [6]

### 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 271715 2017-03-23 10:13:51Z piotr.tanski $
2 #include <cstdint>
3 extern std::int32_t a1;
4 extern void f4();
5 namespace n
6 {
7 void f2();
8 std::int32_t a5; // Compliant, external linkage
9 }

1 // % $Id: A3-3-1.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
2 #include "A3-3-1.hpp"
3 std::int32_t a1 = 0; // Compliant, external linkage
4 std::int32_t a2 = 0; // Non-compliant, static keyword not used
```

```
5 static std::int32_t a3 = 0; // Compliant, internal linkage
6 namespace
7 {
8 std::int32_t a4 = 0; // Compliant by exception
9 void f1() // Compliant by exception
10 {
11 }
12 }
13 namespace n
14 {
15 void f2() // Compliant, external linkage
16 {
17 }
18 std::int32_t a6 = 0; // Non-compliant, external linkage
19 }
20 extern std::int32_t a7; // Non-compliant, extern object declared in .cpp file
21 static void f3() // Compliant, static keyword used
22 {
23 }
24 void f4() // Compliant, external linkage
25 {
26     a1 = 1;
27     a2 = 1;
28     a3 = 1;
29     a4 = 1;
30     n::a5 = 1;
31     n::a6 = 1;
32     a7 = 1;
33 }
34 void f5() // Non-compliant, static keyword not used
35 {
36     a1 = 2;
37     a2 = 2;
38     a3 = 2;
39     a4 = 2;
40     n::a5 = 2;
41     n::a6 = 2;
42     a7 = 2;
43 }
44 int main(int, char**) // Compliant by exception
45 {
46     f1();
47     n::f2();
48     f3();
49     f4();
50     f5();
51 }
```

## See also

- MISRA C++ 2008 [6]: Rule 3-3-1 Objects or functions with external linkage shall be declared in a header file.



**Rule A3-3-2 (required, implementation, automated)  
Non-POD type objects with static storage duration shall not be used.****Rationale**

Using global and static variables of a non-POD type makes the API of a class to be spurious about its true dependencies, as they can be accessed from any place of the source code. Using static or global variables makes the code more difficult to maintain, less readable and significantly less testable.

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

Note that the rule applies to:

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

**Exception**

Defining “static constexpr” variables of non-POD type is permitted.

**Example**

```
1 // $Id: A3-3-2.cpp 271927 2017-03-24 12:01:35Z piotr.tanski $
2 #include <cstdint>
3 #include <limits>
4 #include <string>
5 class A // Non-POD type
6 {
7     public:
8         static std::uint8_t instanceId; // Compliant - static variable of POD type
9         static float const pi; // Compliant - static variable of POD type
10        static std::string const
11            separator; // Non-compliant - static variable of nonPOD type
12
13        // Implementation
14 };
15 std::uint8_t A::instanceId = 0;
16 float const A::pi = 3.14159265359;
17 std::string const A::separator = "=====";
18
19 class B
20 {
21     public:
22        // Implementation
23
```

```
24     private:
25         static A a; // Non-compliant - static variable of non-POD type
26     };
27
28     class C
29     {
30     public:
31         constexpr C() = default;
32     };
33
34     namespace
35     {
36     constexpr std::int32_t maxInt32 =
37         std::numeric_limits<std::int32_t>::max(); // Compliant - static constexpr
38                                                     // variable of POD type
39
40     A instance{}; // Non-compliant - static variable of non-POD type
41
42     constexpr C
43         c{}; // Compliant by exception - constexpr static variable of non-POD type
44     } // namespace
45
46     void fn() noexcept
47     {
48         static A a{}; // Non-compliant
49         static std::int32_t counter{0}; // Compliant
50     }
51
52     class D // Singleton
53     {
54     public:
55         D() = default;
56         D(D const&) = default;
57         D(D&&) = default;
58         D& operator=(D const&) = default;
59         D& operator=(D&&) = default;
60         ~D() = default;
61
62     private:
63         static B* instance; // Compliant - static variable of non-POD type, because
64                             // it is a raw pointer
65     };
66     B* D::instance = nullptr;
```

## See also

- [HIC++ v4.0 \[8\]: 3.3.1 Do not use variables with static storage duration.](#)
- [Google C++ Style Guide \[11\]: 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 [6]

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 [6]

### 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 [6]

**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 271389 2017-03-21 14:41:05Z piotr.tanski $
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 [6]: 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 [6]

## 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 [6]

#### 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 [15].

## Example

```
1 // $Id: A4-5-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <stdint>
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     }
24 }
25 enum class Car : std::uint8_t
26 {
27     Modell1,
28     Model2,
29     Model3,
30     ModelsCount
31 };
32 void f2() noexcept(false)
33 {
34     Car car = Car::Modell1;
```

```
35     if (car != Car::Model2) // Compliant
36     {
37     }
38
39     if (car == Car::Model3) // Compliant
40     {
41     }
42
43     // if (car == (Car::Model1 + Car::Model2)) // Non-compliant -
44     // operator+ not provided for Car enum class, compilation error
45     //{
46     //}
47     if (car < Car::ModelsCount) // Compliant
48     {
49     }
50 }
51 Car operator+(Car lhs, Car rhs)
52 {
53     return Car::Model3;
54 }
55 void f3() noexcept(false)
56 {
57     Car car = Car::Model3;
58     if (car == (Car::Model1 + Car::Model2)) // Non-compliant - overloaded
59                                             // operator+ provided, no
60                                             // compilation error
61     {
62     }
63 }
64 enum Team : std::uint8_t
65 {
66     TeamMember1 = 0,
67     TeamMember2 = 1,
68     TeamMember3 = 2,
69     TeamMember4 = 3,
70     TeamMembersStart = TeamMember1,
71     TeamMembersEnd = TeamMember2,
72     TeamMembersCount = 4
73 };
74 void f4(const char* teamMember)
75 {
76     // Implementation
77 }
78 void f5()
79 {
80     const char* team[TeamMembersCount]; // Compliant
81     // ...
82     f4(team[TeamMember2]); // Compliant
83 }
```

## See also

- MISRA C++ 2008 [6]: 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 [6]

### 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

```

1 // $Id: A4-7-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <cstdint>
3 #include <stdexcept>
4 std::int8_t fn1(std::int8_t x, std::int8_t y) noexcept
5 {
6     return (x + y); // Non-compliant - may lead to overflow
7 }
8 std::int8_t fn2(std::int8_t x, std::int8_t y)
9 {
10    if (x > 100 || y > 100) // Range check
11    {
12        throw std::logic_error("Preconditions check error");
13    }
14    return (x + y); // Compliant - ranges of x and y checked before the
15                  // arithmetic operation
16 }
17 std::int16_t fn3(std::int8_t x, std::int8_t y) noexcept
18 {
19    return (static_cast<std::int16_t>(x) + y); // Compliant - std::int16_t type
20                                              // is enough for this arithmetic
21                                              // operation

```

```
22 }
23 std::uint8_t fn4(std::uint8_t x, std::uint8_t y) noexcept
24 {
25     return (x * y); // Non-compliant - may lead to wrap-around
26 }
27 std::int8_t fn5(std::int16_t x)
28 {
29     return static_cast<std::int8_t>(x); // Non-compliant - data loss
30 }
31 std::int8_t fn6(std::int16_t x)
32 {
33     return x; // Non-compliant - data loss by implicit conversion
34 }
35 void f()
36 {
37     std::int8_t x1 =
38         fn1(5, 10); // Compliant - overflow will not occur for these values
39     std::int8_t x2 = fn1(250, 250); // Non-compliant - Overflow occurs
40     try
41     {
42         std::int8_t x3 =
43             fn2(250, 250); // Compliant - No overflow, range checks
44                             // inside fn2() function
45     }
46     catch (std::logic_error&)
47     {
48         // Handle an error
49     }
50     std::int16_t x4 = fn3(250, 250); // Compliant - No overflow, arithmetic
51                                     // operation underlying type is wider than
52                                     // std::int8_t
53     std::uint8_t x5 = fn4(50, 10); // Non-compliant - Wrap-around occurs
54     std::int8_t x6 = fn5(100); // Compliant - data loss will not occur
55     std::int8_t x7 = fn5(300); // Non-compliant - Data loss occurs
56     std::int8_t x8 = fn6(300); // Non-compliant - Data loss occurs
57
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 [6]: Rule 5-0-6 An implicit integral or floating-point conversion shall not reduce the size of the underlying type.
- MISRA C++ 2008 [6]: 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 [8]: 4.2.2 Ensure that data loss does not demonstrably occur in an integral expression.
- C++ Core Guidelines [10]: 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 [6]

**Rule A4-10-1 (required, 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

```
1  /// $Id: A4-10-1.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
2  #include <cstdint>
3  #include <cstdint>
4
5  void f1(std::int32_t);
6  void f2(std::int32_t*);
7  void f3()
8  {
9      f1(0); // Compliant
10     f1(NULL); // Non-compliant - NULL used as an integer,
11     // compilable
12     // f1(nullptr); // Non-compliant - nullptr used as an integer
13     // compilation error
14     f2(0); // Non-compliant - 0 used as the null pointer constant
15     f2(NULL); // Non-compliant - NULL used as the null pointer constant
16     f2(nullptr); // Compliant
17 }
18 void f4(std::int32_t*);
```

```
19 template <class F, class A>
20 void f5(F f, A a)
21 {
22     f4(a);
23 }
24 void f6()
25 {
26     // f5(f4, NULL); // Non-compliant - function f4(std::int32_t) not declared
27     f5(f4, nullptr); // Compliant
28 }
```

## See also

- HIC++ v4.0 [8]: 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 [6]

## 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 271927 2017-03-24 12:01:35Z piotr.tanski $
2 #include <cstdlib>
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
33 // 3) Function pointers
34 // If a function is called via a function pointer there shall be no
35 // dependence
36 // on the order in which function-designator and function arguments are
37 // evaluated.
38 struct S
39 {
40     void taskStartFn(S* obj) noexcept(false);
41 };
42 void f4(S* p) noexcept(false)
43 {
44     p->taskStartFn(p++);
45 }
46
47 // 4) Function calls
48 // Functions may have additional effects when they are called (e.g. modifying
49 // some global data). Dependence on order of evaluation could be avoided by
50 // invoking the function prior to the expression that uses it, making use of a
51 // temporary variable for the value. For example:
52 extern std::uint16_t g(std::uint8_t) noexcept(false);
53 extern std::uint16_t z(std::uint8_t) noexcept(false);
54 void f5(std::uint8_t a) noexcept(false)
55 {
56     std::uint16_t x = g(a) + z(a);
```

```
57 }
58 // could be written as
59 void f6(std::uint8_t a) noexcept(false)
60 {
61     std::uint16_t x = g(a);
62     x += z(a);
63 }
64 // As an example of what can go wrong, consider an expression to take two values
65 // off a stack, subtract the second from the first, and push the result back on
66 // the stack:
67 std::int32_t pop(std::stack<std::int32_t>& s)
68 {
69     std::int32_t ret = s.top();
70     s.pop();
71     return ret;
72 }
73 void f7(std::stack<std::int32_t>& s)
74 {
75     s.push(pop(s) - pop(s));
76 }
77 // This will give different results depending on which of the pop() function
78 // calls is evaluated first (because pop() has side effects).
79
80 // 5) Nested assignment statements
81 // Assignments nested within expressions cause additional side effects. The best
82 // way to avoid any possibility of this leading to a dependence on order of
83 // evaluation is not to embed assignments within expressions. For example, the
84 // following is not recommended:
85 void f8(std::int32_t& x) noexcept(false)
86 {
87     std::int32_t y = 4;
88     x = y = y++; // It is undefined whether the final value of y is 4 or 5
89 }
90 // 6) Accessing a volatile
91 // The volatile type qualifier is provided in C++ to denote objects whose value
92 // can change independently of the execution of the program (for example an
93 // input register). If an object of volatile qualified type is accessed this may
94 // change its value. C++ compilers will not optimize out reads of a volatile. In
95 // addition, as far as a C++ program is concerned, a read of a volatile has a
96 // side effect (changing the value of the volatile). It will usually be
97 // necessary to access volatile data as part of an expression, which then means
98 // there may be dependence on order of evaluation. Where possible, though, it is
99 // recommended that volatiles only be accessed in simple assignment statements,
100 // such as the following:
101 void f9(std::uint16_t& x) noexcept(false)
102 {
103     volatile std::uint16_t v;
104     // ...
105     x = v;
106 }
107
```

```
108 // The rule addresses the order of evaluation problem with side effects. Note
109 // that there may also be an issue with the number of times a sub-expression is
110 // evaluated, which is not covered by this rule. This can be a problem with
111 // function invocations where the function is implemented as a macro. For
112 // example, consider the following function-like macro and its invocation:
113 #define MAX(a, b) ((a) > (b)) ? (a) : (b)
114 // ...
115 void f10(std::uint32_t& i, std::uint32_t j)
116 {
117     std::uint32_t z = MAX(i++, j);
118 }
119 // The definition evaluates the first parameter twice if a > b but only once if
120 // a = b. The macro invocation may thus increment i either once or twice,
121 // depending on the values of i and j.
122 // It should be noted that magnitude-dependent effects, such as those due to
123 // floating-point rounding, are also not addressed by this rule. Although
124 // the
125 // order in which side effects occur is undefined, the result of an operation is
126 // otherwise well-defined and is controlled by the structure of the expression.
127 // In the following example, f1 and f2 are floating-point variables; F3, F4
128 // and
129 // F5 denote expressions with floating-point types.
130
131 // f1 = F3 + ( F4 + F5 );
132 // f2 = ( F3 + F4 ) + F5;
133
134 // The addition operations are, or at least appear to be, performed in the order
135 // determined by the position of the parentheses, i.e. firstly F4 is added to F5
136 // then secondly F3 is added to give the value of f1. Provided that F3, F4 and
137 // F5 contain no side effects, their values are independent of the order in
138 // which they are evaluated. However, the values assigned to f1 and f2 are not
139 // guaranteed to be the same because floating-point rounding following the
140 // addition operations are dependent on the values being added.
```

## See also

- MISRA C++ 2008 [6]: Rule 5-0-1 The value of an expression shall be the same under any order of evaluation that the standard permits

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

See MISRA C++ 2008 [6]

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

See MISRA C++ 2008 [6]

**Rule M5-0-4 (required, implementation, automated)**

An implicit integral conversion shall not change the signedness of the underlying type.

See MISRA C++ 2008 [6]

**Rule M5-0-5 (required, implementation, automated)**

There shall be no implicit floating-integral conversions.

See MISRA C++ 2008 [6]

**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 [6]

**Rule M5-0-7 (required, implementation, automated)**

There shall be no explicit floating-integral conversions of a cvalue expression.

See MISRA C++ 2008 [6]

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

```
1 void fn() noexcept
2 {
3     float f = -4.5;
4     std::int8_t x1 = static_cast<std::int8_t>(f); // Compliant, x1 = -4
5     std::int8_t x2 =
6         static_cast<std::int8_t>(std::floor(f)); // Compliant, x2 = -5
7     std::int8_t x3 =
8         static_cast<std::int8_t>(std::ceil(f)); // Compliant, x3 = -4
9 }
```

**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 [6]

**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 [6]

**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 [6]

**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 [6]

**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 [6]

**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 271927 2017-03-24 12:01:35Z piotr.tanski $
2 #include <stdint>
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 [6]: 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 [6]

#### **Rule M5-0-15 (required, implementation, automated)**

**Array indexing shall be the only form of pointer arithmetic.**

See MISRA C++ 2008 [6]

#### **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 [6]

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 [6]

#### **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 [6]

**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 271687 2017-03-23 08:57:35Z piotr.tanski $
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 [6]: 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 [6]

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

See MISRA C++ 2008 [6]

### 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

```
1 // $Id: A5-1-1.cpp 271929 2017-03-24 12:05:53Z piotr.tanski $
2 #include <array>
3 #include <stdint>
4 #include <iostream>
5 #include <stdexcept>
6 namespace
7 {
8     const std::int32_t maxIterations = 10;    // Compliant - assignment
9     const char* const loopIterStr = "iter "; // Compliant - assignment
10    const char separator = ':';              // Compliant - assignment
11 }
12 void f1() noexcept
13 {
14     for (std::int32_t i = 0; i < 10; ++i) // Non-compliant
15     {
16         std::cout << "iter " << i << ':' << '\n'; // Compliant by exception
17     }
18
19     for (std::int32_t i = 0; i < maxIterations; ++i) // Compliant
20     {
21         std::cout << loopIterStr << i << separator << '\n'; // Compliant
22     }
23
24     for (std::int32_t i = 0; i < maxIterations; ++i) // Compliant
25     {
26         std::cout << "iter " << i << ':' << '\n'; // Compliant by exception
27     }
28 }
29 void f2()
30 {
31     // ...
32     throw std::logic_error("Logic Error"); // Compliant
33     // initialization of exception object
34 }
35 class C
36 {
37     public:
38     C() : x(0), y(nullptr) // Compliant - initialization
39     {
40     }
41     C(std::int8_t num, std::int32_t* ptr) : x(num), y(ptr) {}
42
43     private:
44     std::int8_t x;
```

```
45     std::int32_t* y;
46 };
47 static std::int32_t* globalPointer = nullptr; // Compliant - assignment
48 void f3() noexcept
49 {
50     C c1;
51     // ...
52     C c2(0, globalPointer); // Compliant - initialization of C object
53 }
54 std::int32_t f4(std::int32_t x, std::int32_t y) noexcept
55 {
56     return x + y;
57 }
58 void f5() noexcept
59 {
60     std::int32_t ret = f4(2, 5); // Non-compliant
61     // ...
62     std::int32_t x = 2;
63     std::int32_t y = 5;
64     ret = f4(x, y); // Compliant
65
66     std::array<std::int8_t, 5> arr{{1, 2, 3, 4, 5}}; // Compliant
67 }
```

## See also

- HIC++ v4.0 [8]: 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

```
1 // $Id: A5-1-2.cpp 271752 2017-03-23 12:07:07Z piotr.tanski $
2 #include <algorithm>
3 #include <cstdlib>
4 #include <vector>
5 void fn1(const std::vector<std::int32_t>& v)
6 {
7     std::uint64_t sum = 0;
8     std::for_each(v.begin(), v.end(), [&](std::int32_t lhs) {
9         sum += lhs;
```

```
10     }); // Non-compliant
11
12     sum = 0;
13     std::for_each(v.begin(), v.end(), [&sum](std::int32_t lhs) {
14         sum += lhs;
15     }); // Compliant
16 }
17 void fn2()
18 {
19     constexpr std::uint8_t n = 10;
20     static std::int32_t j = 0;
21     [n]() {
22         std::int32_t array[n]; // Compliant
23         j += 1;                // Compliant by exception
24     };
25 }
```

### See also

- HIC++ v4.0 [8]: 5.1.4 Do not capture variables implicitly in a lambda.

**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

```
1 // $Id: A5-1-3.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <cstdlib>
3 void fn()
4 {
5     std::int32_t i = 0;
6     std::int32_t j = 0;
7     auto lambda1 = [&i, &j] { ++i, ++j; }; // Non-compliant
8     auto lambda2 = [&i, &j]() {
9         ++i;
10        ++j;
11    }; // Compliant
12 }
```

### See also

- HIC++ v4.0 [8]: 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

```
1 // $Id: A5-1-4.cpp 272338 2017-03-28 08:15:01Z piotr.tanski $
2 #include <cstdint>
3 #include <functional>
4 std::function<std::int32_t()> f()
5 {
6     std::int32_t i = 12;
7     return ([&i]() -> std::int32_t {
8         i = 100;
9         return i;
10    }); // Non-compliant
11 }
12 std::function<std::int32_t()> g()
13 {
14     std::int32_t i = 12;
15     return ([i]() mutable -> std::int32_t { return ++i; }); // Compliant
16 }
17 void fn()
18 {
19     auto lambda1 = f();
20     std::int32_t i = lambda1(); // Undefined behavior
21     auto lambda2 = g();
22     i = lambda2(); // lambda2() returns 13
23 }
```

### See also

- SEI CERT C++ [9]: EXP61-CPP. A lambda object must not outlive any of its reference captured objects.

**Rule A5-1-5 (advisory, implementation, non-automated)**  
**If a lambda expression is used in the same scope in which it has been defined, the lambda should capture objects by reference.**

### Rationale

Copying objects captured to lambda by value may be a performance overhead. It is correct to capture objects by reference when using the lambda expression locally only.

## Exception

It is permitted to capture by copy objects that size is lesser or equal to 16 bytes (i.e.

`4 * sizeof(std::uint32_t)`).

## Example

```
1 // $Id: A5-1-5.cpp 272338 2017-03-28 08:15:01Z piotr.tanski $
2 #include <cstdint>
3 #include <functional>
4 namespace
5 {
6 constexpr std::int32_t bufferMax = 1024;
7 constexpr std::int8_t receiversMax = 10;
8 }
9 class UDPClient
10 {
11     // Implementation - size of UDPClient class exceeds 16 bytes
12 };
13 void f1() noexcept(false)
14 {
15     UDPClient client;
16     std::uint8_t buffer[bufferMax];
17     auto lambda1 = [client,
18                   buffer]() // Non-compliant - it is inefficient to capture
19                           // UDPClient and buffer objects by copy in lambda
20     {
21         // Code
22     };
23     lambda1(); // lambda1 used locally only
24
25     auto lambda2 =
26         [&client, &buffer]() // Compliant - be aware that this construct
27         // may introduce data races in parallel calls.
28     {
29         // Code
30     };
31     lambda2(); // lambda2 used locally only
32
33     std::uint32_t number1 = 10;
34     std::uint32_t number2 = 20;
35     auto lambda3 = [number1, number2]() // Compliant by exception - the size of
36                                       // std::uint32_t is 4 bytes (lesser or
37                                       // equal to the size of a pointer -
38                                       // depending on architecture)
39     {
40         // Code
41     };
42     lambda3(); // lambda3 used locally only
43 }
44 void f2(std::int8_t currentReceiver) noexcept(false)
45 {
46     std::function<void()> receiver;
```



```

47
48     if (currentReceiver < receiversMax)
49     {
50         UDPClient client;
51         receiver =
52             [&client]() // Non-compliant - lambda is not used locally, client
53                         // object will go out of scope
54             {
55                 // Code
56             };
57     }
58
59     // ...
60     receiver(); // Undefined behavior, client object went out of scope
61 }
  
```

### See also

- C++ Core Guidelines [10]: F.52: Prefer capturing by reference in lambdas that will be used locally, including passed to algorithms captured objects.

**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

```

1 // $Id: A5-1-6.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <stdint>
3 void fn() noexcept
4 {
5     auto lambda1 = []() -> std::uint8_t {
6         std::uint8_t ret = 0U;
7         // ...
8         return ret;
9     }; // Compliant
10    auto lambda2 = []() {
11        // ...
12        return 0U;
13    }; // Non-compliant - returned type is not specified
14    auto x = lambda1(); // Type of x is std::uint8_t
15    auto y = lambda2(); // What is the type of y?
16 }
  
```

## See also

- none

**Rule A5-1-7 (required, implementation, automated)**  
**The underlying type of lambda expression shall not be used.**

## 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 either as function argument, template argument or operand to built-in or overloaded operator.

## Example

```

1 // $Id: A5-1-7.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <cstdint>
3 #include <functional>
4 #include <vector>
5 void fn()
6 {
7     auto lambda1 = []() -> std::int8_t { return 1; };
8     auto lambda2 = []() -> std::int8_t { return 1; };
9
10    if (typeid(lambda1) == typeid(lambda2)) // Non-compliant - types of lambda1
11                                           // and lambda2 are different
12    {
13        // ...
14    }
15
16    std::vector<decltype(lambda1)> v; // Non-compliant
17    // v.push_back([]() { return 1; }); // Compilation error, type of pushed
18    // lambda is different than decltype(lambda1)
19    // using mylambda_t = decltype([](){ return 1; }); // Non-compliant -
20    // compilation error
21    auto lambda3 = []() { return 2; };
22    using lambda3_t = decltype(lambda3); // Non-compliant - lambda3_t type can
23                                         // not be used for lambda expression
24                                         // declarations
25    // lambda3_t lambda4 = []() { return 2; }; // Conversion error at
26    // compile-time
27    std::function<std::int32_t()> f1 = []() { return 3; };
28    std::function<std::int32_t()> f2 = []() { return 3; };
29
30    if (typeid(f1) == typeid(f2)) // Compliant - types are equal
31    {
32        // ...
33    }
34 }

```

## 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

```
1 // $Id: A5-1-8.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <cstdlib>
3 void fn1()
4 {
5     std::int16_t x = 0;
6     auto f1 = [&x]() {
7
8         auto f2 = []() {}; // Non-compliant
9         f2();
10
11        auto f4 = []() {}; // Non-compliant
12        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

## 6.5.2 Postfix expressions

**Rule M5-2-1 (required, implementation, automated)**  
**Each operand of a logical `&&`, `||` shall be a postfix expression.**

See MISRA C++ 2008 [6]

**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 [6]

**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 [6]

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 271715 2017-03-23 10:13:51Z piotr.tanski $
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

- C++ Core Guidelines [10]: 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 271927 2017-03-24 12:01:35Z piotr.tanski $
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 }
21 void f() noexcept(false)
22 {
23     C a1 = C{10};           // Compliant
24     C* a2 = (C*)(&a1);     // Non-compliant
25     const C a3(5);
26     C* a4 = const_cast<C*>(&a3); // Compliant - violates another rule
27     E* d1 = reinterpret_cast<E*>(a4); // Compliant - violates another rule
28     D* d2 = dynamic_cast<D*>(a2); // Compliant - violates another rule
29     std::int16_t x1 = 20;
30     std::int32_t x2 = static_cast<std::int32_t>(x1); // Compliant
31     std::int32_t x3 = (std::int32_t)x1; // Non-compliant
32     std::int32_t x4 = 10;
33     float f1 = static_cast<float>(x4); // Compliant
34     float f2 = (float)x4; // Non-compliant
35     std::int32_t x5 = static_cast<std::int32_t>(f1); // Compliant
36     std::int32_t x6 = (std::int32_t)f1; // Non-compliant
37     (void)g(); // Non-compliant
38     static_cast<void>(g()); // Compliant
39 }

```

## See also

- MISRA C++ 2008 [6]: 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 [7]: 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

```
1 // $Id: A5-2-3.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
```

```
2 #include <cstdint>
3 void f1(const char* str) noexcept(false)
4 {
5     *(const_cast<char*>(str)) =
6         '\0'; // Non-compliant - const qualification removed
7 }
8 class C
9 {
10 public:
11     explicit C(std::int32_t) {}
12 };
13 void f2() noexcept(false)
14 {
15     C const a1 = C(10);
16     C* a2 = const_cast<C*>(&a1); // Non-compliant - const qualification removed
17     C* a3 = (C*)&a1; // Non-compliant - const qualification removed
18 }
19 extern volatile std::int32_t* f3() noexcept;
20 void f4() noexcept
21 {
22     volatile std::int32_t* ptr1 = f3();
23     // ...
24     std::int32_t* ptr2 = const_cast<std::int32_t*>(
25         ptr1); // Non-compliant - volatile qualification removed
26     // ...
27     std::int32_t* ptr3 =
28         (std::int32_t*)ptr1; // Non-compliant - volatile qualification removed
29 }
```

## See also

- MISRA C++ 2008 [6]: 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 [6]

### 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

```

1 // $Id: A5-2-4.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
2 #include <cstdint>
3 #include <string>
4 void f1() noexcept
5 {
6     std::string str = "Hello";
7     std::int32_t* ptr = reinterpret_cast<std::int32_t*>(&str); // Non-compliant
8 }
9 struct A
10 {
11     std::int32_t x;
12     std::int32_t y;
13 };
14 class B
15 {
16     public:
17     virtual ~B() {}
18
19     private:
20     std::int32_t x;
21 };
22 class C : public B
23 {
24 };
25 class D : public B
26 {
27 };
28 void f2(A* ptr) noexcept
29 {
30     B* b1 = reinterpret_cast<B*>(ptr); // Non-compliant
31     std::int32_t num = 0;
32     A* a1 = reinterpret_cast<A*>(num); // Non-compliant
33     A* a2 = (A*)
34         num; // Compliant with this rule, but non-compliant with Rule A5-2-2.
35     B* b2 = reinterpret_cast<B*>(num); // Non-compliant
36     D d;
37     C* c1 = reinterpret_cast<C*>(&d); // Non-compliant - cross cast
38     C* c2 = (C*)&d; // Compliant with this rule, but non-compliant with Rule
39                 // A5-2-2. Cross-cast.
40     B* b3 = &d; // Compliant - class D is a subclass of class B
41 }

```

## See also

- MISRA C++ 2008 [6]: 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 [10]: Type.1: Don't use `reinterpret_cast`.



**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 [6]

**Rule M5-2-9 (required, implementation, automated)**

**A cast shall not convert a pointer type to an integral type.**

See MISRA C++ 2008 [6]

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 [6]

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 [6]

**Rule A5-2-5 (required, implementation, automated)**

**An array shall not be accessed beyond its range.**

## Rationale

To avoid undefined behavior, range checks should be coded to ensure that the array access via pointer arithmetic or subscript operator is within defined bounds.

This could be also achieved by accessing an array via subscript operator with constant indexes only.

Note that this rule applies to C-style arrays and all other containers that access their elements using input index without range-checks.

Also, note that calculating an address one past the last element of the array is well defined, but dereferencing it is not.

## Example

```
1 // $Id: A5-2-5.cpp 271752 2017-03-23 12:07:07Z piotr.tanski $
2 #include <array>
```

```
3 #include <stdint>
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 [8]: 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 [6]

### 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 [6]

**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 [6]

**Rule M5-3-3 (required, implementation, automated)**

The unary `&` operator shall not be overloaded.

See MISRA C++ 2008 [6]

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

See MISRA C++ 2008 [6]

**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 271696 2017-03-23 09:23:09Z piotr.tanski $
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 [8]: 5.1.6 Do not code side effects into the right-hand operands of: `&&`, `||`, `sizeof`, `typeid` or a function passed to `condition_variable::wait`.

## 6.5.6 Multiplicative operators

**Rule A5-5-1 (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: A5-5-1.cpp 272338 2017-03-28 08:15:01Z piotr.tanski $
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 [8]: 5.5.1 Ensure that the right hand operand of the division or remainder operators is demonstrably non-zero.
- C++ Core Guidelines [10]: 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 [6]

## 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 271927 2017-03-24 12:01:35Z piotr.tanski $
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 [8]: 5.7.2 Ensure that a pointer to member that is a virtual function is only compared (`==`) with `nullptr`.
- JSF December 2005 [7]: 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 [6]

## 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 271927 2017-03-24 12:01:35Z piotr.tanski $
2 #include <stdint>
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 [8]: 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 [6]

## 6.5.19 Comma operator

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

See MISRA C++ 2008 [6]

### 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 [6]

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 [6]

#### 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 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 [6]

**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 [6]



### 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 [6]

### 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 [6]

**Rule M6-4-2 (required, implementation, automated)**

All if ... else if constructs shall be terminated with an else clause.

See MISRA C++ 2008 [6]

**Rule M6-4-3 (required, implementation, automated)**

A switch statement shall be a well-formed switch statement.

See MISRA C++ 2008 [6]

**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 [6]

**Rule M6-4-5 (required, implementation, automated)**

An unconditional throw or break statement shall terminate every non-empty switch-clause.

See MISRA C++ 2008 [6]

**Rule M6-4-6 (required, implementation, automated)**

The final clause of a switch statement shall be the default-clause.

See MISRA C++ 2008 [6]

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

See MISRA C++ 2008 [6]

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 271687 2017-03-23 08:57:35Z piotr.tanski $
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 [6]: Rule 6-4-8 Every switch statement shall have at least one case-clause.
- HIC++ v4.0 [8]: 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 271752 2017-03-23 12:07:07Z piotr.tanski $
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 through 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 [8]: 6.2.1 Implement a loop that only uses element values as a range-based loop.
- C++ Core Guidelines [10]: 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

```
1 // $Id: A6-5-2.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
2 #include <stdint>
3 namespace
4 {
5 constexpr std::int32_t xlimit = 20;
6 constexpr std::int32_t ylimit = 15;
7 constexpr float zlimit = 2.5F;
8 constexpr std::int32_t ilimit = 100;
9 }
10 void fn() noexcept
11 {
12     std::int32_t y = 0;
13
14     for (std::int32_t x = 0; x < xlimit && y < ylimit;
15         x++, y++) // Non-compliant, two loop-counters
16     {
17         // ...
18     }
19
20     for (float z = 0.0F; z != zlimit;
21         z += 0.1F) // Non-compliant, float with !=
22     {
23         // ...
24     }
25
26     for (float z = 0.0F; z < zlimit; z += 0.1F) // Non-compliant, float with <
27     {
28         // ...
29     }
30
31     for (std::int32_t i = 0; i < ilimit; ++i) // Compliant
32     {
33         // ...
34     }
35 }
```

## See also

- MISRA C++ 2008 [6]: Rule 6-5-1 A for loop shall contain a single loop-counter which shall not have floating type.

**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 [6]

**Rule M6-5-3 (required, implementation, automated)**

The loop-counter shall not be modified within condition or statement.

See MISRA C++ 2008 [6]

**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 [6]

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 [6]

**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 [6]

## 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

```
1 // $Id: A6-6-1.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
2 #include <cstdlib>
3 namespace
4 {
5 constexpr std::int32_t loopLimit = 100;
6 }
7 void f1(std::int32_t n) noexcept
8 {
9     if (n < 0)
10    {
11        // goto exit; // Non-compliant - jumping to exit from here crosses ptr
12        // pointer initialization, compilation
13        // error
14    }
15
16    std::int32_t* ptr = new std::int32_t(n);
17 // ...
18 exit:
19    delete ptr;
20 }
21 void f2() noexcept
22 {
23    // ...
24    goto error; // Non-compliant
25 // ...
26 error;; // Error handling and cleanup
27 }
28 void f3() noexcept
29 {
30    for (std::int32_t i = 0; i < loopLimit; ++i)
31    {
32        for (std::int32_t j = 0; j < loopLimit; ++j)
33        {
34            for (std::int32_t k = 0; k < loopLimit; ++k)
35            {
36                if ((i == j) && (j == k))
37                {
38                    // ...
39                    goto loop_break; // Non-compliant
40                }
41            }
42        }
43    }
44
45 loop_break;; // ...
46 }
```

## See also

- JSF December 2005 [7]: AV Rule 189 The goto statement shall not be used.
- C++ Core Guidelines [10]: ES.76: Avoid goto.

- C++ Core Guidelines [10]: 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 [6]

**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 [6]

**Rule M6-6-3 (required, implementation, automated)**

The continue statement shall only be used within a well-formed for loop.

See MISRA C++ 2008 [6]

## 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

```
1 //% $Id: A7-1-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <stdint>
3 #include <limits>
4 void fn()
5 {
6     const std::int16_t x1 = 5;           // Compliant
7     constexpr std::int16_t x2 = 5;     // Compliant
8     std::int16_t x3 =
9         5; // Non-compliant - x3 is not modified but not declared as
10           // constant (const or constexpr)
11 }
```



## See also

- C++ Core Guidelines [10]: ES.25: Declare objects const or constexpr unless you want to modify its value later on.

### 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

```
1 //% $Id: A7-1-2.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
2 #include <cstdint>
3 std::int32_t pow1(std::int32_t number)
4 {
5     return (number * number);
6 }
7 constexpr std::int32_t pow2(
8     std::int32_t number) // Possible compile-time computing
9                          // because of constexpr specifier
10 {
11     return (number * number);
12 }
13 void fn()
14 {
15     constexpr std::int16_t i1 = 20; // Compliant, evaluated at compile-time
16     const std::int16_t i2 = 20; // Non-compliant, possible run-time evaluation
17     std::int32_t twoSquare =
18         pow1(2); // Non-compliant, possible run-time evaluation
19     const std::int32_t threeSquare =
20         pow1(3); // Non-compliant, possible run-time evaluation
21     // static_assert(threeSquare == 9, "pow1(3) did not succeed."); // Value
22     // can not be static_assert-ed
23     constexpr std::int32_t fiveSquare =
24         pow2(5); // Compliant, evaluated at compile time
25     static_assert(fiveSquare == 25,
26                 "pow2(5) did not succeed."); // Compliant, constexpr
27                                               // evaluated at compile time
28     // constexpr std::int32_t int32Max =
```

```

29     // std::numeric_limits<std::int32_t>::max() + 1; //
30     // Compliant - compilation error due to
31     // compile-time evaluation (integer overflow)
32 }
33 class A
34 {
35     public:
36         static constexpr double pi = 3.14159265; // Compliant - value of PI can be
37                                                     // determined in compile time
38
39         // constexpr double e = 2.71828182; // Non-compliant - constexprs need
40         // to be static members, compilation error
41
42     constexpr A() = default; // Compliant
43 };
  
```

## See also

- C++ Core Guidelines [10]: 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 [6]

### 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

```

1 // $Id: A7-1-3.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <cstdint>
3 using IntPtr = std::int32_t*;
4 using IntConstPtr = std::int32_t* const;
5 using ConstIntPtr = const std::int32_t*;
6 void fn(const std::uint8_t& input) // Compliant
7 {
8     std::int32_t value1 = 10;
9     std::int32_t value2 = 20;
10
11     const IntPtr ptr1 =
  
```

```

12     &value1; // Non-compliant - deduced type is std::int32_t*
13           // const, not const std::int32_t*
14
15 // ptr1 = &value2; // Compilation error, ptr1 is read-only variable
16
17 IntPtr const ptr2 =
18     &value1; // Compliant - deduced type is std::int32_t* const
19
20 // ptr2 = &value2; // Compilation error, ptr2 is read-only variable
21
22 IntPtrConst ptr3 = &value1; // Compliant - type is std::int32_t* const, no
23           // additional qualifiers needed
24
25 // ptr3 = &value2; // Compilation error, ptr3 is read-only variable
26
27 IntPtrConst ptr4 = &value1; // Compliant - type is const std::int32_t*
28
29 const IntPtrConst ptr5 = &value1; // Non-compliant, type is const
30           // std::int32_t* const, not const const
31           // std::int32_t*
32 IntPtrConst const ptr6 =
33     &value1; // Compliant - type is const std::int32_t* const
34 }

```

## See also

- HIC++ v4.0 [8]: 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

```

1 // $Id: A7-1-4.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <cstdint>
3 std::int32_t f1(register std::int16_t number) noexcept // Non-compliant
4 {
5     return ((number * number) + number);
6 }
7 void f2(std::int16_t number) noexcept // Compliant
8 {
9     register std::int8_t x = 10; // Non-compliant
10    std::int32_t result = f1(number); // Compliant

```

```
11     // ...
12 }
```

## See also

- JSF December 2005 [7]: AV Rule 140 The register storage class specifier shall not be used.
- HIC++ v4.0 [8]: 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 272338 2017-03-28 08:15:01Z piotr.tanski $
2 #include <stdint>
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 [8]: 7.1.8 Use auto id = expr when declaring a variable to have the same type as its initializer function call.
- C++ Core Guidelines [10]: Use auto.
- Google C++ Style Guide [11]: 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 <stdint>
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 [10]: T.43: Prefer using over typedef for defining aliases
- C++ Standard Core Language Active Issues, Revision 96 [17]: 1554. Access and alias templates.
- C++ Standard Core Language Defect Reports and Accepted Issues, Revision 96 [17]: 1558. Unused arguments in alias template specializations.

**Rule A7-1-7 (required, implementation, automated)**  
**Each identifier shall be declared 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 on the same line may become confusing.

### Exception

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

### Example

```
1 // $Id: A7-1-7.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
2 #include <stdint>
3 #include <vector>
4
5 typedef std::int32_t* ptr; // Compliant
6 typedef std::int32_t *pointer, value; // Non-compliant
```

```
7
8 void fn1() noexcept
9 {
10     std::int32_t x = 0;           // Compliant
11     std::int32_t y = 7, *p1 = nullptr; // Non-compliant
12     std::int32_t const *p2, z = 1; // Non-compliant
13 }
14
15 void fn2()
16 {
17     std::vector<std::int32_t> v{1, 2, 3, 4, 5};
18     for (auto iter{v.begin()}, end{v.end()}; iter != end;
19         ++iter) // Compliant by exception
20     {
21         // ...
22     }
23 }
```

### See also

- HIC++ v4.0 [8]: 7.1.1 Declare each identifier on a separate line in a separate declaration.
- JSF December 2005 [7]: AV Rule 42 Each expression-statement will be on a separate line.
- C++ Core Guidelines [10]: 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 271715 2017-03-23 10:13:51Z piotr.tanski $
2 #include <stdint>
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 [8]: 7.1.3 Do not place type specifiers before non-type specifiers in a declaration.

## 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 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <stdint>
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 [6]: 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

```
1 // $Id: A7-2-2.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
2 #include <cstdint>
3 enum class E1 // Non-compliant
4 {
5     E10,
6     E11,
7     E12
8 };
9 enum class E2 : std::uint8_t // Compliant
10 {
11     E20,
12     E21,
13     E22
14 };
15 enum E3 // Non-compliant
16 {
17     E30,
18     E31,
19     E32
20 };
21 enum E4 : std::uint8_t // Compliant - violating another rule
22 {
23     E40,
24     E41,
25     E42
26 };
27 enum class E5 : std::uint8_t // Non-compliant - will not compile
28 {
29     E50 = 255,
30     // E5_1, // E5_1 = 256 which is outside of range of underlying type
31     // std::uint8_t
32     // - compilation error
33     // E5_2 // E5_2 = 257 which is outside of range of underlying type
34     // std::uint8_t
35     // - compilation error
36 };
```

## See also

- HIC++ v4.0 [8]: 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 271715 2017-03-23 10:13:51Z piotr.tanski $
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 [10]: 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 [6]: 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 [8]: 7.2.2 Initialize none, the first only or all enumerators in an enumeration.

### 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 [6]

**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 [6]

**Rule M7-3-3 (required, implementation, automated)**

There shall be no unnamed namespaces in header files.

See MISRA C++ 2008 [6]

**Rule M7-3-4 (required, implementation, automated)**

Using-directives shall not be used.

See MISRA C++ 2008 [6]

See: Using-directive [15] concerns an inclusion of specific namespace with all its types, e.g. using namespace std.

**Rule M7-3-5 (required, implementation, automated)**

Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier.

See MISRA C++ 2008 [6]

**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 [6]

See: Using-declaration [15] 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

```
1 // $Id: A7-4-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <stdint>
3 std::int32_t fn1(std::int32_t b) noexcept
4 {
5     std::int32_t ret = 0;
6     // ...
7     asm("pushq %%rax \n"
8         "movl %0, %%eax \n"
9         "addl %1, %%eax \n"
10        "movl %%eax, %0 \n"
11        "popq %%rax"
12        : "=r"(ret)
13        : "r"(b)); // Non-compliant
14    return ret;
15 }
16 std::int32_t fn2(std::int32_t b) noexcept
17 {
18     std::int32_t ret = 0;
19     // ...
20     ret += b; // Compliant - equivalent to asm(...) above
21     return ret;
22 }
```

### See also

- HIC++ v4.0 [8]: 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 [6]

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

See MISRA C++ 2008 [6]

**Rule M7-4-3 (required, implementation, automated)**  
**Assembly language shall be encapsulated and isolated.**

See MISRA C++ 2008 [6]

## 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 [6]

**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 [6]

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.

**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

```
1 // $Id: A7-5-1.cpp 271927 2017-03-24 12:01:35Z piotr.tanski $
2 #include <stdint>
3 class A
4 {
5     public:
6         explicit A(std::uint8_t n) : number(n) {}
7         ~A() { number = 0U; }
8         // Implementation
9
10    private:
```

```
11     std::uint8_t number;
12 };
13 const A& fn1(const A& ref) noexcept // Non-compliant - the function returns a
14                                     // reference to const reference parameter
15                                     // which may bind to temporary objects.
16                                     // According to C++14 Language Standard, it
17 // is undefined whether a temporary object is introduced for const
18 // reference
19 // parameter
20 {
21     // ...
22     return ref;
23 }
24 const A& fn2(A& ref) noexcept // Compliant - non-const reference parameter does
25                               // not bind to temporary objects, it is allowed
26                               // that the function returns a reference to such
27                               // a parameter
28 {
29     // ...
30     return ref;
31 }
32 const A* fn3(const A& ref) noexcept // Non-compliant - the function returns a
33 // pointer to const reference parameter
34 // which may bind to temporary objects.
35 // According to C++14 Language Standard, it
36 // is undefined whether a temporary object is introduced for const
37 // reference
38 // parameter
39 {
40     // ...
41     return &ref;
42 }
43 template <typename T>
44 T& fn4(T& v) // Compliant - the function will not bind to temporary objects
45 {
46     // ...
47     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 destroyed
68 // object
69 // const A& ref6 = fn4(A{20}); // Compilation error - invalid
70 // initialization of non-const reference
71 }
```

## See also

- MISRA C++ 2008 [6]: 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 271687 2017-03-23 08:57:35Z piotr.tanski $
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
52                                 // arguments are known during compile time
53 }
54 std::int32_t fn6() noexcept
55 {
56     std::int32_t sum = fn5<std::int32_t, std::uint8_t, float, double>(
57         10, 5, 2.5, 3.5); // An example call to variadic template function
58     // ...
59     return sum;
60 }
61 constexpr std::int32_t fn7(std::int32_t x, std::int8_t n)
62 {
63     if (n >= 0)
64     {
65         x += x;
66         return fn5(x, --n); // Compliant by exception - recursion evaluated at
67                             // compile time

```

```
68     }  
69     return x;  
70 }
```

## See also

- MISRA C++ 2008 [6]: Rule 7-5-4 Functions should not call themselves, either directly or indirectly.
- JSF December 2005 [7]: AV Rule 119 Functions shall not call themselves, either directly or indirectly (i.e. recursion shall not be allowed).
- HIC++ v4.0 [8]: 5.2.2 Ensure that functions do not call themselves, either directly or indirectly.

## 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 [6]

### 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

```
1 // $Id: A8-2-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $  
2 #include <stdint>  
3 template <typename T>  
4 class A  
5 {  
6     public:  
7         using Type = std::int32_t;  
8  
9         Type f(T const&) noexcept;
```

```
10     Type g(T const&) noexcept;
11 };
12 template <typename T>
13 typename A<T>::Type A<T>::f(T const&) noexcept // Non-compliant
14 {
15     // Implementation
16 }
17 template <typename T>
18 auto A<T>::g(T const&) noexcept -> Type // Compliant
19 {
20     // Implementation
21 }
```

## See also

- HIC++ v4.0 [8]: 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 [6]

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 271687 2017-03-23 08:57:35Z piotr.tanski $
```

```
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 [6]: Rule 8-4-1 Functions shall not be defined using the ellipsis notation.
- HIC++ v4.0 [8]: 14.1.1 Use variadic templates rather than an ellipsis.
- C++ Core Guidelines [10]: Type.8: Avoid reading from varargs or passing vararg arguments. Prefer variadic template parameters instead.

#### **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 [6]

#### **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

```
1 // $Id: A8-4-2.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <cstdlib>
3 #include <stdexcept>
4 std::int32_t f1() noexcept // Non-compliant
5 {
6 }
7 std::int32_t f2(std::int32_t x) noexcept(false)
8 {
```

```
9     if (x > 100)
10    {
11        throw std::logic_error("Logic Error"); // Compliant by exception
12    }
13
14    return x; // Compliant
15 }
16 std::int32_t f3(std::int32_t x, std::int32_t y)
17 {
18     if (x > 100 || y > 100)
19     {
20         throw std::logic_error("Logic Error"); // Compliant by exception
21     }
22     if (y > x)
23     {
24         return (y - x); // Compliant
25     }
26     return (x - y); // Compliant
27 }
```

## See also

- MISRA C++ 2008 [6]: 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++ [9]: 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 [6]

## 6.8.5 Initalizers

### **Rule M8-5-1 (required, implementation, automated)**

**All variables shall have a defined value before they are used.**

See MISRA C++ 2008 [6]

### **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 {
6 };
7 class B
8 {
9 };
10 class C : public virtual B, public A
11 {
12     public:
13         C() : B(), A(), s() {} // Compliant
14
15         // C() : A(), B() { } // Non-compliant - incorrect order of initialization
16
17     private:
18         std::string s;
19 };
20 class D
21 {
22 };
23 class E
24 {
25 };
26 class F : public virtual A, public B, public virtual D, public E
27 {
28     public:
29         F() : A(), D(), B(), E(), number1(0), number2(0U) {} // Compliant
30         F(F const& oth)
31             : B(), E(), A(), D(), number1(oth.number1), number2(oth.number2)
32         {
33         } // Non-compliant - incorrect
34         // order of initialization
```

```
35
36 private:
37     std::int32_t number1;
38     std::uint8_t number2;
39 };
```

## See also

- HIC++ v4.0 [8]: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 [6]

### **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.

## Example

```
1 // $Id: A8-5-2.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
2 #include <cstdlib>
3 #include <initializer_list>
4 void f1() noexcept
```



```
5 {
6     std::int32_t x1 =
7         7.9; // Non-compliant - x1 becomes 7 without compilation error
8     // std::int32_t y {7.9}; // Compliant - compilation error, narrowing
9     std::int8_t x2{50}; // Compliant
10    std::int8_t x3 = {50}; // Non-compliant - std::int8_t x3 {50} is equivalent
11                                // and more readable
12
13    std::int8_t x4 =
14        1.0; // Non-compliant - implicit conversion from double to std::int8_t
15    std::int8_t x5 = 300; // Non-compliant - narrowing occurs implicitly
16    std::int8_t x6(x5); // Non-compliant
17 }
18 class A
19 {
20     public:
21         A(std::int32_t first, std::int32_t second) : x{first}, y{second} {}
22
23     private:
24         std::int32_t x;
25         std::int32_t y;
26 };
27 struct B
28 {
29     std::int16_t x;
30     std::int16_t y;
31 };
32 class C
33 {
34     public:
35         C(std::int32_t first, std::int32_t second) : x{first}, y{second} {}
36         C(std::initializer_list<std::int32_t> list) : x{0}, y{0} {}
37
38     private:
39         std::int32_t x;
40         std::int32_t y;
41 };
42 void f2() noexcept
43 {
44     A a1{1, 5}; // Compliant - calls constructor of class A
45     A a2 = {1, 5}; // Non-compliant - calls a default constructor of class A
46                                // and not copy constructor or assignment operator.
47     A a3(1, 5); // Non-compliant
48     B b1{5, 0}; // Compliant - struct members initialization
49     C c1{2, 2}; // Compliant - C(std::initializer_list<std::int32_t>)
50                                // constructor is
51                                // called
52     C c2(2, 2); // Compliant by exception - this is the only way to call
53                                // C(std::int32_t, std::int32_t) constructor
54     C c3{{}; // Compliant - C(std::initializer_list<std::int32_t>) constructor
55                                // is
56                                // called with an empty initializer_list
```

```

56     C c4({2, 2}); // Compliant by exception -
57                 // C(std::initializer_list<std::int32_t>)
58                 // constructor is called
59 };
60 template <typename T, typename U>
61 void f1(T t, U u) noexcept(false)
62 {
63     std::int32_t x = 0;
64     T v1(x); // Non-compliant
65     T v2{x}; // Compliant - v2 is a variable
66     // auto y = T(u); // Non-compliant - is it construction or cast?
67     // Compilation error
68 };
69 void f3() noexcept
70 {
71     f1(0, "abcd"); // Compile-time error, cast from const char* to int
72 }

```

## See also

- C++ Core Guidelines [10]: ES.23 Prefer the {} initializer syntax.
- C++ Core Guidelines [10]: T.68: Use {} rather than () within templates to avoid ambiguities.
- Effective Modern C++ [12]: 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 271715 2017-03-23 10:13:51Z piotr.tanski $
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++ [12]: Item 2. Understand auto type deduction.
- Effective Modern C++ [12]: Item 7. Distinguish between () and {} when creating objects.

### Rule A8-5-4 (advisory, implementation, non-automated)

**A constructor taking parameter of type `std::initializer_list` shall only be defined in classes that internally store a collection of objects.**

## Rationale

If an object is initialized using {} braced-initialization, the compiler strongly prefers constructor taking parameter of type `std::initializer_list` to other constructors. Usage of constructors taking parameter of type `std::initializer_list` needs to be limited in order to avoid implicit calls to wrongly deduced constructor candidate of a class.

A class that internally stores a collection of objects is the case in which constructors taking parameter of type `std::initializer_list` are reasonable, allowing readable initialization of a class with a list of its elements.

## Example

```

1 // $Id: A8-5-4.cpp 271752 2017-03-23 12:07:07Z piotr.tanski $
2 #include <algorithm>
3 #include <cstdint>
4 #include <initializer_list>
5 #include <vector>
6 class A
7 {
8     public:
9         A() : x(0), y(0) {}
10        A(std::int32_t first, std::int32_t second) : x(first), y(second) {}
11        A(std::initializer_list<std::int32_t> list)
12            : x(0), y(0) // Non-compliant - class A does not store a collection
13                      // of objects
14        {
15            // ...
16        }
17
18     private:

```

```
19     std::int32_t x;
20     std::int32_t y;
21 };
22 class B
23 {
24     public:
25     B() : collection() {}
26     B(std::int32_t size, std::int32_t value) : collection(size, value) {}
27     B(std::initializer_list<std::int32_t> list)
28         : collection(
29             list) // Compliant - class B stores a collection of objects
30     {
31     }
32
33     private:
34     std::vector<std::int32_t> collection;
35 };
36 class C
37 {
38     public:
39     C() : array{0} {}
40     C(std::initializer_list<std::int32_t> list)
41         : array{0} // Compliant - class C stores a collection of objects
42     {
43         std::copy(list.begin(), list.end(), array);
44     }
45
46     private:
47     static constexpr std::int32_t size = 100;
48     std::int32_t array[size];
49 };
50 class D : public C
51 {
52     public:
53     D() : C() {}
54     D(std::initializer_list<std::int32_t> list)
55         : C{list} // Compliant - class D inherits a collection of objects
56                 // from class C
57     {
58     }
59 };
60 class E
61 {
62     public:
63     E() : container() {}
64     E(std::initializer_list<std::int32_t> list)
65         : container{list} // Compliant - class E stores class C which
66                           // stores a collection of objects
67     {
68     }
69 }
```

```
70     private:
71         C container;
72     };
73     void f1() noexcept
74     {
75         A a1{};           // Calls A::A()
76         A a2{{{}}};      // Calls A::A(std::initializer_list<std::int32_t>)
77         A a3{0, 1};      // Calls A::A(std::initializer_list<std::int32_t>)
78         A a4(0, 1);      // Calls A::A(std::int32_t, std::int32_t)
79     }
80     void f2() noexcept
81     {
82         B b1{};          // Calls B::B()
83         B b2{{{}}};      // Calls B::B(std::initializer_list<std::int32_t>)
84         B b3{1, 2};      // Calls B::B(std::initializer_list<std::int32_t>)
85         B b4(10, 0);     // Calls B::B(std::int32_t, std::int32_t)
86     }
87     void f3() noexcept
88     {
89         C c1{};          // Calls C::C()
90         C c2{{{}}};      // Calls C::C(std::initializer_list<std::int32_t>)
91         C c3{1, 2, 3};   // Calls C::C(std::initializer_list<std::int32_t>)
92     }
93     void f4() noexcept
94     {
95         D d1{};          // Calls D::D()
96         D d2{{{}}};      // Calls D::D(std::initializer_list<std::int32_t>)
97         D d3{1, 2, 3};   // Calls D::D(std::initializer_list<std::int32_t>)
98     }
99     void f5() noexcept
100    {
101        E e1{};           // Calls E::E()
102        E e2{{{}}};      // Calls E::E(std::initializer_list<std::int32_t>)
103        E e3{1, 2, 3};   // Calls E::E(std::initializer_list<std::int32_t>)
104    }
```

## See also

- Effective Modern C++ [12]: Item 7. Distinguish between () and {} when creating objects.

## 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 [6]

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 [6]]

**Rule A9-3-1 (required, implementation, 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 271927 2017-03-24 12:01:35Z piotr.tanski $
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 [6]: Rule 9-3-2 Member functions shall not return non-const handles to class-data.

### **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 [6]

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.

## 6.9.5 Unions

### **Rule M9-5-1 (required, implementation, automated)**

**Unions shall not be used.**

See MISRA C++ 2008 [6]

## 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 [6]

### **Rule A9-6-1 (required, implementation, automated)**

**Bit-fields shall be either unsigned integral, or enumeration (with underlying type of unsigned integral type).**

## Rationale

Explicitly declaring a bit-field unsigned prevents unexpected sign extension, overflows and implementation-defined behavior.



Note that if a bit-field has enumeration type, then the enumeration base needs to be declared of an explicitly unsigned type.

### Example

```
1 // $Id: A9-6-1.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
2 #include <cstdint>
3 enum class E1 : std::uint8_t
4 {
5     E11,
6     E12,
7     E13
8 };
9 enum class E2 : std::int16_t
10 {
11     E21,
12     E22,
13     E23
14 };
15 enum class E3
16 {
17     E31,
18     E32,
19     E33
20 };
21 enum E4
22 {
23     E41,
24     E42,
25     E43
26 };
27 class C
28 {
29     public:
30         std::int32_t a : 2; // Non-compliant - signed integral type
31         std::uint8_t b : 2U; // Compliant
32         bool c : 1; // Non-compliant - it is implementation-defined whether bool is
33                     // signed or unsigned
34         char d : 2; // Non-compliant
35         wchar_t e : 2; // Non-compliant
36         E1 f1 : 2; // Compliant
37         E2 f2 : 2; // Non-compliant - E2 enum class underlying type is signed
38                     // int
39         E3 f3 : 2; // Non-compliant - E3 enum class does not explicitly define
40                     // underlying type
41         E4 f4 : 2; // Non-compliant - E4 enum does not explicitly define underlying
42                     // type
43 };
44 void fn() noexcept
45 {
46     C c;
47     c.f1 = E1::E11;
```

48 }

## See also

- JSF December 2005 [7]: AV Rule 154 Bit-fields shall have explicitly unsigned integral or enumeration types only.
- HIC++ v4.0 [8]: 9.2.1 Declare bit-fields with an explicitly unsigned integral or enumeration type.

## 6.10 Derived Classes

### 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 271715 2017-03-23 10:13:51Z piotr.tanski $
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 [7]: 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 [8]: 10.3.1 Ensure that a derived class has at most one base class which is not an interface class.
- C++ Core Guidelines [10]: 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 [6]

**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 [6]

**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 [6]

**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 [6]

**Rule A10-2-1 (required, implementation, automated)****Non-virtual 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.

**Example**

```
1 // $Id: A10-2-1.cpp 271927 2017-03-24 12:01:35Z piotr.tanski $
2 class A
3 {
4     public:
5         virtual ~A() = default;
6         void f() noexcept {}
7         virtual void g() noexcept {}
8 };
9
10 class B : public A
11 {
12     public:
13         void
14         f() noexcept // Non-compliant - f() function from A class hidden by B class
15         {
16         }
17         void g() noexcept override // Compliant - g() function from A class
18                                     // overridden by B class
19         {
20         }
21 };
```

```

22 void fn1(A& object) noexcept
23 {
24     object.f(); // Calls f() function from A
25     object.g(); // Calls g() function from B
26 }
27 void fn2() noexcept
28 {
29     B b;
30     fn1(b);
31 }
  
```

### See also

- JSF December 2005 [7]: AV Rule 94 An inherited nonvirtual function shall not be redefined in a derived class.
- C++ Core Guidelines [10]: 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

```

1 // $Id: A10-3-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 class A
3 {
4     public:
5         virtual ~A() {} // Compliant
6         virtual void f() noexcept = 0; // Compliant
7         virtual void g() noexcept final = 0; // Non-compliant - virtual final pure
8                                             // function is redundant
9         virtual void
10        h() noexcept final // Non-compliant - function is virtual and final
11        {
12        }
13        virtual void k() noexcept // Compliant
14        {
  
```

```
15     }
16     virtual void j() noexcept {}
17     virtual void m() noexcept // Compliant
18     {
19     }
20     virtual void z() noexcept // Compliant
21     {
22     }
23     virtual A& operator+=(A const& rhs) noexcept // Compliant
24     {
25         // ...
26         return *this;
27     }
28 };
29 class B : public A
30 {
31     public:
32     ~B() override {} // Compliant
33     virtual void f() noexcept override // Non-compliant - function is specified
34                                         // 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 [10]: 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 271687 2017-03-23 08:57:35Z piotr.tanski $
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 [8]: 10.2.1 Use the override special identifier when overriding a virtual function
- C++ Core Guidelines [10]: 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

```
1 // $Id: A10-3-3.cpp 271927 2017-03-24 12:01:35Z piotr.tanski $
2 class A
3 {
4     public:
5         virtual ~A() = default;
6         virtual void f() noexcept = 0;
7         virtual void g() noexcept {}
8 };
9 class B final : public A
10 {
11     public:
12         void f() noexcept final // Compliant
13         {
14         }
15         void g() noexcept override // Non-compliant
16         {
17         }
18         virtual void h() noexcept = 0; // Non-compliant
19         virtual void z() noexcept // Non-compliant
20         {
21         }
22 };
```



## See also

- HIC++ v4.0 [8]: 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 overrides. 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

```
1 // $Id: A10-3-4.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 class A
3 {
4     public:
5         virtual A& operator=(A const& oth) = 0;    // Non-compliant
6         virtual A& operator+=(A const& rhs) = 0;  // Non-compliant
7 };
8 class B : public A
9 {
10     public:
11         B& operator=(A const& oth) override // It needs to take an argument of type
12                                             // A& in order to override
13         {
14             return *this;
15         }
16         B& operator+=(A const& oth) override // It needs to take an argument of
17                                             // type A& in order to override
18         {
19             return *this;
20         }
21         B& operator--=(B const& oth) // Compliant
22         {
23             return *this;
24         }
25 };
26 class C : public A
27 {
28     public:
29         C& operator=(A const& oth) override // It needs to take an argument of type
30                                             // A& in order to override
31         {
32             return *this;
33         }
```

```
34     C& operator+=(A const& oth) override // It needs to take an argument of
35                                         // type A& in order to override
36     {
37         return *this;
38     }
39     C& operator-=(C const& oth) // Compliant
40     {
41         return *this;
42     }
43 };
44 // class D : public A
45 //{
46 // public:
47 //     D& operator=(D const& oth) override // Compile time error - this method
48 //     does not override because of different
49 //     signature
50 //     {
51 //         return *this;
52 //     }
53 //     D& operator+=(D const& oth) override // Compile time error - this method
54 //     does not override because of different
55 //     signature
56 //     {
57 //         return *this;
58 //     }
59 //};
60 void fn() noexcept
61 {
62     B b;
63     C c;
64     b = c; // Calls B::operator= and accepts an argument of type C
65     b += c; // Calls B::operator+= and accepts an argument of type C
66     c = b; // Calls C::operator= and accepts an argument of type B
67     c += b; // Calls C::operator+= and accepts an argument of type B
68     // b -= c; // Compilation error, because of types mismatch. Expected
69     // behavior
70     // c -= b; // Compilation error, because of types mismatch. Expected
71     // behavior
72
73     B b2;
74     C c2;
75     b -= b2;
76     c -= c2;
77 }
```

## 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 [6]

See: [A10-3-2](#) for pure virtual function overrides declaration.

## 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 [6]

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

```
1 // $Id: A11-0-1.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
2 #include <stdint>
3 #include <limits>
4 class A // Compliant - A provides user-defined constructors, invariant and
5         // interface
6 {
7     std::int32_t x; // Data member is private by default
8
9     public:
10    static constexpr std::int32_t maxValue =
11        std::numeric_limits<std::int32_t>::max();
12    A() : x(maxValue) {}
13    explicit A(std::int32_t number) : x(number) {}
14    A(A const&) = default;
15    A(A&&) = default;
16    A& operator=(A const&) = default;
17    A& operator=(A&&) = default;
```

```
18
19     std::int32_t getX() const noexcept { return x; }
20     void setX(std::int32_t number) noexcept { x = number; }
21 };
22 struct B // Non-compliant - non-POD type defined as struct
23 {
24     public:
25         static constexpr std::int32_t maxValue =
26             std::numeric_limits<std::int32_t>::max();
27         B() : x(maxValue) {}
28         explicit B(std::int32_t number) : x(number) {}
29         B(B const&) = default;
30         B(B&&) = default;
31         B& operator=(B const&) = default;
32         B& operator=(B&&) = default;
33
34         std::int32_t getX() const noexcept { return x; }
35         void setX(std::int32_t number) noexcept { x = number; }
36
37     private:
38         std::int32_t x; // Need to provide private access specifier for x member
39 };
40 struct C // Compliant - POD type defined as struct
41 {
42     std::int32_t x;
43     std::int32_t y;
44 };
45 class D // Compliant - POD type defined as class, but not compliant with
46         // M11-0-1
47 {
48     public:
49         std::int32_t x;
50         std::int32_t y;
51 };
```

## See also

- C++ Core Guidelines [10]: C.2: Use class if the class has an invariant; use struct if the data members can vary independently.
- [stackoverflow.com \[16\]](https://stackoverflow.com/questions/11111111/when-should-you-use-a-class-vs-a-struct-in-c): 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

```
1 // $Id: A11-0-2.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
2 #include <stdint>
3 struct A // Compliant
4 {
5     std::int32_t x;
6     double y;
7 };
8 struct B // Compliant
9 {
10    std::uint8_t x;
11    A a;
12 };
13 struct C // Compliant
14 {
15    float x = 0.0f;
16    std::int32_t y = 0;
17    std::uint8_t z = 0U;
18 };
19 struct D // Non-compliant
20 {
21    public:
22        std::int32_t x;
23
24    protected:
25        std::int32_t y;
26
27    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 \[16\]](https://stackoverflow.com/questions/1611244/when-should-you-use-a-class-vs-a-struct-in-c): 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.

#### Example

```
1 // $Id: A11-3-1.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
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 };
13 B const operator+(B const& lhs, B const& rhs) // Compliant
14 {
15     // Implementation
16 }
```

## See also

- JSF December 2005 [7]: 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 [8]: 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 defines any special member function “=default”, “=delete” or with a

**function definition, then all of the special member functions shall be defined.**

## Rationale

Semantics of all of the special member functions are closely related to each other. If any special member function needs to be non-default, then all others need to be defined too.

Note that this rule is also known as “the rule of five” or “the rule of six”.

Also, note that the rule allows to follow “the rule of zero” for types that do not need to define any special member function.

## Example

```
1 // $Id: A12-0-1.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
2 #include <stdint>
3 class A // Compliant - the class A follow the "Rule of six" rule
4 {
5     public:
6         A(); // Non-default constructor
7         ~A() = default;
8         A(A const&) = default;
9         A& operator=(A const&) = default;
10        A(A&&) = delete;
11        A& operator=(A&&) = delete;
12 };
13 class B // Non-compliant - some special functions are defined but not all of
14         // them
15 {
16     public:
17         B();
18         ~B();
19
20     private:
21         std::int32_t* pointer;
22 };
23 struct C // Compliant - no special functions are defined
24 {
25     std::int32_t number;
26 };
```

## See also

- C++ Core Guidelines [10]: C.21: If you define or =delete any default operation, define or =delete them all.

### 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 [6]: 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 [8]:12.4.2 Ensure that a constructor initializes explicitly all base classes and non-static data members.



**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 [6]

Note: This rule prohibits both direct and indirect usage of object's dynamic type from its constructor or destructor.

**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

```
1 // $Id: A12-1-2.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
2 #include <stdint>
3 #include <utility>
4 class A
5 {
6     public:
7         A() : i1{0}, i2{0} // Compliant - i1 and i2 are initialized by the
8                        // constructor only. Not compliant with A12-1-3
9         {
10        }
11        // Implementation
12
13     private:
14         std::int32_t i1;
15         std::int32_t i2;
16 };
17 class B
18 {
19     public:
20         // Implementation
21
22     private:
23         std::int32_t i1{0};
```

```
24     std::int32_t i2{
25         0}; // Compliant - both i1 and i2 are initialized by NSDMI only
26 };
27 class C
28 {
29     public:
30     C() : i2{0} // Non-compliant - i1 is initialized by NSDMI, i2 is in
31           // member in member initializer list
32     {
33     }
34     C(C const& oth) : i1{oth.i1}, i2{oth.i2} // Compliant by exception
35     {
36     }
37     C(C&& oth)
38         : i1{std::move(oth.i1)},
39           i2{std::move(oth.i2)} // Compliant by exception
40     {
41     }
42     // Implementation
43
44     private:
45     std::int32_t i1{0};
46     std::int32_t i2;
47 };
```

## See also

- HIC++ v4.0 [8]: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 271696 2017-03-23 09:23:09Z piotr.tanski $
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 };
27 class C
28 {
29     public:
30         C() : x(0), y(0.0F), str() // Compliant
31         {
32         }
33         C(std::int32_t i, float f, std::string s) : x(i), y(f), str(s) // Compliant
34         {
35         }
36         // ...
37
38     private:
39         std::int32_t x =
40             0;                // Non-compliant - there's a constructor that initializes C
41                               // class with user input
42         float y = 0.0F;      // Non-compliant - there's a constructor that initializes C
43                               // class with user input
44         std::string str = ""; // Non-compliant - there's a constructor that
45                               // initializes C class with user input
46 };

```

## See also

- C++ Core Guidelines [10]: 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

```
1 // $Id: A12-1-4.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
2 #include <stdint>
3 class A
4 {
5     public:
6         explicit A(std::int32_t number) : x(number) {} // Compliant
7         A(A const&) = default;
8         A(A&&) = default;
9         A& operator=(A const&) = default;
10        A& operator=(A&&) = default;
11
12    private:
13        std::int32_t x;
14 };
15 class B
16 {
17     public:
18         B(std::int32_t number) : x(number) {} // Non-compliant
19         B(B const&) = default;
20         B(B&&) = default;
21         B& operator=(B const&) = default;
22         B& operator=(B&&) = default;
23
24    private:
25        std::int32_t x;
26 };
27 void f1(A a) noexcept
28 {
29 }
30 void f2(B b) noexcept
31 {
32 }
33 void f3() noexcept
34 {
35     f1(A(10));
36     // f1(10); // Compilation error - because of explicit constructor it is not
37     // possible to implicitly convert integer
38     // to type of class A
39     f2(B(20));
40     f2(20); // No compilation error - implicit conversion occurs
41 }
```

### See also

- MISRA C++ 2008 [6]: Rule 12-1-3 (Required) All constructors that are callable with a single argument of fundamental type shall be declared explicit.

## 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 271687 2017-03-23 08:57:35Z piotr.tanski $
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 [7]: AV Rule 78 All base classes with a virtual function shall define a virtual destructor.
- HIC++ v4.0 [8]: 12.2.1 Declare virtual, private or protected the destructor of a type used as a base class.
- C++ Core Guidelines [10]: C.35: A base class destructor should be either public and virtual, or protected and nonvirtual.
- C++ Core Guidelines [10]: 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

```

1 // $Id: A12-4-2.cpp 270924 2017-03-17 14:50:50Z piotr.tanski $
2 class A // Non-compliant - class A should not be used as a base class because
3         // its destructor is not virtual, but it is
4         // not declared final
5 {
6     public:
7         A() = default;
8         A(A const&) = default;
9         A(A&&) = default;
10        A& operator=(A const&) = default;
11        A& operator=(A&&) = default;
12        ~A() = default; // Public non-virtual destructor
13 };
14 class B final // Compliant - class B can not be used as a base class, because
15              // it is declared final, and it should not be derived
16              // because its destructor is not virtual
17 {
18     public:
  
```

```

19     B() = default;
20     B(B const&) = default;
21     B(B&&) = default;
22     B& operator=(B const&) = default;
23     B& operator=(B&&) = default;
24     ~B() = default; // Public non-virtual destructor
25 };
26 class C // Compliant - class C is not final, and its destructor is virtual. It
27         // can be used as a base class
28 {
29     public:
30     C() = default;
31     C(C const&) = default;
32     C(C&&) = default;
33     C& operator=(C const&) = default;
34     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

- none

## 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 <stdint>
3 #include <string>
4 class A
5 {
6     public:
7         A(std::int32_t n, std::string s) : number{n}, str{s} // Compliant
8         {
9         }
10        // Implementation
11
12     private:
13         std::int32_t number;
14         std::string str;
15 };
16 class B
17 {
18     public:
19         B(std::int32_t n, std::string s) // Non-compliant - no member initializers
20         {
21             number = n;
22             str = s;
23         }
24        // Implementation
25
26     private:
27         std::int32_t number;
28         std::string str;
29 };
30 class C
31 {
32     public:
33         C(std::int32_t n, std::string s) : number{n}, str{s} // Compliant
34         {
35             n += 1; // This does not violate the rule
36             str.erase(str.begin(),
37                 str.begin() + 1); // This does not violate the rule
38         }
39        // Implementation
40
41     private:
42         std::int32_t number;
```



```
43     std::string str;
44 };
```

### See also

- C++ Core Guidelines [10]: 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 <cstdlib>
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) {} //
9                                     Compliant // -
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
31     private:
32         std::int32_t x;
33         std::int32_t y;
34 };
35 class B
36 {
37     public:
38     B() {} // Non-compliant - x and y are not initialized
39           // should be replaced with: B() : x{0}, y{0} {}
40     B(std::int32_t first, std::int32_t second) : x(first), y(second) {} //
41     Compliant
42     B(const B&) =
43         default; // Compliant - equivalent to the copy constructor of class A
44     B(B&&) =
45         default; // Compliant - equivalent to the move constructor of class A
46     ~B() = default; // Compliant - equivalent to the destructor of class A
47
48     private:
49         std::int32_t x;
50         std::int32_t y;
51 };
52 class C
53 {
54     public:
55     C() = default; // Compliant
56     C(const C&) = default; // Compliant
57     C(C&&) = default; // Compliant
58 };
59 class D
60 {
61     public:
62     D() : ptr(nullptr) {} // Compliant - this is not equivalent to what the
63                           // implicitly defined default constructor would do
64     D(C* p) : ptr(p) {} // Compliant
65     D(const D&) = default; // Shallow copy will be performed, user-defined copy
66                           // constructor is needed to perform deep copy on ptr variable
67     D(D&&) = default; // ptr variable will be moved, so ptr will still point to
68                       // the same object
69     ~D() = default; // ptr will not be deleted, the user-defined destructor is
70                     // needed to delete allocated memory

```

```
70
71     private:
72         C* ptr;
73     };
74     class E // Compliant - special member functions definitions are not needed as
75             // class E uses only implicit definitions
76     {
77     };
```

## See also

- HIC++ v4.0 [8]: 12.5.2 Define special members =default if the behavior is equivalent.
- C++ Core Guidelines [10]: 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 only 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 eventually set copied-from or 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.

### Example

```
1 // $Id: A12-8-1.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
2 #include <stdint>
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 };
```

## See also

- MISRA C++ 2008 [6]: 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 [8]: 12.5.3 Ensure that a user defined move/copy constructor only moves/copies base and member objects.

**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

```
1 // $Id: A12-8-2.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
2 #include <cstdlib>
3 #include <utility>
```

```
4 class A
5 {
6     public:
7         A(const A& oth)
8         {
9             // ...
10        }
11        A(A&& oth) noexcept
12        {
13            // ...
14        }
15        A& operator=(const A& oth) & // Compliant
16        {
17            A tmp(oth);
18            swap(*this, tmp);
19            return *this;
20        }
21        A& operator=(A&& oth) & noexcept // Compliant
22        {
23            A tmp(std::move(oth));
24            swap(*this, tmp);
25            return *this;
26        }
27        static void swap(A& lhs, A& rhs) noexcept
28        {
29            std::swap(lhs.ptr1, rhs.ptr1);
30            std::swap(lhs.ptr2, rhs.ptr2);
31        }
32
33    private:
34        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 [8]: 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 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <cstdlib>
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()
15 {
16     std::unique_ptr<std::int32_t> ptr1 = std::make_unique<std::int32_t>(0);
17     std::unique_ptr<std::int32_t> ptr2{std::move(ptr1)};
18     std::cout << ptr1.get() << std::endl; // Compliant by exception - move
19                                           // construction of std::unique_ptr
20                                           // leaves moved-from object in a
21                                           // well-specified state
22 }

```

### See also

- SEI CERT C++ [9]: 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

```
1 // $Id: A12-8-4.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
2 #include <stdint>
3 #include <string>
4 class A
5 {
6     public:
7         // ...
8         A(A&& oth)
9             : x(std::move(oth.x)), // Compliant
10              s(std::move(oth.s)) // Compliant
11     {
12     }
13
14     private:
15         std::int32_t x;
16         std::string s;
17 };
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++ [9]: 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

```
1 // $Id: A12-8-5.cpp 271773 2017-03-23 13:16:53Z piotr.tanski $  
2 #include <cstdlib>  
3 #include <stdexcept>  
4 struct A  
5 {  
6     std::int32_t number;  
7     std::int32_t* ptr;  
8     // Implementation  
9 };  
10 class B  
11 {  
12     public:  
13     // ...  
14     B& operator=(B const& oth) // Non-compliant  
15     {  
16         i = oth.i;  
17         delete aPtr;  
18  
19         try  
20         {  
21             aPtr = new A(*oth.aPtr); // If this is the self-copy case, then  
22                                     // the oth.a_ptr is already deleted  
23         }  
24         catch (std::bad_alloc&)  
25         {
```

```
26         aPtr = nullptr;
27     }
28
29     return *this;
30 }
31
32 private:
33     std::int16_t i = 0;
34     A* aPtr = nullptr;
35 };
36 class C
37 {
38 public:
39     C& operator=(C const& oth) // Compliant
40     {
41         if (this != &oth)
42         {
43             A* tmpPtr = new A(*oth.aPtr);
44
45             i = oth.i;
46             delete aPtr;
47             aPtr = tmpPtr;
48         }
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++ [9]: OOP54-CPP Gracefully handle self-assignment.
- C++ Core Guidelines [10]: 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 271927 2017-03-24 12:01:35Z piotr.tanski $
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 [6]: Rule 12-8-2 The copy assignment operator shall be declared protected or private in an abstract class.
- HIC++ v4.0 [8]: 12.5.8 Make the copy assignment operator of an abstract class protected or define it =delete.
- C++ Core Guidelines [10]: C.67: A base class should suppress copying, and provide a virtual clone instead if "copying" is desired.

**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 271687 2017-03-23 08:57:35Z piotr.tanski $
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 [8]: 12.5.7 Declare assignment operators with the ref-qualifier &.
- cppreference.com [15]: Assignment operators.

## 6.13 Overloading

### 6.13.1 Overloadable declarations

**Rule A13-1-1 (required, implementation, automated)**  
**User-defined literals shall not be used.**

#### Rationale

User-defined literals permits only following types for parameter lists:

- `const char*`
- `unsigned long long int`
- `long double`
- `char`
- `wchar_t`
- `char16_t`
- `char32_t`
- `const char*`, `std::size_t`
- `const wchar_t*`, `std::size_t`
- `const char16_t*`, `std::size_t`
- `const char32_t*`, `std::size_t`

A programmer has limited control on the types of parameters passed to user-defined conversion operators. Also, it is implementation-defined whether fixed-size types from `<cstdint>` header are compatible with the types allowed by user-defined literals.

Note that user-defined literals are not widespread in C++ Standard Library. They are used to convert literals to objects of type `std::chrono::duration` (i.e. `h`, `min`, `s`, `ms`, `us`, `ns`), `std::complex` and `basic_string`.

#### Example

```
1 // $Id: A13-1-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <cstdint>
3 constexpr unsigned long long int operator"" _ft(
4     unsigned long long int feet) // Non-compliant
5 {
6     // Implementation
7     return feet;
8 }
9 constexpr std::uint64_t operator"" _m(std::uint64_t meters) // Non-compliant
10 {
11     // Implementation
12     return meters;
13 }
14 void fn() noexcept
15 {
16     unsigned long long int feets = 200_ft;
17     // std::uint64_t meters = 300_m; // Compilation error - uint64_t has
18     // invalid
19     // type, not compatible with user-defined literals. On 64-bit machines,
20     // uint64_t is defined as unsigned long, and not unsigned long long
21 }
```

## See also

- none

### 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

```
1 // $Id: A13-1-2.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 constexpr long double operator"" _m(long double meters) // Compliant
3 {
4     // Implementation
5     return meters;
6 }
7 constexpr long double operator"" _kg(long double kilograms) // Compliant
8 {
9     // Implementation
10    return kilograms;
11 }
12 constexpr long double operator"" m(long double meters) // Non-compliant
13 {
14     // Implementation
15     return meters;
16 }
```



```
16 }
17 constexpr long double operator"" kilograms(
18     long double kilograms) // Non-compliant
19 {
20     // Implementation
21     return kilograms;
22 }
23 void fn()
24 {
25     long double weight = 20.0_kg;
26     long double distance = 204.8_m;
27 }
```

## 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

```
1 // $Id: A13-1-3.cpp 271927 2017-03-24 12:01:35Z piotr.tanski $
2 #include <cstdint>
3 #include <iostream>
4 struct Cube
5 {
6     unsigned long long int volume;
7     constexpr explicit Cube(unsigned long long int v) : volume(v) {}
8 };
9 constexpr Cube operator"" _m3(unsigned long long int volume)
10 {
11     return Cube(volume); // Compliant
12 }
13 struct Temperature
14 {
15     unsigned long long int kelvins;
16     constexpr explicit Temperature(unsigned long long int k) : kelvins(k) {}
17 };
18 constexpr Temperature operator"" _K(unsigned long long int kelvins)
19 {
20     return Temperature(kelvins); // Compliant
21 }
22 static void sumDistances(std::int32_t distance)
23 {
```

```

24     static std::int32_t overallDistance = 0;
25     overallDistance += distance;
26 }
27 struct Distance
28 {
29     long double kilometers;
30     explicit Distance(long double kms) : kilometers(kms) {}
31 };
32 Distance operator"" _m(long double meters)
33 {
34     sumDistances(meters); // Non-compliant - function has a side-effect
35     return Distance(meters / 1000);
36 }
37 void operator"" _print(const char* str)
38 {
39     std::cout << str << '\n'; // Non-compliant - user-defined literal operator
40                               // does not perform conversion and has a
41                               // side-effect
42 }
  
```

## 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

```

1 // $Id: A13-2-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 class A
3 {
4     public:
5         // ...
6         A& operator=(const A&) & // Compliant
7         {
8             // ...
9             return *this;
10        }
11 };
12
13 class B
14 {
  
```

```
15 public:
16     // ...
17     const B& operator=(const B&) & // Non-compliant - violating consistency
18                                     // with standard types
19     {
20         // ...
21         return *this;
22     }
23 };
24
25 class C
26 {
27     public:
28         // ...
29         C operator=(const C&) & // Non-compliant
30         {
31             // ...
32             return *this;
33         }
34 };
35
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 [8]: 13.2.2 Ensure that the return type of an overloaded binary operator matches the built-in counterparts.
- C++ Core Guidelines [10]: 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 [8]: 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
29 {
30     return lhs;
31 }
```

## See also

- HIC++ v4.0 [8]: 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.

#### Example

```
1 // $Id: A13-3-1.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
2 #include <cstdlib>
3 template <typename T>
4 void f1(T&& t) noexcept(false)
```

```
5 {
6 }
7 void f1(
8     std::int32_t&& t) noexcept // Non-compliant - overloading a function with
9                               // forwarding reference
10 {
11 }
12 template <typename T>
13 void f2(T&& t) noexcept(false)
14 {
15 }
16 void f2(std::int32_t&) = delete; // Compliant by exception
17
18 int main(int, char**)
19 {
20     std::int32_t x = 0;
21     f1(x); // Calls f1(T&&) with T = int&
22     f1(+x); // Calls f1(std::int32_t&&)
23     f1(0); // Calls f1(std::int32_t&&)
24     f1(0U); // Calls f1(T&&) with T = unsigned int
25     f2(0); // Calls f2(T&&) with T = int
26     // f2(x); // Compilation error, the overloading function is deleted
27 }
```

## See also

- HIC++ v4.0 [8]: 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++ [12]: 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 271715 2017-03-23 10:13:51Z piotr.tanski $
2 #include <cstdint>
3 class Container1
4 {
5     public:
6         std::int32_t& operator[](
7             std::int32_t index) // Compliant - non-const version
8         {
9             return container[index];
10        }
11        std::int32_t operator[](
12            std::int32_t index) const // Compliant - const version
13        {
14            return container[index];
15        }
16
17    private:
18        static constexpr std::int32_t maxSize = 10;
19        std::int32_t container[maxSize];
20 };
21 void fn() noexcept
22 {
23     Container1 c1;
24     std::int32_t e = c1[0]; // Non-const version called
25     c1[0] = 20;           // Non-const version called
26     Container1 const c2{};
27     e = c2[0]; // Const version called
28     // c2[0] = 20; // Compilation error
29 }
30 class Container2 // Non-compliant - only non-const version of operator[]
31                 // implemented
32 {
33     public:
34         std::int32_t& operator[](std::int32_t index) { return container[index]; }
35
36     private:
37         static constexpr std::int32_t maxSize = 10;
38         std::int32_t container[maxSize];
39 };
```

## See also

- HIC++ v4.0 [8]: 13.2.4 When overloading the subscript operator (operator[]) implement both const and non-const versions.

### 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 every 4 digits, e.g. 0b1001'1101'0010

## Example

```

1 // $Id: A13-6-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
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'FFFF; // Non-compliant
12    std::uint8_t binary1 = 0b1001'0011; // Compliant
13    std::uint8_t binary2 = 0b10'00'10'01; // Non-compliant
14 }
```

## See also

- none
- ISO 26262-6 [4]: 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 271696 2017-03-23 09:23:09Z piotr.tanski $
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>
23 void f1(T const& obj) noexcept(false)
24 {
25     static_assert(
26         std::is_copy_constructible<T>(),
27         "Given template type is not copy constructible."); // Compliant
28 }
29 template <typename T>
30 class C
31 {
32     // Compliant
33     static_assert(std::is_trivially_copy_constructible<T>(),
34                 "Given template type is not trivially copy constructible.");
35
36     // Compliant
37     static_assert(std::is_trivially_move_constructible<T>(),
38                 "Given template type is not trivially move constructible.");
39
40     // Compliant
```

```
41     static_assert(std::is_trivially_copy_assignable<T>(),
42                 "Given template type is not trivially copy assignable.");
43
44     // Compliant
45     static_assert(std::is_trivially_move_assignable<T>(),
46                 "Given template type is not trivially move assignable.");
47
48     public:
49         C() = default;
50         C(C const&) = default;
51         C& operator=(C const&) = default;
52         C(C&&) = default;
53         C& operator=(C&&) = default;
54
55     private:
56         T c;
57 };
58 template <typename T>
59 class D
60 {
61     public:
62         D() = default;
63         D(D const&) = default;           // Non-compliant - T may not be copyable
64         D& operator=(D const&) = default; // Non-compliant - T may not be copyable
65         D(D&&) = default;               // Non-compliant - T may not be movable
66         D& operator=(D&&) = default;     // Non-compliant - T may not be movable
67
68     private:
69         T d;
70 };
71 void f2() noexcept
72 {
73     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

- C++ Core Guidelines [10]: T.150: Check that a class matches a concept using `static_assert`.

### 6.14.5 Template declarations

**Rule M14-5-2 (required, implementation, automated)**  
**A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter.**

See MISRA C++ 2008 [6]

Note: Move constructor will not be generated implicitly if a user defines a copy constructor.

**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 [6]

### 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 [6]

### 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 271696 2017-03-23 09:23:09Z piotr.tanski $
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 [6]: 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 M14-7-3 (required, implementation, automated)**  
**All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template.**

See MISRA C++ 2008 [6]

Note: If no partial or explicit specializations for a template are needed, then they do not have to be declared.

### 6.14.8 Function template specializations

**Rule M14-8-1 (required, implementation, automated)**  
**Overloaded function templates shall not be explicitly specialized.**

See MISRA C++ 2008 [6]

**Rule A14-8-1 (advisory, implementation, automated)**  
**The set of function overloads should not contain function templates, functions specializations and non-template overloading functions.**

### Rationale

If a function template or function specialization and a non-template overloading function are equivalent after overload resolution, the non-template overloading function will be chosen by the compiler. This may be inconsistent with developer expectations.

### Exception

This rule does not apply to copy constructors or copy assignment operators.

### Example

```
1 // $Id: A14-8-1.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
2 #include <stdint>
3 template <typename T>
4 void f1(T t)
5 {
6     // Implementation
7 }
8 void f1(std::int16_t n)
```

```
9 {
10     // Implementation
11 }
12 void fn1() noexcept
13 {
14     std::int16_t x = 0;
15     f1(x);           // f1(std::int16_t) is called
16     f1(x + 10);     // f1(T) is called with T = int
17     f1<>(x);        // explicit call to f1(T) with T = short int
18     f1<>(x + 10);   // explicit call to f1(T) with T = int
19 }
20 template <typename T>
21 void f2(T t)
22 {
23     // Implementation
24 }
25 template <>
26 void f2<std::int16_t>(std::int16_t n)
27 {
28     // Implementation
29 }
30 void f2(std::int16_t n)
31 {
32     // Implementation
33 }
34 void fn2() noexcept
35 {
36     std::int16_t x = 0;
37     f2(x);           // f2(std::int16_t) is called
38     f2(x + 10);     // f2(T) is called with T = int
39     f2<>(x);        // explicit call to f2<std::int16_t>(std::int16_t)
40     f2<>(x + 10);   // explicit call to f2(T) with T = int
41     f2<std::int16_t>(x +
42                     10); // explicit call to f2<std::int16_t>(std::int16_t)
43 }
44 void f3(std::int16_t n)
45 {
46     // Implementation
47 }
48 void f3(std::int32_t n)
49 {
50     // Implementation
51 }
52 void fn3() noexcept
53 {
54     std::int16_t x = 0;
55     f3(x);           // f3(std::int16_t) is called
56     f3(x + 10);     // f3(std::int32_t) is called
57 }
58 template <typename T>
59 void f4(T t)
```

```
60 {
61     // Implementation
62 }
63 template <>
64 void f4<std::int16_t>(std::int16_t n)
65 {
66     // Implementation
67 }
68 void fn4() noexcept
69 {
70     std::int16_t x = 0;
71     f4(x); // f4(T) with T = short int is called
72     f4(x + 10); // f4(T) with T = int is called
73     f4<std::int16_t>(x + 100); // explicit call to f4(T) with T = short int
74 }
```

### See also

- MISRA C++ 2008 [6]: Rule 14-8-2 (Advisory) The viable function set for a function call should either contain no function specializations, or only contain function specializations.

## 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 <a href="#">A1-2-1</a> 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 <a href="#">A15-0-1</a> 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 <a href="#">A15-0-1</a> 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. Moreover, AUTOSAR C++ Coding Guidelines does not force developers to strictly follow single-point of exit approach as it does not necessarily make the code more readable or easier to maintain.
Code readability	If exceptions are used correctly, in particularly by using checked and unchecked exception types, see Rules: <a href="#">A15-0-4</a> and <a href="#">A15-0-5</a> , 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 <a href="#">A15-0-2</a> 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 <a href="#">A15-0-7</a> 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:

- **Unchecked Exceptions:** Used to represent errors that a program can not recover from, so they are not supposed to be declared by functions nor caught by caller functions. These are all exceptions (either built-in or user-defined) that are instances or subclasses of one of the following standard exception type:
  - `logic_error`
  - `bad_typeid`
  - `bad_cast`
  - `bad_weak_ptr`
  - `bad_function_call`
  - `bad_alloc`
  - `bad_exception`
- **Checked Exceptions:** Used to represent errors that are expected and reasonable to recover from, so they are supposed to be declared by functions and caught and handled. These are all standard and user-defined exceptions that are not classified as "unchecked", i.e. they are instances or subclasses of one of the following standard exception type:
  - `exception`

– runtime\_error

“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 [14]]

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, 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 [13]]

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 271752 2017-03-23 12:07:07Z piotr.tanski $
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 [6]: 15-0-1 (Document) Exceptions shall only be used for error handling.
- C++ Core Guidelines [10]: E.3: Use exceptions for error handling only
- Effective Java 2nd Edition [14]: Item 57: Use exceptions only for exceptional conditions
- The C++ Programming Language [13], 13.1.1. Exceptions

### **Rule A15-0-2 (required, 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

```
1 //% $Id: A15-0-2.cpp 271927 2017-03-24 12:01:35Z piotr.tanski $
2 #include <stdint>
3 #include <cstring>
4 class C1
5 {
6     public:
7     C1(const C1& rhs)
8     {
9         copyBad(rhs);    // Non-compliant - if an exception is thrown, an object
10                        // will be left in an indeterminate state
11        copyGood(rhs);   // Compliant - full object will be properly copied or
12                        // none of its properties will be changed
13    }
14    ~C1() { delete[] e; }
15    void copyBad(const C1& rhs)
16    {
17        if (this != &rhs)
18        {
```

```
19     delete[] e;
20     e = nullptr; // e changed before the block where an exception can
21                 // be thrown
22     s = rhs.s; // s changed before the block where an exception can be
23               // thrown
24
25     if (s > 0)
26     {
27         e = new std::int32_t[s]; // If an exception will be thrown
28                                 // here, the
29                                 // object will be left in an indeterminate
30                                 // state
31         std::memcpy(e, rhs.e, s * sizeof(std::int32_t));
32     }
33 }
34 }
35 void copyGood(const C1& rhs)
36 {
37     std::int32_t* eTmp = nullptr;
38
39     if (rhs.s > 0)
40     {
41         eTmp = new std::int32_t[rhs.s]; // If an exception will be thrown
42                                           // here, the
43                                           // object will be left unchanged
44         std::memcpy(eTmp, rhs.e, rhs.s * sizeof(std::int32_t));
45     }
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++ [9]: 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 <stdint>
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, 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 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 271752 2017-03-23 12:07:07Z piotr.tanski $
2 #include <cstdlib>
3 #include <stdexcept>
4 class InvalidArguments : public std::logic_error // Compliant - invalid
5                                     // arguments error is
6                                     // "unchecked" exception
7 {
8     public:
9         using std::logic_error::logic_error;
10 };
11 class OutOfMemory : public std::bad_alloc // Compliant - insufficient memory
12                                     // error is "unchecked" exception
13 {
  
```

```
14 public:
15     using std::bad_alloc::bad_alloc;
16 };
17 class DivisionByZero : public std::logic_error // Compliant - division by zero
18                                     // error is "unchecked"
19                                     // exception
20 {
21     public:
22     using std::logic_error::logic_error;
23 };
24 class CommunicationError : public std::logic_error // Non-compliant -
25                                     // communication error
26                                     // should be "checked"
27 // exception but defined to be "unchecked"
28 {
29     public:
30     using std::logic_error::logic_error;
31 };
32 double division(std::int32_t a, std::int32_t b) noexcept(false)
33 {
34     // ...
35     if (b == 0)
36     {
37         throw DivisionByZero(
38             "Division by zero error"); // Unchecked exception thrown correctly
39     }
40
41     // ...
42 }
43 void allocate(std::uint32_t bytes) noexcept(false)
44 {
45     // ...
46     throw OutOfMemory(); // Unchecked exception thrown correctly
47 }
48 void initializeSocket() noexcept(false)
49 {
50     bool validParameters = true;
51
52     // ...
53     if (!validParameters)
54     {
55         throw InvalidArguments("Invalid parameters passed"); // Unchecked
56                                                         // exception
57                                                         // thrown
58                                                         // correctly
59     }
60 }
61 void sendData(std::int32_t socket) noexcept(false)
62 {
63     // ...
64     bool isSentSuccessfully = true;
```

```
65
66     // ...
67     if (!isSentSuccessfully)
68     {
69         throw CommunicationError("Could not send data"); // Unchecked exception
70                                                         // thrown when checked
71                                                         // exception should
72                                                         // be.
73     }
74 }
```

## 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, 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, in C++ represented by instances or subclasses of `std::exception` and `std::runtime_error` 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 [14]]

## Example

```
1 //% $Id: A15-0-5.cpp 271752 2017-03-23 12:07:07Z piotr.tanski $
2 #include <cstdlib>
3 #include <stdexcept>
4 class CommunicationError
5     : public std::exception // Compliant - communication error is "checked"
6 {
7     public:
8         explicit CommunicationError(const char* message) : msg(message) {}
9         CommunicationError(CommunicationError const&) noexcept = default;
```

```
10     CommunicationError& operator=(CommunicationError const&) noexcept = default;
11     ~CommunicationError() override = default;
12
13     const char* what() const noexcept override { return msg; }
14
15     private:
16         const char* msg;
17 };
18 class BusError
19     : public CommunicationError // Compliant - bus error is "checked"
20 {
21     public:
22         using CommunicationError::CommunicationError;
23 };
24 class Timeout : public std::runtime_error // Compliant - communication timeout
25                                     // is "checked"
26 {
27     public:
28         using std::runtime_error::runtime_error;
29 };
30 class PreconditionsError : public std::exception // Non-compliant - error on
31                                             // preconditions check should
32                                             // be "unchecked" but is
33                                             // defined to be "checked"
34 {
35     // Implementation
36 };
37 void fn1(std::uint8_t* buffer, std::uint8_t bufferLength) noexcept(false)
38 {
39     bool sentSuccessfully = true;
40
41     // ...
42     if (!sentSuccessfully)
43     {
44         throw CommunicationError(
45             "Could not send data"); // Checked exception thrown correctly
46     }
47 }
48 void fn2(std::uint8_t* buffer, std::uint8_t bufferLength) noexcept(false)
49 {
50     bool initSuccessfully = true;
51
52     if (!initSuccessfully)
53     {
54         throw PreconditionsError(); // An exception thrown on preconditions
55                                     // check failure should be "Unchecked", but
56                                     // PreconditionsError is "Checked"
57     }
58
59     // ...
60     bool sentSuccessfully = true;
```

```
61     bool isTimeout = false;
62
63     // ...
64     if (!sentSuccessfully)
65     {
66         throw BusError(
67             "Could not send data"); // Checked exception thrown correctly
68     }
69
70     // ...
71     if (isTimeout)
72     {
73         throw Timeout("Timeout reached"); // Checked exception thrown correctly
74     }
75 }
76 void fn3(std::uint8_t* buffer) noexcept(false)
77 {
78     bool isResourceBusy = false;
79
80     // ...
81     if (isResourceBusy)
82     {
83         throw std::runtime_error(
84             "Resource is busy now"); // Checked exception thrown correctly
85     }
86 }
```

## See also

- Effective Java: Item 58 - Use checked exceptions for recoverable conditions and runtime exceptions for programming errors.

### Rule A15-0-6 (required, verification, 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, implementation, 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

```
1 //% $Id: A15-0-7.cpp 271752 2017-03-23 12:07:07Z piotr.tanski $
2 #include <cstdlib>
3 #include <cstring>
4 struct __cxa_exception
5 {
6     // Exception's structure implementation
7 };
8 extern "C" void fatalError(const char* msg)
9 {
10     // Reports an error and terminates the program
11 }
12
13 extern "C" void* __cxa_allocate_exception_dynamically(size_t thrownSize)
14 {
15     size_t size = thrownSize + sizeof(__cxa_exception);
16     __cxa_exception* buffer = static_cast<__cxa_exception*>(
17         malloc(size)); // Non-compliant - dynamic memory allocation used
18
19     if (!buffer)
20     {
```

```
21     fatalError("Not enough memory to allocate exception!");
22 }
23
24     memset(buffer, 0, sizeof(__cxa_exception));
25     return buffer + 1;
26 }
27 extern "C" void* static_malloc(size_t size)
28 {
29     void* mem = NULL;
30     // Allocates memory using static memory pool
31     return mem;
32 }
33 extern "C" void* __cxa_allocate_exception_statically(size_t thrownSize)
34 {
35     size_t size = thrownSize + sizeof(__cxa_exception);
36     __cxa_exception* buffer = static_cast<__cxa_exception*>(static_malloc(
37         size)); // Compliant - memory allocation on static memory pool used
38
39     if (!buffer)
40     {
41         fatalError("Not enough memory to allocate exception!");
42     }
43
44     memset(buffer, 0, sizeof(__cxa_exception));
45     return buffer + 1;
46 }
```

## See also

- none

### **Rule A15-0-8 (required, implementation, 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 [6]: 15-0-1 (Document) Exceptions shall only be used for error handling.



- open-std.org [17]: ISO/IEC TR 18015:2006(E). Technical Report on C++ Performance

### 6.15.1 Throwing an exception

**Rule A15-1-1 (required, implementation, automated)**  
**Only instances of types derived from `std::exception` shall 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 a 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 [14]]

This means that only standard library exceptions or user-defined exceptions that inherit from `std::exception` base class shall 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 271715 2017-03-23 10:13:51Z piotr.tanski $
2  #include <memory>
3  #include <stdexcept>
4  class A
5  {
6      // Implementation
7  };
8  class MyException : public std::logic_error
9  {
10     public:
11         using std::logic_error::logic_error;
12         // Implementation
13 };
14 void f1()
15 {
16     throw - 1; // Non-compliant - integer literal thrown
17 }
18 void f2()
19 {
20     throw nullptr; // Non-compliant - null-pointer-constant thrown
21 }
22 void f3()
23 {
24     throw A(); // Non-compliant - user-defined type that does not inherit from
25                // std::exception thrown
26 }
27 void f4()
```

```

28 {
29     throw std::logic_error{
30         "Logic Error"}; // Compliant - std library exception thrown
31 }
32 void f5()
33 {
34     throw MyException{"Logic Error"}; // Compliant - user-defined type that
35                                         // inherits from std::exception thrown
36 }
37 void f6()
38 {
39     throw std::make_shared<std::exception>(
40         std::logic_error("Logic Error")); // Non-compliant - shared_ptr does
41                                         // not inherit from std::exception
42 }
43 void f7()
44 {
45     try
46     {
47         f6();
48     }
49
50     catch (std::exception& e) // An exception of
51                             // std::shared_ptr<std::exception> type will not
52                             // be caught here
53     {
54         // Handle an exception
55     }
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 [8]: 15.1.1 Only use instances of `std::exception` for exceptions
- C++ Core Guidelines [10]: E.14: Use purpose-designed user-defined types as exceptions (not built-in types)
- Effective Java 2nd Edition [14]: 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 271715 2017-03-23 10:13:51Z piotr.tanski $
2 #include <cstdlib>
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 [6]: 15-0-2 An exception object should not have pointer type.
- C++ Core Guidelines [10]: 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 [6]

**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 [6]

**Rule M15-1-2 (required, implementation, automated)**

**NULL shall not be thrown explicitly.**

See MISRA C++ 2008 [6]

**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 [6]

**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 271752 2017-03-23 12:07:07Z piotr.tanski $
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 [14]: 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

```
1  //% $Id: A15-1-4.cpp 272338 2017-03-28 08:15:01Z piotr.tanski $
2  #include <stdint>
3  #include <memory>
4  #include <stdexcept>
5  extern std::uint32_t f1();
6  void fVeryBad() noexcept(false)
7  {
8      std::logic_error* e = new std::logic_error("Logic Error 1");
9      // ...
10     std::uint32_t i = f1();
11
12     if (i < 10)
13     {
14         throw(*e); // Non-compliant - fVeryBad() is not able to clean-up
15                   // allocated memory
16     }
17
18     // ...
19     delete e;
20 }
21 void fBad() noexcept(false)
22 {
23     std::int32_t* x = new std::int32_t(0);
24     // ...
25     std::uint32_t i = f1();
26
27     if (i < 10)
```

```

28     {
29         throw std::logic_error("Logic Error 2"); // Non-compliant - exits from
30                                                    // fBad() without cleaning-up
31                                                    // allocated resources and
32                                                    // causes a memory leak
33     }
34
35     else if (i < 20)
36     {
37         throw std::runtime_error("Runtime Error 3"); // Non-compliant - exits
38                                                       // from fBad() without
39                                                       // cleaning-up allocated
40                                                       // resources and causes a
41                                                       // memory leak
42     }
43
44     // ...
45     delete x; // Deallocates claimed resource only in the end of fBad() scope
46 }
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++ [9]: ERR57-CPP. Do not leak resources when handling exceptions
- C++ Core Guidelines [10]: 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++ [9]: 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 <cstdlib>
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 };
33 static A a1; // Compliant - default constructor of type A is noexcept
34 static A a2(5); // Non-compliant - constructor of type A throws, and the
35 // exception will not be caught by the handler in main function
36 static A a3(5, 10); // Compliant - constructor of type A is noexcept, it
37 // handles exceptions internally
38 int main(int, char**)
39 {
40     try
41     {
42         // program code
43     }
44     catch (...)
45     {
46         // Handle exceptions
47     }
48
49     return 0;
50 }
```

## See also

- SEI CERT C++ [9]: 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

```
1  //% $Id: A15-2-2.cpp 271927 2017-03-24 12:01:35Z piotr.tanski $
2  #include <fstream>
3  #include <stdexcept>
4  class A
5  {
6  public:
7      A() = default;
8  };
9  class C1
10 {
11 public:
12     C1()
13     noexcept(false)
14         : a1(new A), a2(new A) // Non-compliant - if a2 memory allocation
15                               // fails, a1 will never be deallocated
16     {
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 [10]: 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 [6]

**Rule A15-3-1 (advisory, implementation, automated)**  
**Unchecked exceptions should be handled only in main or thread's main functions.**

### Rationale

Unchecked exceptions (e.g. `bad_alloc`, `out_of_range`, `length_error`, `invalid_argument`) either are a consequence of faulty logic within the program or are unpreventable and the program can not recover from them with meaningful action. In this case, propagate the exception up the call tree to the main (or thread's main) function where one common handler will be executed.

### Exception

This rule does not apply to C++ Standard Library, as it can provide different error handling mechanisms that depend on user input or usage.

### Example

```
1 //% $Id: A15-3-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
```

```
2 #include <stdint>
3 #include <stdexcept>
4 class OutOfMemory : public std::logic_error // Unchecked exception
5 {
6     public:
7     using std::logic_error::logic_error;
8 };
9 void f1() noexcept(false)
10 {
11     // ...
12     throw OutOfMemory("Not enough memory");
13 }
14 void f2() noexcept
15 {
16     try
17     {
18         f1();
19     }
20     catch (OutOfMemory& e) // Non-compliant - program is not able to handle an
21                             // OutOfMemory error in a meaningful way, the error
22                             // will still exist
23     {
24         // Handle an exception
25     }
26 }
27 void f3() noexcept(false)
28 {
29     // ...
30     try
31     {
32         // ...
33         f1();
34     }
35     catch (OutOfMemory& e)
36     {
37         // Nothing to do, just re-throw
38         throw; // Non-compliant - it is inefficient to catch and re-throw an
39                // error that can not be handled in f3()
40     }
41 }
42 void f4() noexcept(false)
43 {
44     // ...
45     f1(); // Compliant - OutOfMemory error can not be handled in f4()
46     // ...
47 }
48 int main(int, char**)
49 {
50     try
51     {
52         f4();
```



```

53     }
54     catch (OutOfMemory& e) // Compliant - OutOfMemory caught in main() function
55                             // so the program can clean-up and exit correctly
56     {
57         // Report the error and exit from the program correctly
58     }
59
60     return 0;
61 }
  
```

## See also

- none

### Rule A15-3-2 (required, implementation, non-automated)

**If a function throws a checked exception, it shall be handled when meaningful actions can be taken, otherwise it shall be propagated.**

## Rationale

Provide checked exception handlers only for functions that actually are able to take recovery actions. Implementing meaningless exception handlers that only re-throw caught exception results in a code that is inefficient and difficult to maintain.

## Example

```

1  // % $Id: A15-3-2.cpp 271752 2017-03-23 12:07:07Z piotr.tanski $
2  #include <stdint>
3  #include <iostream>
4  #include <stdexcept>
5  class CommunicationError : public std::exception
6  {
7      // Implementation
8  };
9  /// @throw CommunicationError Exceptional communication errors
10 extern void send(std::uint8_t* buffer) noexcept(false);
11 void sendData1(std::uint8_t* data) noexcept(false)
12 {
13     try
14     {
15         send(data);
16     }
17
18     catch (CommunicationError& e)
19     {
20         std::cerr << "Communication error occurred" << std::endl;
21         throw; // Non-compliant - exception is not handled, just re-throw
22     }
23 }
24 extern void busRestart() noexcept;
25 extern void bufferClean() noexcept;
26 void sendData2(std::uint8_t* data) noexcept(false)
  
```

```
27 {
28     try
29     {
30         send(data);
31     }
32
33     catch (CommunicationError& e)
34     {
35         std::cerr << "Communication error occurred" << std::endl;
36         bufferClean();
37         throw; // Compliant - exception is partially handled and re-thrown
38     }
39 }
40 void f1() noexcept
41 {
42     std::uint8_t* buffer = nullptr;
43
44     // ...
45     try
46     {
47         sendData2(buffer);
48     }
49
50     catch (CommunicationError& e)
51     {
52         std::cerr << "Communication error occurred" << std::endl;
53         busRestart();
54         // Compliant - including SendData2() exception handler, exception is now
55         // fully handled
56     }
57 }
58 void sendData3(std::uint8_t* data) noexcept
59 {
60     try
61     {
62         send(data);
63     }
64
65     catch (CommunicationError& e)
66     {
67         std::cerr << "Communication error occurred" << std::endl;
68         bufferClean();
69         busRestart();
70         // Compliant - exception is fully handled
71     }
72 }
```

## See also

- none

**Rule A15-3-3 (required, implementation, automated)****There shall be at least one exception handler to catch 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.

Note that the objective of the previous rule is that a program should catch all exceptions that it is expected to throw. This rule’s objective is to ensure that exceptions that were not expected are also caught.

**Example**

```
1  //% $Id: A15-3-3.cpp 271752 2017-03-23 12:07:07Z piotr.tanski $
2  #include <stdexcept>
3  int mainGood(int, char**) // Compliant
4  {
5      try
6      {
7          // program code
8      }
9      catch (std::runtime_error& e)
10     {
11         // Handle runtime errors
12     }
13     catch (std::logic_error& e)
14     {
15         // Handle logic errors
16     }
17     catch (std::exception& e)
18     {
19         // Handle all expected exceptions
20     }
21     catch (...)
22     {
23         // Handle all unexpected exceptions
24     }
25
26     return 0;
27 }
28 int mainBad(int,
29             char**) // Non-compliant - unexpected exceptions are not caught
30 {
31     try
32     {
```

```
33     // program code
34 }
35 catch (std::runtime_error& e)
36 {
37     // Handle runtime errors
38 }
39 catch (std::logic_error& e)
40 {
41     // Handle logic errors
42 }
43 catch (std::exception& e)
44 {
45     // Handle all expected exceptions
46 }
47
48 return 0;
49 }
50 void threadMainGood() // Compliant
51 {
52     try
53     {
54         // thread code
55     }
56     catch (std::exception& e)
57     {
58         // Handle all expected exception
59     }
60     catch (...)
61     {
62         // Handle all unexpected exception
63     }
64 }
65 void threadMainBad() // Non-compliant - unexpected exceptions are not caught
66 {
67     try
68     {
69         // thread code
70     }
71     catch (std::exception& e)
72     {
73         // Handle all expected exceptions
74     }
75
76     // Uncaught unexpected exception will cause an immediate program termination
77 }
```

## See also

- MISRA C++ 2008 [6]: 15-3-2 There should be at least one exception handler to catch all otherwise unhandled exceptions.
- SEI CERT C++ [9]: ERR51-CPP. Handle all exceptions

- Effective Java 2nd Edition [14]: 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 271752 2017-03-23 12:07:07Z piotr.tanski $
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 occurred
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 [6]

**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 [6]

**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

```
1 //% $Id: A15-3-5.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
2 #include <iostream>
3 #include <stdexcept>
4 class Exception : public std::runtime_error
5 {
6     public:
7         using std::runtime_error::runtime_error;
8         const char* what() const noexcept(true) override
9         {
10             return "Exception error message";
11         }
12 };
13 void fn()
14 {
15     try
16     {
17         // ...
18         throw std::runtime_error("Error");
19         // ...
20         throw Exception("Error");
21     }
22
23     catch (const std::logic_error& e) // Compliant - caught by const reference
```



```
24     {
25         // Handle exception
26     }
27     catch (std::runtime_error& e) // Compliant - caught by reference
28     {
29         std::cout << e.what() << "\n"; // "Error" or "Exception error message"
30         // will be printed, depending upon the
31         // actual type of thrown object
32         throw e; // The exception re-thrown is of its original type
33     }
34
35     catch (
36         std::runtime_error
37         e) // Non-compliant - derived types will be caught as the base type
38     {
39         std::cout
40             << e.what()
41             << "\n"; // Will always call what() method from std::runtime_error
42         throw e; // The exception re-thrown is of the std::runtime_error type,
43                 // not the original exception type
44     }
45 }
```

### See also

- MISRA C++ 2008 [6]: 15-3-5 A class type exception shall always be caught by reference.
- SEI CERT C++ [9]: ERR61-CPP. Catch exceptions by lvalue reference
- C++ Core Guidelines [10]: 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 [6]

#### **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 [6]

## 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.

**Example**

```
1 //% $Id: A15-4-1.cpp 270728 2017-03-16 10:38:20Z piotr.tanski $
2 #include <stdexcept>
3 void f1() noexcept; // Compliant - note that noexcept is the same as
4 // noexcept(true)
5 void f2() throw(); // Non-compliant - dynamic exception-specification is
6 // deprecated
7 void f3() noexcept(false); // Compliant
8 void f4() throw(std::runtime_error); // Non-compliant - dynamic
9 // exception-specification is deprecated
10 void f5() throw(
11     ...); // Non-compliant - dynamic exception-specification is deprecated
12 template <class T>
13 void f6() noexcept(noexcept(T())); // Compliant
```

**See also**

- C++ Core Guidelines [10]: E.12: Use `noexcept` when exiting a function because of a throw is impossible or unacceptable
- [open-std.org \[17\]](#): 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**

```
1  // % $Id: A15-4-2.cpp 271752 2017-03-23 12:07:07Z piotr.tanski $
2  #include <stdexcept>
3  // library.h
4  void libraryFunc();
5  // project.cpp
6  void f1() noexcept
7  {
8      // ...
9      throw std::runtime_error("Error"); // Non-compliant - f1 declared to be
10                                         // noexcept, but exits with exception.
11                                         // This leads to std::terminate() call
12 }
13 void f2() noexcept(true)
14 {
15     try
16     {
17         // ...
18         throw std::runtime_error(
19             "Error"); // Compliant - exception will not leave f2
20     }
21     catch (std::runtime_error& e)
22     {
23         // Handle runtime error
24     }
25 }
26 void f3() noexcept(false)
27 {
28     // ...
29     throw std::runtime_error("Error"); // Compliant
30 }
31 void f4() noexcept(
32     false) // Compliant - no information whether library_func() throws or not
33 {
34     libraryFunc();
35 }
```

**See also**

- MISRA C++ 2008 [6]: 15-5-3 The terminate() function shall not be called implicitly.
- HIC++ v4.0 [8]: 15.3.2 Ensure that a program does not result in a call to std::terminate
- SEI CERT C++ [9]: ERR50-CPP. Do not abruptly terminate the program

**Rule A15-4-3 (required, implementation, automated)**  
**Function's noexcept specification shall be either identical or more restrictive across all translation units and all overriders.**

## Rationale

If any declaration of a function has a noexcept specification, other declarations of the same function have to specify either the same or more restrictive noexcept specification. The same restriction apply to all overriders of a member function.

## Example

```
1  // % $Id: A15-4-3.cpp 270728 2017-03-16 10:38:20Z piotr.tanski $
2  // f1.hpp
3  void fn() noexcept;
4  // f1.cpp
5  // #include <f1.hpp>
6  void fn() noexcept // Compliant
7  {
8      // Implementation
9  }
10 // f2.cpp
11 // #include <f1.hpp>
12 // void fn() noexcept(false) // Non-compliant - different exception specifier
13 // {
14 // Implementation
15 // }
16 class A
17 {
18     public:
19         void f() noexcept;
20         void g() noexcept(false);
21         virtual void v1() noexcept = 0;
22         virtual void v2() noexcept(false) = 0;
23 };
24 void A::f() noexcept // Compliant
25 // void A::f() noexcept(false) // Non-compliant - different exception specifier
26 // than in declaration
27 {
28     // Implementation
29 }
30 void A::g() noexcept(false) // Compliant
31 // void A::g() noexcept // Non-compliant - different exception specifier than
32 // in declaration
33 {
```

```
34     // Implementation
35 }
36 class B : public A
37 {
38     public:
39     void v1() noexcept override // Compliant
40     // void v1() noexcept(false) override // Non-compliant - looser exception
41     // specifier in derived method,
42     // non-compilable
43     {
44         // Implementation
45     }
46     void v2() noexcept override // Compliant
47     {
48         // Implementation
49     }
50 };
```

## See also

- none

### **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 271715 2017-03-23 10:13:51Z piotr.tanski $
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 using the “@throw ExceptionName description” syntax, and they shall be identical in all function declarations and for all its overrides.

## 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 271752 2017-03-23 12:07:07Z piotr.tanski $
```

```
2 #include <stdint>
3 #include <stdexcept>
4 class CommunicationError : public std::exception
5 {
6     // Implementation
7 };
8 class BusError : public CommunicationError
9 {
10    // Implementation
11 };
12 class Timeout : public std::runtime_error
13 {
14     public:
15     using std::runtime_error::runtime_error;
16     // Implementation
17 };
18 /// @throw CommunicationError Communication error
19 /// @throw BusError Bus error
20 /// @throw Timeout On send timeout exception
21 void send1(
22     std::uint8_t* buffer,
23     std::uint8_t bufferLength) noexcept(false) // Compliant - All and only
24                                                 // those checked exceptions
25                                                 // that can be thrown are
26                                                 // specified
27 {
28     // ...
29     throw CommunicationError();
30     // ...
31     throw BusError();
32     // ...
33     throw Timeout("Timeout reached");
34     // ...
35 }
36 /// @throw CommunicationError Communication error
37 void send2(
38     std::uint8_t* buffer,
39     std::uint8_t bufferLength) noexcept(false) // Non-compliant - checked
40                                                 // exceptions that can be
41                                                 // thrown are missing from
42                                                 // specification
43 {
44     // ...
45     throw CommunicationError();
46     // ...
47     throw Timeout("Timeout reached");
48     // ...
49 }
50 class MemoryPartitioningError : std::exception
51 {
52     // Implementation
```



```
53 };
54 /// @throw CommunicationError Communication error
55 /// @throw BusError Bus error
56 /// @throw Timeout On send timeout exception
57 /// @throw MemoryPartitioningError Memory partitioning error prevents message
58 /// from being sent.
59 void send3(
60     std::uint8_t* buffer,
61     std::uint8_t bufferLength) noexcept(false) // Non-compliant - additional
62                                             // checked exceptions are
63                                             // specified
64 {
65     // ...
66     throw CommunicationError();
67     // ...
68     throw Timeout("Timeout reached");
69     // ...
70 }
```

## See also

- Effective Java 2nd Edition [14]: Item 62: Document all exceptions thrown by each method

### Rule A15-4-6 (advisory, implementation, automated)

**Unchecked exceptions should not be specified together with a function declaration.**

## Rationale

Unchecked exceptions are those which do not have an appropriate application-specific handling by the caller - it is only needed to catch them in main (or in task main functions). Specifying them is a significant overhead, while they do not bring added value and they restrict the evolution of functions. Such exceptions can occur anywhere in a program, and in a typical one they can be very numerous. Having to add such exceptions in every method declaration would reduce a program's clarity.

## Exception

Specifying unchecked exceptions in function declarations by C++ Standard Library does not violate this rule. Standard library can not know if an exception is meaningful for the caller.

## Example

```
1 //% $Id: A15-4-6.cpp 271752 2017-03-23 12:07:07Z piotr.tanski $
2 #include <stdint>
3 #include <stdexcept>
4 class InvalidInitParameters : public std::logic_error
5 {
6     public:
7         using std::logic_error::logic_error;
```

```
8 };
9 /// @throw InvalidInitParameters Error caused by passing invalid initialization
10 /// parameters
11 void send1(std::uint8_t* buffer, std::uint8_t bufferLength) noexcept (
12     false) // Non-compliant - unchecked exception documented
13 {
14     // ...
15     throw InvalidInitParameters("Invalid parameters");
16 }
17 void send2(std::uint8_t* buffer, std::uint8_t bufferLength) noexcept (
18     false) // Compliant - unchecked exception not documented
19 {
20     // ...
21     throw InvalidInitParameters("Invalid parameters");
22 }
```

## See also

- none

### 6.15.5 Special functions

#### **Rule A15-5-1 (required, implementation, automated)**

**A class destructor, “delete” operators, move constructor, move assignment operator and “swap” function shall not exit with an exception. They shall be all specified as “noexcept”.**

#### 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 constructor and move assignment operator are intended to be noexcept. If they throw exceptions, strong exception safety can not be guaranteed, because the original type values could be already modified or partially modified.

The standard-library containers and algorithms will not work correctly if a swap of two elements exits with an exception.

Note that if move constructor is not noexcept, then the standard library containers will use the copy constructor rather than the move constructor.

Note that it is acceptable for a destructor to throw an exception that is handled within this destructor, for example within a try-catch block.

Also, note that a destructor is noexcept by default, but the keyword noexcept needs to be specified to explicitly state that it can not throw any exception.

## Example

```
1  //% $Id: A15-5-1.cpp 270728 2017-03-16 10:38:20Z piotr.tanski $
2  #include <stdexcept>
3  class C1
4  {
5      public:
6          C1() = default;
7          C1(C1&& rhs)
8              noexcept // Compliant - move constructor declared to be noexcept
9              {
10             }
11         C1& operator=(C1&& rhs) noexcept // Compliant - move assignment operator
12                                         // declared to be noexcept
13         {
14             return *this;
15         }
16         ~C1() noexcept // Compliant - no exceptions thrown from destructor
17         {
18         }
19     };
20     void swap(C1& lhs, C1& rhs) noexcept // Compliant - swap function does not exit
21                                         // with an exception
22     {
23         // Implementation
24     }
25     class C2
26     {
27     public:
28         C2() = default;
29         C2(C2&& rhs)
30             noexcept // Compliant - move constructor declared to be noexcept
31             {
32                 try
33                 {
34                     // ...
35                     throw std::runtime_error(
36                         "Error"); // Exception will not leave move constructor
37                 }
38
39                 catch (std::exception& e)
40                 {
41                     // Handle runtime error
42                 }
43             }
44         C2& operator=(C2&& rhs) noexcept // Compliant - move assignment operator
45                                         // declared to be noexcept
46         {
47             try
48             {
49                 // ...
50                 throw std::runtime_error(
```

```
51         "Error"); // Exception will not leave assignment operator
52     }
53
54     catch (std::exception& e)
55     {
56         // Handle runtime error
57     }
58
59     return *this;
60 }
61 ~C2() // Non-compliant - the destructor does not contain the noexcept
62     // specification
63 {
64     try
65     {
66         // ...
67         throw std::runtime_error(
68             "Error"); // Exception will not leave the destructor
69     }
70
71     catch (std::exception& e)
72     {
73         // Handle runtime error
74     }
75 }
76 };
77 void swap(C2& lhs, C2& rhs) noexcept(
78     false) // Non-compliant - swap function declared to be noexcept(false)
79 {
80     // Implementation
81     throw std::runtime_error("Swap function failed"); // Non-compliant - swap
82                                                         // function exits with an
83                                                         // exception
84 }
85 class C3
86 {
87     public:
88     C3() = default;
89     C3(C3&& rhs) // Non-compliant - move constructor throws
90     {
91         // ...
92         throw std::runtime_error("Error");
93     }
94     C3& operator=(C3&& rhs) // Non-compliant - move assignment operator throws
95     {
96         // ...
97         throw std::runtime_error("Error");
98         return *this;
99     }
100     ~C3() // Non-compliant - destructor exits with an exception
101     {
```

```
102     throw std::runtime_error("Error");
103 }
104 static void operator delete(void* ptr, std::size_t sz)
105 {
106     // ...
107     throw std::runtime_error("Error"); // Non-compliant - operator delete
108                                         // exits with an exception
109 }
110 };
111 void fn()
112 {
113     C3 c1; // program terminates when c1 is destroyed
114     C3* c2 = new C3;
115     // ...
116     delete c2; // program terminates when c2 is deleted
117 }
```

### See also

- MISRA C++ 2008 [6]: 15-5-1 A class destructor shall not exit with an exception.
- HIC++ v4.0 [8]: 15.2.1 Do not throw an exception from a destructor
- C++ Core Guidelines [10]: E.16: Destructors, deallocation, and swap must never fail, C.85: Make swap noexcept

**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.

### Exception

Calling an `std::exit()` function from `main()` or from task main functions is acceptable, because it properly deallocates resources and calls `std::atexit()` handlers.

### Example

```
1 //% $Id: A15-5-2.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
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 [6]: 15-5-3 (Required) The terminate() function shall not be called implicitly.
- HIC++ v4.0 [8]: 15.3.2 Ensure that a program does not result in a call to std::terminate
- SEI CERT C++ [9]: 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 [15]):

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

### Example

```
1 //% $Id: A15-5-3.cpp 271752 2017-03-23 12:07:07Z piotr.tanski $
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;
33 }
34 void threadFunc()
35 {
36     A a; // Throws an exception from a's constructor and does not handle it in
37         // thread_func()
38 }
39 void f3()
40 {
41     try
42     {
43         std::thread t(&threadFunc); // Non-compliant - std::terminate() called
44                                     // on throwing an exception from
45                                     // thread_func()
46
47         if (f1())
48         {
49             throw std::logic_error("Error4");
50         }
51
52         else
53         {
54             f2(); // Non-compliant - std::terminate() called if there is no
55                 // active exception to be re-thrown by f2

```



```
56     }
57 }
58 catch (...)
59 {
60     B b; // Non-compliant - std::terminate() called on throwing an
61         // exception from b's destructor during exception handling
62
63     // ...
64     f2();
65 }
66 }
67 static A a; // Non-compliant - std::terminate() called on throwing an exception
68             // during program's start-up phase
69 int main(int, char**)
70 {
71     f3(); // Non-compliant - std::terminate() called if std::logic_error is
72         // thrown
73     return 0;
74 }
```

### See also

- MISRA C++ 2008 [6]: 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

```
1 // $Id: A16-0-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #pragma once // Non-compliant - implementation-defined feature
3
4 #ifndef HEADER_FILE_NAME // Compliant - include guard
5 #define HEADER_FILE_NAME // Compliant - include guard
```

```
6
7 #include <cstdint> // Compliant - unconditional file inclusion
8
9 #ifdef WIN32
10 #include <windows.h> // Compliant - conditional file inclusion
11 #endif
12
13 #ifdef WIN32
14 std::int32_t fn1(
15     std::int16_t x,
16     std::int16_t y) noexcept; // Non-compliant - not a file inclusion
17 #endif
18
19 #if defined VERSION && VERSION > 2011L // Compliant
20 #include <array> // Compliant - conditional file inclusion
21 #elif VERSION > 1998L // Compliant
22 #include <vector> // Compliant - conditional file inclusion
23 #endif // Compliant
24
25 #define MAX_ARRAY_SIZE 1024U // Non-compliant
26 #ifndef MAX_ARRAY_SIZE // Non-compliant
27 #error "MAX_ARRAY_SIZE has not been defined" // Non-compliant
28 #endif // Non-compliant
29 #undef MAX_ARRAY_SIZE // Non-compliant
30
31 #define MIN(a, b) ((a) < (b)) ? (a) : (b) // Non-compliant
32 #define PLUS2(X) ((X) + 2) // Non-compliant - function should be used instead
33 #define PI 3.14159F // Non-compliant - constexpr should be used instead
34 #define std ::int32_t long // Non-compliant - 'using' should be used instead
35 #define STARTIF if( // Non-compliant - language redefinition
36 #define HEADER "filename.h" // Non-compliant - string literal
37
38 void fn2() noexcept
39 {
40 #ifdef __linux__ // Non-compliant - ifdef not used for file inclusion
41
42 // ...
43
44 #elif WIN32 // Non-compliant - elif not used for file inclusion
45
46 // ...
47
48 #elif __APPLE__ // Non-compliant - elif not used for file inclusion
49
50 // ...
51
52 #else // Non-compliant - else not used for file inclusion
53
54 // ...
55
56 #endif // Non-compliant - endif not used for file inclusion or include guards
```

```
57 }  
58  
59 #endif // Compliant - include guard
```

## See also

- MISRA C++ 2008 [6]: Rule 16-2-1 The pre-processor shall only be used for file inclusion and include guards.
- MISRA C++ 2008 [6]: Rule 16-2-2 C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.
- JSF December 2005 [7]: AV Rule 26 Only the following pre-processor directives shall be used: 1. #ifndef 2. #define 3. #endif 4. #include.
- JSF December 2005 [7]: 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 [7]: 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 [7]: 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 [7]: 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 [7]: 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 [7]: AV Rule 32 The #include pre-processor directive will only be used to include header (\*.h) files.
- HIC++ v4.0 [8]: 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 [6]

**Rule M16-0-2 (required, implementation, automated)**  
**Macros shall only be #define'd or #undef'd in the global namespace.**

See MISRA C++ 2008 [6]

**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 [6]

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 [6]

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 [6]

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 [6]

### 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 [6]

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 [6]

Note: “`#if`”, “`#elif`”, “`#else`” and “`#ifdef`” 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 [6]

**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

```
1 // $Id: A16-2-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2
3 // #include <directory/headerfile.hpp> // Compliant
4 // #include <headerfile.hpp> // Compliant
5 // #include "directory/headerfile.hpp" // Compliant
6 // #include "headerfile.hpp" // Compliant
7 // #include <directory/*.hpp> // Non-compliant
8 // #include <header'file.hpp> // Non-compliant
9 // #include <"headerfile.hpp"> // Non-compliant
10 // #include <directory\\headerfile.hpp> // Non-compliant
```

#### See also

- MISRA C++ 2008 [6]: Rule 16-2-4 The `'`, `"`, `/*` or `/*/*` characters shall not occur in a header file name.
- MISRA C++ 2008 [6]: 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.

**Example**

```
1 // $Id: A16-2-2.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <algorithm> // Non-compliant - nothing from algorithm header file is
   used
3 #include <array> // Non-compliant - nothing from array header file is used
4 #include <cstdint> // Compliant - std::int32_t, std::uint8_t are used
5 #include <iostream> // Compliant - cout is used
6 #include <stdexcept> // Compliant - out_of_range is used
7 #include <vector> // Compliant - vector is used
8 void fn1() noexcept
9 {
10     std::int32_t x = 0;
11     // ...
12     std::uint8_t y = 0;
13     // ...
14 }
15 void fn2() noexcept(false)
16 {
17     try
18     {
19         std::vector<std::int32_t> v;
20         // ...
21         std::uint8_t idx = 3;
22         std::int32_t value = v.at(idx);
23     }
24     catch (std::out_of_range& e)
25     {
26         std::cout << e.what() << '\n';
27     }
28 }
```

**See also**

- HIC++ v4.0 [8]: 16.1.5 Include directly the minimum number of headers required for compilation.

**Rule A16-2-3 (required, implementation, non-automated)**  
**All used include directives shall be explicitly stated.****Rationale**

All header files that define types used in a file should be included explicitly. Relying on inclusion dependencies of other header files makes the code more difficult to maintain.

Note that in header files some include directives could be easily replaced with forward declarations.

### Example

```
1 // $Id: A16-2-3.hpp 271696 2017-03-23 09:23:09Z piotr.tanski $
2 #ifndef HEADER_HPP
3 #define HEADER_HPP
4
5 #include <array>
6 #include <cstdint>
7
8 class B; // Compliant - type B can be included using forward declaration
9         // std::into
10        // the header file
11
12 class OutOfRangeException
13     : public std::out_of_range // Non-compliant - <stdexcept> which defines
14                               // out_of_range included
15                               // implicitly through <array>
16 {
17     public:
18         using std::out_of_range::out_of_range;
19 };
20
21 class A
22 {
23     public:
24         // Interface of class A
25
26     private:
27         std::array<std::uint32_t, 10>
28             m_array; // Compliant - <array> included explicitly
29         B* m_b;
30         std::int32_t m_x; // Compliant - <cstdint> included explicitly
31 };
32
33 #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 [6]

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 [6]

### 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

```
1 // $Id: A16-6-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <stdint>
3 #include <type_traits>
4 constexpr std::int32_t value = 0;
5 #if value > 10
6 #error "Incorrect value" // Non-compliant
7 #endif
8 void f1() noexcept
9 {
10     static_assert(value <= 10, "Incorrect value"); // Compliant
11     // ...
12 }
13 template <typename T>
14 void f2(T& a)
15 {
16     static_assert(std::is_copy_constructible<T>::value,
17                 "f2() function requires copying"); // Compliant
18     // ...
19 }
```

#### 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 [6]: 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 [6]: 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 [6]

**Rule M17-0-3 (required, implementation, automated)**  
**The names of standard library functions shall not be overridden.**

See MISRA C++ 2008 [6]

**Rule A17-0-2 (required, implementation, 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 [6]

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 271687 2017-03-23 08:57:35Z piotr.tanski $
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);
18 }
19
20 void fn2() noexcept
21 {
22     try
23     {
24         fn1("filename.txt"); // Compliant - fn1() allows you to use C code like
25                               // C++ code
26
27         // ...
28     }
```

```
29     catch (std::system_error& e)
30     {
31         std::cerr << "Error: " << e.code() << " - " << e.what() << '\n';
32     }
33 }
34
35 std::int32_t fn3(const char* filename) noexcept // Non-compliant - placing C
36 // functions calls along with C++
37 // code forces a developer to be
38 // responsible for C-specific error
39 // handling, explicit resource
40 // cleanup, etc.
41 {
42     FILE* handle = fopen(filename, "rb");
43     if (handle == NULL)
44     {
45         std::cerr << "An error occurred: " << errno << " - " << strerror(errno)
46             << '\n';
47         return errno;
48     }
49
50     try
51     {
52         // ...
53         fclose(handle);
54     }
55     catch (std::system_error& e)
56     {
57         fclose(handle);
58     }
59     catch (std::exception& e)
60     {
61         fclose(handle);
62     }
63
64     return errno;
65 }
```

## See also

- MISRA C++ 2008 [6]: Rule 19-3-1 The error indicator `errno` shall not be used.
- HIC++ v4.0 [8]: 17.2.1 Wrap use of the C Standard Library.
- JSF December 2005 [7]: 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.

## 6.18 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 [6]: Rule 18-0-1 (Required) The C library shall not be used.
- HIC++ v4.0 [8]: 1.3.3 Do not use the C Standard Library .h headers.

**Rule A18-0-2 (required, implementation, automated)**  
**The library functions atof, atoi and atol from library <cstdlib> shall not be used.**

#### Rationale

These functions have undefined behavior associated with them when the string cannot be converted.

Since C++11 Language Standard, new numeric conversion functions (See: `std::stoi`, `std::stol`, `std::stoll` [15]) were introduced. They guarantee defined behavior.

#### Example

```
1 // $Id: A18-0-2.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <cstdint>
3 #include <cstdlib>
4 #include <string>
5 std::int32_t f1(const char* str) noexcept
6 {
7     return atoi(str); // Non-compliant - undefined behavior if str can not
8                       // be converted
9 }
10 std::int32_t f2(std::string const& str) noexcept(false)
11 {
12     return std::stoi(str); // Compliant - throws a std::invalid_argument
13                            // exception if str can not be converted
14 }
```

#### See also

- MISRA C++ 2008 [6]: Rule 18-0-2 The library functions `atof`, `atoi` and `atol` from library `<cstdlib>` shall not be used.

**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 [6]

**Rule M18-0-4 (required, implementation, automated)**  
**The time handling functions of library `<ctime>` shall not be used.**

See MISRA C++ 2008 [6]

**Rule M18-0-5 (required, implementation, automated)**  
**The unbounded functions of library `<cstring>` shall not be used.**

See MISRA C++ 2008 [6]

**Rule A18-0-3 (required, implementation, automated)**  
**The library `<locale>` (`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`, `wscoll`, `iscntrl`, `iswalpha`, `iswspace`, `strcoll`, `wctod`, `isdigit`, `iswblank`, `iswupper`, `strerror`, `wctombs`, `isgraph`, `iswcntrl`, `iswxdigit`, `strtod`, `wcsxfrm`, `islower`, `iswctype`, `isxdigit`, `strxfrm`, `wctomb`.

## See also

- JSF December 2005 [7]: AV Rule 19 `<locale.h>` and the `setlocale` function shall not be used.

### 6.18.1 Types

**Rule A18-1-1 (advisory, implementation, automated)**  
**C-style arrays should not be used.**

## Rationale

C-style array is implicitly convertible to raw pointer and easily loses information about its size. This construct is unsafe, unmaintainable and is a source of potential errors.

For fixed-size, stack-allocated arrays, `std::array` is supposed to be used instead. The STL library `std::array` is designed to work with STL 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 271752 2017-03-23 12:07:07Z piotr.tanski $
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 [10]: ES.27: Use `std::array` or `stack_array` for arrays on the stack.
- C++ Core Guidelines [10]: 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>` 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 other types.

The C++ Language Standard guarantees that elements of an STL container can be safely concurrently modified, except for an `std::vector<bool>`.

Note that fixed-size `std::array` of bools, `std::deque<bool>` or creating POD wrapper for bool type and using it with `std::vector` are possible alternatives.

### Example

```

1 // $Id: A18-1-2.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <stdint>
3 #include <vector>
4 void fn() noexcept
5 {
6     std::vector<std::uint8_t> v1; // Compliant
7     std::vector<bool> v2; // Non-compliant
8 }

```

## See also

- HIC++ v4.0 [8]: 17.1.1 Do not use `std::vector<bool>`.

**Rule A18-1-3 (required, implementation, automated)**  
The `std::auto_ptr` shall not be used.

## Rationale

The `std::auto_ptr` smart pointer is deprecated since C++11 Language Standard and it is planned to be withdrawn in C++17 Language Standard.

The `std::auto_ptr` provides unusual copy semantics and it can not be placed in STL containers. It is recommended to use `std::unique_ptr` instead.

## Example

```

1 // $Id: A18-1-3.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <stdint>
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 [8]: 1.3.4 Do not use deprecated STL library features.
- cppreference.com [15]: `std::auto_ptr`.

**Rule A18-1-4 (required, implementation, automated)**  
The `std::shared_ptr` shall not refer to an array type.

## Rationale

Memory allocated for array type needs to be deallocated using `delete[]` syntax. Shared pointers do not have such information, and it is not possible to pass a custom array deleter to `std::make_shared` function.



An `std::array` or `std::vector` can be used instead of the raw array type.

Note that it is allowed to use the `std::unique_ptr` with `T[]` template argument.

### Example

```

1 // $Id: A18-1-4.cpp 271752 2017-03-23 12:07:07Z piotr.tanski $
2 #include <array>
3 #include <memory>
4 #include <vector>
5 class A
6 {
7 };
8 void f1() noexcept(false)
9 {
10     A cArray[10]; // Compliant
11     std::shared_ptr<A> ptr1(
12         cArray); // Non-compliant - ptr1 will not delete all of its elements
13     std::shared_ptr<A> ptr2(
14         new A[10]); // Non-compliant - ptr2 will not delete all of its elements
15     // std::shared_ptr<A[]> ptr3(new A[10]); // Non-compliant - compilation
16     // error
17     std::shared_ptr<std::array<A, 10>> ptr4 =
18         std::make_shared<std::array<A, 10>>(); // Compliant
19     std::shared_ptr<std::vector<A>> ptr5 =
20         std::make_shared<std::vector<A>>(10, A()); // Compliant
21 }
22 void f2() noexcept(false)
23 {
24     // std::unique_ptr<A> ptr1 = std::make_unique<A>(10); // Non-compliant - no
25     // such constructor in class A
26     std::unique_ptr<A> ptr2{
27         new A[10]}; // Non-compliant - ptr2 will not delete all of its elements
28     std::unique_ptr<A[]> ptr3 =
29         std::make_unique<A[]>(10); // Compliant - std::unique_ptr provides a
30         // specialization for T[] types
31 }

```

### See also

- HIC++ v4.0 [8]: 17.3.4 Do not create smart pointers of array type.

**Rule A18-1-5 (required, implementation, automated)**  
 The `std::unique_ptr` shall not be passed to a function by const reference.

### Rationale

A parameter of type `const std::unique_ptr&` provides constness benefits for `std::unique_ptr` only, not for an object it is pointing to. This may lead to confusion whether a function is allowed to modify the underlying pointer or not.

Instead, const pointer or const reference to the underlying object should be passed.

## Example

```
1 // $Id: A18-1-5.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <cstdint>
3 #include <memory>
4
5 void fn1(std::unique_ptr<std::int32_t> ptr) // Compliant
6 {
7     // fn1() now owns ptr
8 }
9
10 void fn2(std::unique_ptr<std::int32_t>& ptr) // Compliant
11 {
12     // fn2() is explicitly allowed to make a ptr to refer to a different object
13 }
14
15 void fn3(const std::unique_ptr<std::int32_t>& ptr) // Non-compliant
16 {
17     // fn3 takes ptr by const reference but still it is able to make a ptr to
18     // refer to a different object
19
20     *ptr = 10; // No compilation error
21 }
22
23 void fn4(const std::int32_t* ptr) // Compliant
24 {
25     // fn4 takes a const raw pointer
26
27     //*ptr = 10; // Compilation error
28 }
29
30 void fn5()
31 {
32     fn1(std::make_unique<std::int32_t>(0));
33
34     std::unique_ptr<std::int32_t> ptr = std::make_unique<std::int32_t>(0);
35     fn2(ptr);
36     fn3(ptr);
37     fn4(ptr.get());
38 }
```

## See also

- HIC++ v4.0 [8]: 8.2.4 Do not pass `std::unique_ptr` by const reference.
- C++ Core Guidelines [10]: R.33: Take a `unique_ptr<widget>&` parameter to express that a function reseats the widget.

## 6.18.2 Implementation properties

**Rule M18-2-1 (required, implementation, automated)**  
**The macro offsetof shall not be used.**

See MISRA C++ 2008 [6]

### 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**

**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 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <stdint>
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 [8]: 5.3.2 Allocate memory using new and release it using delete.
- C++ Core Guidelines [10]: R.10: Avoid malloc() and free().

**Rule A18-5-2 (required, implementation, partially automated)**  
**Operators new and delete shall not be called explicitly.**

## Rationale

If a resource returned by operator `new` 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`

## Exception

If the result of explicit resource allocation using `new` operator is immediately given to a manager object or a RAII class which does not provide a safe alternative for memory allocation, then it is not a violation of the rule.

This rule does not apply to dynamic memory allocation/deallocation performed in user-defined RAII classes and managers.

## Example

```
1 // $Id: A1-7-2.cpp 270728 2017-03-16 10:38:20Z piotr.tanski $
2 #include <cstdint>
3 #include <memory>
4 #include <vector>
5 std::int32_t fn1()
6 {
7     std::int32_t errorCode{0};
8
9     std::int32_t* ptr =
10         new std::int32_t{0}; // Non-compliant - new called explicitly
11     // ...
12     if (errorCode != 0)
13     {
14         throw std::runtime_error{"Error"}; // Memory leak could occur here
15     }
16     // ...
17
18     if (errorCode != 0)
19     {
20         return 1; // Memory leak could occur here
21     }
22     // ...
23     return errorCode; // Memory leak could occur here
24 }
25 std::int32_t fn2()
26 {
27     std::int32_t errorCode{0};
```

```
28
29     std::unique_ptr<std::int32_t> ptr1 = std::make_unique<std::int32_t>(
30         0); // Compliant - alternative for 'new std::int32_t(0)'  
31
32     std::unique_ptr<std::int32_t> ptr2(new std::int32_t{
33         0}); // Non-compliant - unique_ptr provides make_unique
34             // function which shall be used instead of explicit
35             // new operator  
36
37     std::shared_ptr<std::int32_t> ptr3 =
38         std::make_shared<std::int32_t>(0); // Compliant  
39
40     std::vector<std::int32_t> array; // Compliant
41                                     // alternative for dynamic array  
42
43     if (errorCode != 0)
44     {
45         throw std::runtime_error{"Error"}; // No memory leaks
46     }
47     // ...
48     if (errorCode != 0)
49     {
50         return 1; // No memory leaks
51     }
52     // ...
53     return errorCode; // No memory leaks
54 }
55 template <typename T>
56 class ObjectManager
57 {
58     public:
59     explicit ObjectManager(T* obj) : object{obj} {}
60     ~ObjectManager() { delete object; }
61     // Implementation
62
63     private:
64     T* object;
65 };
66 std::int32_t fn3()
67 {
68     std::int32_t errorCode{0};
69
70     ObjectManager<std::int32_t> manager{
71         new std::int32_t{0}}; // Compliant by exception
72     if (errorCode != 0)
73     {
74         throw std::runtime_error{"Error"}; // No memory leak
75     }
76     // ...
77     if (errorCode != 0)
78     {
```

```

79     return 1; // No memory leak
80 }
81 // ...
82 return errorCode; // No memory leak
83 }
  
```

## See also

- C++ Core Guidelines [10]: R.11: Avoid calling new and delete explicitly.
- C++ Core Guidelines [10]: R.12: Immediately give the result of an explicit resource allocation to a manager object.
- C++ Core Guidelines [10]: ES.60: Avoid new and delete outside resource management functions.

**Rule A18-5-3 (required, implementation, automated)**  
**The form of delete operator shall match the form of new operator used to allocate the memory.**

## Rationale

Plain and array forms of new and delete operators must not be mixed. If new or new[] operator was used to allocate the memory, then respectively delete or delete[] operator is supposed to be used to deallocate it.

## Example

```

1 // $Id: A18-5-3.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <cstdint>
3 void fn1()
4 {
5     std::int32_t* array =
6         new std::int32_t[10]; // new[] operator used to allocate the
7                               // memory for an array
8     // ...
9     delete array; // Non-compliant - delete[] operator supposed to be used
10 }
11 void fn2()
12 {
13     std::int32_t* object = new std::int32_t{0}; // new operator used to
14                                                  // allocate the memory for an
15                                                  // integer type
16     // ...
17     delete[] object; // Non-compliant - delete operator supposed to be used
18 }
19 void fn3()
20 {
21     std::int32_t* object = new std::int32_t{0};
22     std::int32_t* array = new std::int32_t[10];
23     // ...
24     delete[] array; // Compliant
  
```



```
25     delete object;    // Compliant
26 }
```

### See also

- HIC++ v4.0 [8]: 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: A1-7-4.cpp 270728 2017-03-16 10:38:20Z piotr.tanski $
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, implementation, 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.

### Example

```
1 //% $Id: A18-5-1.cpp 270728 2017-03-16 10:38:20Z piotr.tanski $
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* (*libc_malloc)(size_t) = dlsym(RTLD_NEXT, "malloc");
11     return libc_malloc(size);
12 }
13
14 void freeBad(void* ptr) // Non-compliant, malloc from libc does not guarantee
15                       // deterministic execution time
16 {
17     void (*libc_free)(void*) = dlsym(RTLD_NEXT, "free");
18     libc_free(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, implementation, 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 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: A18-5-3.cpp 270728 2017-03-16 10:38:20Z piotr.tanski $
2  #include <stdint>
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

## 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 [6]

**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

```
1 // $Id: A18-9-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <stdint>
3 #include <functional>
4 class A
5 {
6     // Implementation
7 };
8 void fn(A const& a, double y) noexcept
9 {
10     // Implementation
11 }
12 void f1() noexcept
13 {
14     double y = 0.0;
15     auto function = std::bind(&fn, std::placeholders::_1, y); // Non-compliant
16     // ...
17     A const a{};
18     function(a);
19 }
20 void f2() noexcept
21 {
22     auto lambda = [](A const& a) -> void {
23         double y = 0.0;
24         fn(a, y);
25     }; // Compliant
26     // ...
27     A const a{};
28     lambda(a);
29 }
```

### See also

- Effective Modern C++ [12]: 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 271927 2017-03-24 12:01:35Z piotr.tanski $
2 #include <stdint>
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>
31 void fn3(T&& param)
32 {
33     fn1(std::move(param)); // Non-compliant - forwarding forwarding reference
```

```
34             // via std::move
35     }
36     void fn4() noexcept
37     {
38         B b1;
39         B& b2 = b1;
40         fn2(b2);           // fn1(const B&) is called
41         fn2(std::move(b1)); // fn1(B&&) is called
42         fn3(b2);           // fn1(B&&) is called
43         fn3(std::move(b1)); // fn1(B&&) is called
44     }
```

### See also

- HIC++ v4.0 [8]:17.3.2 Use std::forward to forward universal references
- Effective Modern C++ [12]: 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

```
1 // $Id: A18-9-3.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <utility>
3 class A
4 {
5     // Implementation
6 };
7 void f1()
8 {
9     const A a1{};
10    A a2 = a1;           // Compliant - copy constructor is called
11    A a3 = std::move(a1); // Non-compliant - copy constructor is called
12                        // implicitly instead of move constructor
13 }
```

### See also

- HIC++ v4.0 [8]: 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 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <cstdlib>
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 [8]: 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 [6]

## 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

```
1  //% $Id: A23-0-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2  #include <cstdlib>
3  #include <vector>
4
5  void fn1(std::vector<std::int32_t>& v) noexcept
6  {
7      for (std::vector<std::int32_t>::const_iterator iter{v.cbegin()},
8           end{v.cend()};
9           iter != end;
10          ++iter) // Compliant
11      {
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 [8]: 17.4.1 Use const container calls when result is immediately converted to a const iterator.

## 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 [6]

**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.

#### Example

```
1 // $Id: A27-0-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <string.h>
3 #include <stdint>
4 #include <stdio>
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)
23     {
24         // Handle an error
25     }
26     else if (ret >= len)
27     {
28         // Handle truncated output
```

```
29     }
30
31     fprintf(stderr, msg);
32     delete[] msg;
33 }
34 void f2(const char* name)
35 {
36     static const char format[] = "Name: %s .";
37     fprintf(stderr, format, name); // Compliant - untrusted input passed as one
38                                   // of the variadic arguments, not as part of
39                                   // vulnerable format string
40 }
```

## See also

- SEI CERT C++ [9]: FIO30-C. Exclude user input from format strings.

### Rule A27-0-2 (required, 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.

## Example

```
1 // $Id: A27-0-2.cpp 270728 2017-03-16 10:38:20Z piotr.tanski $
2 #include <iostream>
3 #include <string>
4 void f1() noexcept
5 {
6     char buffer[10];
7     std::cin >> buffer; // Non-compliant - this could lead to a buffer overflow
8 }
9 void f2() noexcept
10 {
11     std::string string1;
12     std::string string2;
13     std::cin >> string1 >> string2; // Compliant - no buffer overflows
14 }
15 void f3(std::istream& in) noexcept
16 {
17     char buffer[32];
18
19     try
20     {
21         in.read(buffer, sizeof(buffer));
```

```
22     }
23
24     catch (std::ios_base::failure&)
25     {
26         // Handle an error
27     }
28
29     std::string str(buffer); // Non-compliant - if 'buffer' is not null
30                             // terminated, then constructing std::string leads
31                             // to undefined behavior.
32 }
33 void f4(std::istream& in) noexcept
34 {
35     char buffer[32];
36
37     try
38     {
39         in.read(buffer, sizeof(buffer));
40     }
41
42     catch (std::ios_base::failure&)
43     {
44         // Handle an error
45     }
46
47     std::string str(buffer, in.gcount()); // Compliant
48 }
```

### See also

- SEI CERT C++ [9]: STR50-CPP. Guarantee that storage for strings has sufficient space for character data and the null terminator.

## 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 26262-8, Road vehicles - Functional safety - Part 8: Supporting processes*, International Organization for Standardization, 2011.
- [6] *MISRA C++:2008 Guidelines for the use of the C++ language in critical systems*, The Motor Industry Software Reliability Association, 2008.
- [7] *Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program*, Document Number 2RDU00001 Rev C, Lockheed Martin Corporation, 2005.
- [8] *High Integrity C++ Coding Standard Version 4.0*, Programming Research Ltd, 2013.
- [9] *Software Engineering Institute CERT C++ Coding Standard*, Software Engineering Institute Division at Carnegie Mellon University, 2016.
- [10] Bjarne Stroustrup, Herb Sutter, *C++ Core Guidelines*, 2017.
- [11] *Google C++ Style Guide*, Google, 2017.
- [12] Scott Meyers, *Effective Modern C++*, ISBN: 978-1-491-90399-5, O'Reilly, 2015.
- [13] Bjarne Stroustrup, *The C++ Programming Language, Fourth Edition*, ISBN: 978-0-321-56384-2, Addison-Wesley, 2013.
- [14] Joshua Bloch, *Effective Java, Second Edition*, ISBN: 978-0321356680, Addison-Wesley, 2008
- [15] *cppreference.com, online reference for the C and C++ languages and standard libraries*, 2017
- [16] *stackoverflow.com, community of programmers*, 2017
- [17] *open-std.org, site holding a number of web pages for groups producing open standards*, 2017

## A 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. Implemented (only for ISO 26262): An ISO 26262 clause is implemented by the AUTOSAR C++14 rules.
6. Not yet analyzed: The rule is not yet analyzed in the current release.

### A.1 Traceability to MISRA C++:2008

MISRA C++:2008 [6] 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	<a href="#">M0-1-1</a>	-
0-1-2 (Required) A project shall not contain infeasible paths.	2 - Small differences	<a href="#">M0-1-2</a>	Note about constexpr functions added.
0-1-3 (Required) A project shall not contain unused variables.	1 - Identical	<a href="#">M0-1-3</a>	-
0-1-4 (Required) A project shall not contain non-volatile POD variables having only one use.	1 - Identical	<a href="#">M0-1-4</a>	-
0-1-5 (Required) A project shall not contain unused type declarations.	1 - Identical	<a href="#">M0-1-5</a>	-

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	<a href="#">A0-1-1</a>	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	<a href="#">A0-1-2</a>	Rationale reformulated.
0-1-8 (Required) All functions with void return type shall have external side effect(s).	1 - Identical	<a href="#">M0-1-8</a>	-
0-1-9 (Required) There shall be no dead code.	1 - Identical	<a href="#">M0-1-9</a>	-
0-1-10 (Required) Every defined function shall be called at least once.	3 - Significant differences	<a href="#">M0-1-10</a> , <a href="#">A0-1-2</a>	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.	1 - Identical	<a href="#">M0-1-11</a>	-
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.	1 - Identical	<a href="#">M0-1-12</a>	-
0-2-1 (Required) An object shall not be assigned to an overlapping object.	1 - Identical	<a href="#">M0-2-1</a>	-
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	<a href="#">M0-3-1</a>	-
0-3-2 (Required) If a function generates error information, then that error information shall be tested.	1 - Identical	<a href="#">M0-3-2</a>	-
0-4-1 (Document) Use of scaled-integer or fixed-point arithmetic shall be documented.	1 - Identical	<a href="#">M0-4-1</a>	-
0-4-2 (Document) Use of floating-point arithmetic shall be documented.	1 - Identical	<a href="#">M0-4-2</a>	-

0-4-3 (Document) Floating-point implementations shall comply with a defined floating-point standard.	3 - Significant differences	<a href="#">A0-4-1</a>	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	<a href="#">A1-1-1</a>	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	<a href="#">M1-0-2</a>	-
1-0-3 (Document) The implementation of integer division in the chosen compiler shall be determined and documented.	3 - Significant differences	<a href="#">A0-4-2</a>	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.	4 - Rejected	-	Rule rejected. The character set explicitly specified in <a href="#">A2-2-1</a> .
2-3-1 (Required) Trigraphs shall not be used.	2 - Small differences	<a href="#">A2-5-1</a>	All trigraphs listed in rationale. Example extended.
2-5-1 (Advisory) Digraphs should not be used.	3 - Significant differences	<a href="#">A2-6-1</a>	Obligation level changed to “Required”.
2-7-1 (Required) The character sequence /* shall not be used within a C-style comment.	3 - Significant differences	<a href="#">A2-8-4</a>	Using the C-style comments is not allowed.
2-7-2 (Required) Sections of code shall not be commented out using C-style comments.	2 - Small differences	<a href="#">A2-8-1</a>	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	<a href="#">A2-8-1</a>	Obligation level changed to “Required”. Commenting-out code sections is not allowed.
2-10-1 (Required) Different identifiers shall be typographically unambiguous.	1 - Identical	<a href="#">M2-10-1</a>	-



2-10-2 (Required) Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.	2 - Small differences	<a href="#">A2-11-1</a>	Added a note to rationale. Example extended.
2-10-3 (Required) A typedef name (including qualification, if any) shall be a unique identifier.	1 - Identical	<a href="#">M2-10-3</a>	-
2-10-4 (Required) A class, union or enum name (including qualification, if any) shall be a unique identifier.	2 - Small differences	<a href="#">A2-11-3</a>	“A class, union or enum name” changed to “A user-defined type name”. Example extended.
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	<a href="#">A2-11-4</a>	Obligation level changed to “Required”. Rationale reformulated.
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.	1 - Identical	<a href="#">M2-10-6</a>	-
2-13-1 (Required) Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.	2 - Small differences	<a href="#">A2-14-1</a>	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	<a href="#">M2-13-2</a>	-
2-13-3 (Required) A “U” suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.	1 - Identical	<a href="#">M2-13-3</a>	-
2-13-4 (Required) Literal suffixes shall be upper case.	1 - Identical	<a href="#">M2-13-4</a>	-
2-13-5 (Required) Narrow and wide string literals shall not be concatenated.	2 - Small differences	<a href="#">A2-14-2</a>	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	<a href="#">A3-1-1</a>	Rationale reformulated. Example extended.
3-1-2 (Required) Functions shall not be declared at block scope.	1 - Identical	<a href="#">M3-1-2</a>	-
3-1-3 (Required) When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.	2 - Small differences	<a href="#">A3-1-4</a>	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	<a href="#">M3-2-1</a>	-

3-2-2 (Required) The One Definition Rule shall not be violated.	1 - Identical	<a href="#">M3-2-2</a>	-
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	<a href="#">M3-2-3</a>	-
3-2-4 (Required) An identifier with external linkage shall have exactly one definition.	1 - Identical	<a href="#">M3-2-4</a>	-
3-3-1 (Required) Objects or functions with external linkage shall be declared in a header file.	2 - Small differences	<a href="#">A3-3-1</a>	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	<a href="#">M3-3-2</a>	-
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	<a href="#">M3-4-1</a>	-
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	<a href="#">M3-9-1</a>	-
3-9-2 (Advisory) typedefs that indicate size and signedness should be used in place of the basic numerical types.	3 - Significant differences	<a href="#">M3-9-1</a>	Rule title and rationale reformulated to use types from <code>&lt;stdint.h&gt;</code> 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	<a href="#">M3-9-3</a>	-
4-5-1 (Required) Expressions with type <code>bool</code> shall not be used as operands to built-in operators other than the assignment operator <code>=</code> , the logical operators <code>&amp;&amp;</code> , <code>  </code> , <code>!</code> , the equality operators <code>==</code> and <code>!=</code> , the unary <code>&amp;</code> operator, and the conditional operator.	1 - Identical	<a href="#">M4-5-1</a>	-
4-5-2 (Required) Expressions with type <code>enum</code> shall not be used as operands to built-in operators other than the subscript operator <code>[ ]</code> , the assignment operator <code>=</code> , the equality operators <code>==</code> and <code>!=</code> , the unary <code>&amp;</code> operator, and the relational operators <code>&lt;</code> , <code>&lt;=</code> , <code>&gt;</code> , <code>&gt;=</code> .	3 - Significant differences	<a href="#">A4-5-1</a>	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	<a href="#">M4-5-3</a>	-
4-10-1 (Required) NULL shall not be used as an integer value.	1 - Identical	<a href="#">M4-10-1</a>	-
4-10-2 (Required) Literal zero (0) shall not be used as the null-pointer-constant.	1 - Identical	<a href="#">M4-10-2</a>	-
5-0-1 (Required) The value of an expression shall be the same under any order of evaluation that the standard permits.	1 - Identical	<a href="#">A5-0-1</a>	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	<a href="#">M5-0-2</a>	-
5-0-3 (Required) A cvalue expression shall not be implicitly converted to a different underlying type.	1 - Identical	<a href="#">M5-0-3</a>	-
5-0-4 (Required) An implicit integral conversion shall not change the signedness of the underlying type.	1 - Identical	<a href="#">M5-0-4</a>	-
5-0-5 (Required) There shall be no implicit floating-integral conversions.	1 - Identical	<a href="#">M5-0-5</a>	-
5-0-6 (Required) An implicit integral or floating-point conversion shall not reduce the size of the underlying type.	1 - Identical	<a href="#">M5-0-6</a>	-
5-0-7 (Required) There shall be no explicit floating-integral conversions of a cvalue expression.	1 - Identical	<a href="#">M5-0-7</a>	-
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	<a href="#">M5-0-8</a>	-
5-0-9 (Required) An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.	1 - Identical	<a href="#">M5-0-9</a>	-
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	<a href="#">M5-0-10</a>	-
5-0-11 (Required) The plain char type shall only be used for the storage and use of character values.	1 - Identical	<a href="#">M5-0-11</a>	-
5-0-12 (Required) signed char and unsigned char type shall only be used for the storage and use of numeric values.	1 - Identical	<a href="#">M5-0-12</a>	-

5-0-13 (Required) The condition of an if-statement and the condition of an iteration statement shall have type bool.	2 - Small differences	<a href="#">A5-0-2</a>	Example extended.
5-0-14 (Required) The first operand of a conditional-operator shall have type bool.	1 - Identical	<a href="#">M5-0-14</a>	-
5-0-15 (Required) Array indexing shall be the only form of pointer arithmetic.	1 - Identical	<a href="#">M5-0-15</a>	-
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	<a href="#">M5-0-16</a>	-
5-0-17 (Required) Subtraction between pointers shall only be applied to pointers that address elements of the same array.	1 - Identical	<a href="#">M5-0-17</a>	-
5-0-18 (Required) >, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.	1 - Identical	<a href="#">M5-0-18</a>	-
5-0-19 (Required) The declaration of objects shall contain no more than two levels of pointer indirection.	2 - Small differences	<a href="#">A5-0-3</a>	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	<a href="#">M5-0-20</a>	-
5-0-21 (Required) Bitwise operators shall only be applied to operands of unsigned underlying type.	1 - Identical	<a href="#">M5-0-21</a>	-
5-2-1 (Required) Each operand of a logical && or    shall be a postfix expression.	1 - Identical	<a href="#">M5-2-1</a>	-
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	<a href="#">M5-2-2</a>	-
5-2-3 (Advisory) Casts from a base class to a derived class should not be performed on polymorphic types.	1 - Identical	<a href="#">M5-2-3</a>	-
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	<a href="#">A5-2-2</a>	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	<a href="#">A5-2-3</a>	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	<a href="#">M5-2-6</a>	-
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	<a href="#">A5-2-4</a>	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	<a href="#">M5-2-8</a>	-
5-2-9 (Advisory) A cast shall not convert a pointer type to an integral type.	2 - Small differences	<a href="#">M5-2-9</a>	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	<a href="#">M5-2-10</a>	Obligation level changed to "Required".
5-2-11 (Required) The comma operator, && operator and the operator shall not be overloaded.	1 - Identical	<a href="#">M5-2-11</a>	-
5-2-12 (Required) An identifier with array type passed as a function argument shall not decay to a pointer.	1 - Identical	<a href="#">M5-2-12</a>	-
5-3-1 (Required) Each operand of the ! operator, the logical && or the logical    operators shall have type bool.	1 - Identical	<a href="#">M5-3-1</a>	-
5-3-2 (Required) The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	1 - Identical	<a href="#">M5-3-2</a>	-
5-3-3 (Required) The unary & operator shall not be overloaded.	1 - Identical	<a href="#">M5-3-3</a>	-
5-3-4 (Required) Evaluation of the operand to the sizeof operator shall not contain side effects.	1 - Identical	<a href="#">M5-3-4</a>	-
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	<a href="#">M5-8-1</a>	-
5-14-1 (Required) The right hand operand of a logical && or    operator shall not contain side effects.	1 - Identical	<a href="#">M5-14-1</a>	-

5-17-1 (Required) The semantic equivalence between a binary operator and its assignment operator form shall be preserved.	1 - Identical	<a href="#">M5-17-1</a>	-
5-18-1 (Required) The comma operator shall not be used.	1 - Identical	<a href="#">M5-18-1</a>	-
5-19-1 (Required) Evaluation of constant unsigned integer expressions shall not lead to wrap-around.	2 - Small differences	<a href="#">M5-19-1</a>	Obligation level changed to "Required".
6-2-1 (Required) Assignment operators shall not be used in subexpressions.	1 - Identical	<a href="#">M6-2-1</a>	-
6-2-2 (Required) Floating-point expressions shall not be directly or indirectly tested for equality or inequality.	1 - Identical	<a href="#">M6-2-2</a>	-
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	<a href="#">M6-2-3</a>	-
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	<a href="#">M6-3-1</a>	-
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	<a href="#">M6-4-1</a>	-
6-4-2 (Required) All if ... else if constructs shall be terminated with an else clause.	1 - Identical	<a href="#">M6-4-2</a>	-
6-4-3 (Required) A switch statement shall be a well-formed switch statement.	1 - Identical	<a href="#">M6-4-3</a>	-
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	<a href="#">M6-4-4</a>	-
6-4-5 (Required) An unconditional throw or break statement shall terminate every non-empty switch clause.	1 - Identical	<a href="#">M6-4-5</a>	-
6-4-6 (Required) The final clause of a switch statement shall be the default-clause.	1 - Identical	<a href="#">M6-4-6</a>	-

6-4-7 (Required) The condition of a switch statement shall not have bool type.	1 - Identical	<a href="#">M6-4-7</a>	-
6-4-8 (Required) Every switch statement shall have at least one case-clause.	3 - Significant differences	<a href="#">A6-4-1</a>	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	<a href="#">A6-5-2</a>	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	<a href="#">M6-5-2</a>	-
6-5-3 (Required) The loop-counter shall not be modified within condition or statement.	1 - Identical	<a href="#">M6-5-3</a>	-
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	<a href="#">M6-5-4</a>	-
6-5-5 (Required) A loop-control-variable other than the loop-counter shall not be modified within condition or expression.	1 - Identical	<a href="#">M6-5-5</a>	-
6-5-6 (Required) A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.	1 - Identical	<a href="#">M6-5-6</a>	-
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	<a href="#">M6-6-1</a>	-
6-6-2 (Required) The goto statement shall jump to a label declared later in the same function body.	1 - Identical	<a href="#">M6-6-2</a>	-
6-6-3 (Required) The continue statement shall only be used within a well-formed for loop.	1 - Identical	<a href="#">M6-6-3</a>	-
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: <a href="#">A6-6-1</a> . 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		Single point of exit approach does not necessarily improve readability, maintainability and testability. A function can have multiple points of exit.
7-1-1 (Required) A variable which is not modified shall be const qualified.	4 - Rejected	-	Rule replaced with <a href="#">A7-1-1</a> , <a href="#">A7-1-2</a> that concern constexpr and const specifiers.
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	<a href="#">M7-1-2</a>	-
7-2-1 (Required) An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.	2 - Small differences	<a href="#">A7-1-2</a>	Example extended.
7-3-1 (Required) The global namespace shall only contain main, namespace declarations and extern "C" declarations.	1 - Identical	<a href="#">M7-3-1</a>	-
7-3-2 (Required) The identifier main shall not be used for a function other than the global function main.	1 - Identical	<a href="#">M7-3-2</a>	-
7-3-3 (Required) There shall be no unnamed namespaces in header files.	1 - Identical	<a href="#">M7-3-3</a>	-
7-3-4 (Required) Using-directives shall not be used.	1 - Identical	<a href="#">M7-3-4</a>	-
7-3-5 (Required) Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier.	1 - Identical	<a href="#">M7-3-5</a>	-
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	<a href="#">M7-3-6</a>	-
7-4-1 (Document) All usage of assembler shall be documented.	1 - Identical	<a href="#">M7-4-1</a>	-
7-4-2 (Required) Assembler instructions shall only be introduced using the asm declaration.	1 - Identical	<a href="#">M7-4-2</a>	-
7-4-3 (Required) Assembly language shall be encapsulated and isolated.	1 - Identical	<a href="#">M7-4-3</a>	-



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	<a href="#">M7-5-1</a>	-
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	<a href="#">M7-5-2</a>	Added a note saying that the rule applies to <code>std::unique_ptr</code> , <code>std::shared_ptr</code> and <code>std::weak_ptr</code> 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	<a href="#">A7-5-2</a>	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	<a href="#">A7-5-1</a>	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	<a href="#">M8-0-1</a>	-
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	<a href="#">M8-3-1</a>	-
8-4-1 (Required) Functions shall not be defined using the ellipsis notation.	3 - Significant differences	<a href="#">A8-4-1</a>	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	<a href="#">M8-4-2</a>	-

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	<a href="#">A8-4-2</a>	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	<a href="#">M8-4-4</a>	-
8-5-1 (Required) All variables shall have a defined value before they are used.	1 - Identical	<a href="#">M8-5-1</a>	-
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	<a href="#">M8-5-2</a>	-
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	<a href="#">A7-2-4</a>	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	<a href="#">M9-3-1</a>	-
9-3-2 (Required) Member functions shall not return non-const handles to class-data.	2 - Small differences	<a href="#">A9-3-1</a>	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	<a href="#">M9-3-3</a>	-
9-5-1 (Required) Unions shall not be used.	1 - Identical	<a href="#">M9-5-1</a>	-
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	<a href="#">M9-6-1</a>	-
9-6-2 (Required) Bit-fields shall be either bool type or an explicitly unsigned or signed integral type.	4 - Rejected	-	Permitted types changed. New rule introduced: <a href="#">A9-6-1</a> .
9-6-3 (Required) Bit-fields shall not have enum type.	4 - Rejected	-	Permitted types changed. New rule introduced: <a href="#">A9-6-1</a> .

9-6-4 (Required) Named bit-fields with signed integer type shall have a length of more than one bit.	4 - Rejected	-	Permitted types changed. New rule introduced: <a href="#">A9-6-1</a> .
10-1-1 (Advisory) Classes should not be derived from virtual bases	1 - Identical	<a href="#">M10-1-1</a>	-
10-1-2 (Required) A base class shall only be declared virtual if it is used in a diamond hierarchy.	4 - Rejected	-	Multiple inheritance is not allowed.
10-1-3 (Required) An accessible base class shall not be both virtual and non-virtual in the same hierarchy.	1 - Identical	<a href="#">M10-1-3</a>	-
10-2-1 (Advisory) All accessible entity names within a multiple inheritance hierarchy should be unique.	1 - Identical	<a href="#">M10-2-1</a>	-
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 <a href="#">A10-1-1</a> and <a href="#">A10-3-1</a>
10-3-2 (Required) Each overriding virtual function shall be declared with the virtual keyword.	3 - Significant differences	<a href="#">A10-3-2</a>	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	<a href="#">M10-3-3</a>	-
11-0-1 (Required) Member data in non-POD class types shall be private.	1 - Identical	<a href="#">M11-0-1</a>	-
12-1-1 (Required) An object's dynamic type shall not be used from the body of its constructor or destructor.	1 - Identical	<a href="#">M12-1-1</a>	-
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	<a href="#">A12-1-1</a>	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	<a href="#">A12-1-4</a>	Example reworked.

<p>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.</p>	<p>3 - Significant differences</p>	<p><a href="#">A12-8-1</a></p>	<p>Rule reformulated to cover move constructors, too. Rationale reformulated. Example reworked.</p>
<p>12-8-2 (Required) The copy assignment operator shall be declared protected or private in an abstract class.</p>	<p>3 - Significant differences</p>	<p><a href="#">A12-8-6</a></p>	<p>Rule reformulated to cover move assignment operators and all base classes. Rationale reformulated. Example reworked.</p>
<p>12-8-2 (Required) The copy assignment operator shall be declared protected or private in an abstract class.</p>	<p>3 - Significant differences</p>	<p><a href="#">A12-8-1</a></p>	<p>Rule reformulated to cover move constructors, too. Rationale reformulated. Example reworked.</p>
<p>14-5-1 (Required) A non-member generic function shall only be declared in a namespace that is not an associated namespace.</p>	<p>4 - Rejected</p>	<p>-</p>	<p>Usage of the ADL functionality is allowed. It is also used in STL for overloaded operators lookup in e.g. out streams, STL containers.</p>
<p>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.</p>	<p>1 - Identical</p>	<p><a href="#">M14-5-2</a></p>	<p>-</p>
<p>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.</p>	<p>1 - Identical</p>	<p><a href="#">M14-5-3</a></p>	<p>-</p>
<p>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-&gt;.</p>	<p>1 - Identical</p>	<p><a href="#">M14-6-1</a></p>	<p>-</p>

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.	1 - Identical	M14-7-3	-
14-8-1 (Required) Overloaded function templates shall not be explicitly specialized.	1 - Identical	M14-8-1	-
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	A14-8-1	Rule slightly reformulated. Example significantly reworked.
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	<a href="#">M15-3-1</a>	-
15-3-2 (Advisory) There should be at least one exception handler to catch all otherwise unhandled exceptions.	2 - Small differences	<a href="#">A15-3-3</a>	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	<a href="#">M15-3-3</a>	-
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	<a href="#">M15-3-4</a>	-
15-3-5 (Required) A class type exception shall always be caught by reference.	2 - Small differences	<a href="#">A15-3-5</a>	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	<a href="#">M15-3-6</a>	-
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	<a href="#">M15-3-7</a>	-
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.	4 - Rejected	-	Dynamic exception specification was prohibited. Noexcept specifier shall be used instead.
15-5-1 (Required) A class destructor shall not exit with an exception.	3 - Significant differences	<a href="#">A15-5-1</a>	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).	4-Rejected	-	Dynamic exception specification was prohibited.
15-5-3 (Required) The std::terminate() function shall not be called implicitly.	2 - Small differences	<a href="#">A15-5-3</a>	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	<a href="#">A16-2-1</a>	Merged with MISRA Rule 16-2-5.
16-2-5 (Advisory) The character \ should not occur in a header file name.	2 - Small differences	<a href="#">A16-2-1</a>	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	<a href="#">M16-3-1</a>	-
16-3-2 (Advisory) The # and ## operators should not be used.	1 - Identical	<a href="#">M16-3-2</a>	-
16-6-1 (Required) All uses of the #pragma directive shall be documented.	4 - Rejected	-	The #pragma directive shall not be used, see: <a href="#">A16-7-1</a> .
17-0-1 (Required) Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.	2 - Small differences	<a href="#">A17-0-1</a>	Example extended.
17-0-2 (Required) The names of standard library macros and objects shall not be reused.	1 - Identical	<a href="#">M17-0-2</a>	-
17-0-3 (Required) The names of standard library functions shall not be overridden.	1 - Identical	<a href="#">M17-0-3</a>	-
17-0-4 (Required) All library code shall conform to MISRA C++.	4 - Rejected	-	The rule replaced with <a href="#">A17-0-2</a> 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	<a href="#">M17-0-5</a>	-
18-0-1 (Required) The C library shall not be used.	2 - Small differences	<a href="#">A18-0-1</a>	Rule reformulated.
18-0-2 (Required) The library functions atof, atoi and atol from library <cstdlib> shall not be used.	2 - Small differences	<a href="#">A18-0-2</a>	Compliant alternatives added into rationale.



18-0-3 (Required) The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.	1 - Identical	<a href="#">M18-0-3</a>	-
18-0-4 (Required) The time handling functions of library <ctime> shall not be used.	1 - Identical	<a href="#">M18-0-4</a>	-
18-0-5 (Required) The unbounded functions of library <cstring> shall not be used.	1 - Identical	<a href="#">M18-0-5</a>	-
18-2-1 (Required) The macro offsetof shall not be used.	1 - Identical	<a href="#">M18-2-1</a>	-
18-4-1 (Required) Dynamic heap memory allocation shall not be used.	4 - Rejected	-	Dynamic heap memory allocation usage is allowed conditionally, see: <a href="#">A18-5-1</a> , <a href="#">A18-5-2</a> , <a href="#">A18-5-3</a> .
18-7-1 (Required) The signal handling facilities of <csignal> shall not be used.	1 - Identical	<a href="#">M18-7-1</a>	-
19-3-1 (Required) The error indicator errno shall not be used.	1 - Identical	<a href="#">M19-3-1</a>	-
27-0-1 (Required) The stream input/output library <cstdio> shall not be used.	1 - Identical	<a href="#">M27-0-1</a>	-

**Table A.1: MISRA C++**

## A.2 Traceability to HIC++ v4.0

The following table demonstrates the traceability to High Integrity C++ Coding Standard Version 4.0 [8]. 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	<a href="#">A1-1-1</a>	Specified that the code shall conform to ISO/IEC 14882:2014
1.2.1 Ensure that all statements are reachable.	2 - Small differences	<a href="#">M0-1-1</a>	
1.2.2 Ensure that no expression or sub-expression is redundant.	2 - Small differences	<a href="#">M0-1-9</a>	
1.3.1 Do not use the increment operator (++) on a variable of type bool	2 - Small differences	<a href="#">M4-5-1</a>	
1.3.2 Do not use the register keyword.	2 - Small differences	<a href="#">A7-1-4</a>	

1.3.3 Do not use the C Standard Library .h headers	2 - Small differences	<a href="#">A18-0-1</a>	
1.3.4 Do not use deprecated STL library features	2 - Small differences	<a href="#">A1-1-1</a> , <a href="#">A18-1-3</a> , <a href="#">A18-9-1</a>	
1.3.5 Do not use throw exception specifications.	2 - Small differences	<a href="#">A15-4-1</a>	
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	<a href="#">A2-5-1</a> , <a href="#">A2-6-1</a>	
2.3.1 Do not use the C comment delimiters /* ... */.	2 - Small differences	<a href="#">A2-8-4</a>	
2.3.2 Do not comment out code.	2 - Small differences	<a href="#">A2-8-2</a>	
2.4.1 Ensure that each identifier is distinct from any other visible identifier.	2 - Small differences	<a href="#">M2-10-1</a>	
2.5.1 Do not concatenate strings with different encoding prefixes.	2 - Small differences	<a href="#">A2-14-2</a>	
2.5.2 Do not use octal constants (other than zero).	2 - Small differences	<a href="#">M2-13-2</a>	
2.5.3 Use nullptr for the null pointer constant.	2 - Small differences	<a href="#">A4-10-1</a>	
3.1.1 Do not hide declarations.	2 - Small differences	<a href="#">A2-11-1</a>	
3.2.1 Do not declare functions at block scope.	2 - Small differences	<a href="#">M3-1-2</a>	
3.3.1 Do not use variables with static storage duration.	3 - Significant differences	<a href="#">A3-3-2</a>	Limited to non-POD type objects only.
3.4.1 Do not return a reference or a pointer to an automatic variable defined within the function.	2 - Small differences	<a href="#">M7-5-1</a>	
3.4.2 Do not assign the address of a variable to a pointer with a greater lifetime.	2 - Small differences	<a href="#">M7-5-2</a>	
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: <a href="#">A15-1-4</a> .
3.5.1 Do not make any assumptions about the internal representation of a value or object.	2 - Small differences	<a href="#">A3-9-1</a> , <a href="#">M3-9-3</a> , <a href="#">M5-0-15</a> , <a href="#">M5-0-21</a> , <a href="#">M9-5-1</a> , <a href="#">M18-2-1</a>	

4.1.1 Ensure that a function argument does not undergo an array-to-pointer conversion.	2 - Small differences	<a href="#">M5-2-12</a>	
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	<a href="#">M2-13-2</a>	
4.2.2 Ensure that data loss does not demonstrably occur in an integral expression.	2 - Small differences	<a href="#">A4-7-1</a> , <a href="#">M5-0-4</a> , <a href="#">M5-0-6</a> , <a href="#">M5-0-9</a>	
4.3.1 Do not convert an expression of wider floating point type to a narrower floating point type.	2 - Small differences	<a href="#">A4-7-1</a> , <a href="#">M5-0-6</a>	
4.4.1 Do not convert floating values to integral types except through use of standard library functions.	4 - Rejected		Rules that are related: <a href="#">M5-0-3</a> , <a href="#">M5-0-5</a> , <a href="#">M5-0-6</a> , <a href="#">M5-0-7</a> ,
5.1.1 Use symbolic names instead of literal values in code.	2 - Small differences	<a href="#">A5-1-1</a>	
5.1.2 Do not rely on the sequence of evaluation within an expression.	2 - Small differences	<a href="#">A5-0-1</a>	
5.1.3 Use parentheses in expressions to specify the intent of the expression.	2 - Small differences	<a href="#">A5-0-1</a> , <a href="#">M5-2-1</a> , <a href="#">M5-2-10</a> ,	
5.1.4 Do not capture variables implicitly in a lambda.	2 - Small differences	<a href="#">A5-1-2</a>	
5.1.5 Include a (possibly empty) parameter list in every lambda expression.	2 - Small differences	<a href="#">A5-1-3</a>	
5.1.6 Do not code side effects into the right-hand operands of: <code>&amp;&amp;</code> , <code>  </code> , <code>sizeof</code> , <code>typeid</code> or a function passed to <code>condition_variable::wait</code> .	3 - Significant differences	<a href="#">A5-3-1</a> , <a href="#">M5-3-4</a> , <a href="#">M5-14-1</a>	The <code>condition_variable::wait</code> 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	<a href="#">A5-2-5</a>	
5.2.2 Ensure that functions do not call themselves, either directly or indirectly.	2 - Small differences	<a href="#">A7-5-2</a>	
5.3.1 Do not apply unary minus to operands of unsigned type.	2 - Small differences	<a href="#">M5-3-2</a>	
5.3.2 Allocate memory using <code>new</code> and release it using <code>delete</code> .	2 - Small differences	<a href="#">A18-5-1</a>	Note that operators <code>new</code> and <code>delete</code> shall not be used explicitly, see: <a href="#">A18-5-2</a> .

5.3.3 Ensure that the form of delete matches the form of new used to allocate the memory.	2 - Small differences	<a href="#">A18-5-3</a>	Note that operators new and delete shall not be used explicitly, see: <a href="#">A18-5-2</a> .
5.4.1 Only use casting forms: static_cast (excl. void*), dynamic_cast or explicit constructor call.	2 - Small differences	<a href="#">A5-2-1</a> , <a href="#">A5-2-2</a> , <a href="#">A5-2-3</a> , <a href="#">A5-2-4</a>	
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: <a href="#">A7-2-1</a> .
5.4.3 Do not convert from a base class to a derived class.	3 - Small differences	<a href="#">M5-2-2</a> , <a href="#">M5-2-3</a> , <a href="#">A5-2-1</a>	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	<a href="#">A5-5-1</a>	
5.6.1 Do not use bitwise operators with signed operands.	2 - Small differences	<a href="#">M5-0-21</a>	
5.7.1 Do not write code that expects floating point calculations to yield exact results.	2 - Small differences	<a href="#">M6-2-2</a>	
5.7.2 Ensure that a pointer to member that is a virtual function is only compared (==) with nullptr.	2 - Small differences	<a href="#">A5-10-1</a>	
5.8.1 Do not use the conditional operator (?:) as a sub-expression.	2 - Small differences	<a href="#">A5-16-1</a>	
6.1.1 Enclose the body of a selection or an iteration statement in a compound statement.	2 - Small differences	<a href="#">M6-3-1</a> , <a href="#">M6-4-1</a>	
6.1.2 Explicitly cover all paths through multi-way selection statements.	2 - Small differences	<a href="#">M6-4-2</a>	
6.1.3 Ensure that a non-empty case statement block does not fall through to the next label.	2 - Small differences	<a href="#">M6-4-5</a>	
6.1.4 Ensure that a switch statement has at least two case labels, distinct from the default label.	2 - Small differences	<a href="#">A6-4-1</a>	
6.2.1 Implement a loop that only uses element values as a range-based loop.	2 - Small differences	<a href="#">A6-5-1</a>	
6.2.2 Ensure that a loop has a single loop counter, an optional control variable, and is not degenerate.	2 - Small differences	<a href="#">A6-5-2</a>	

6.2.3 Do not alter a control or counter variable more than once in a loop.	3 - Significant differences	<a href="#">M6-5-3</a>	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	<a href="#">M6-5-3</a>	
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	<a href="#">M6-6-1</a>	
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	<a href="#">A8-4-2</a>	
6.4.1 Postpone variable definitions as long as possible.	2 - Small differences	<a href="#">M3-4-1</a>	
7.1.1 Declare each identifier on a separate line in a separate declaration.	2 - Small differences	<a href="#">A7-1-7</a>	
7.1.2 Use const whenever possible.	2 - Small differences	<a href="#">A7-1-1</a> , <a href="#">A7-1-2</a>	
7.1.3 Do not place type specifiers before non-type specifiers in a declaration.	2 - Small differences	<a href="#">A7-1-8</a>	
7.1.4 Place CV-qualifiers on the right hand side of the type they apply to.	2 - Small differences	<a href="#">A7-1-3</a>	
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	<a href="#">A3-9-1</a>	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	<a href="#">A8-2-1</a>	
7.1.8 Use auto id = expr when declaring a variable to have the same type as its initializer function call.	3 - Significant differences	<a href="#">A7-1-5</a>	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: <a href="#">A5-1-6</a> .

7.1.10 Use <code>static_assert</code> for assertions involving compile time constants.	3 - Significant differences	<a href="#">A16-6-1</a>	It is recommended to use the <code>static_assert</code> instead of <code>#error</code> directive.
7.2.1 Use an explicit enumeration base and ensure that it is large enough to store all enumerators.	2 - Small differences	<a href="#">A7-2-2</a>	
7.2.2 Initialize none, the first only or all enumerators in an enumeration.	2 - Small differences	<a href="#">A7-2-4</a>	
7.3.1 Do not use using directives.	2 - Small differences	<a href="#">M7-3-4</a>	
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	<a href="#">A3-1-1</a> , <a href="#">M3-2-2</a>	
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	<a href="#">A3-1-1</a> , <a href="#">M3-2-4</a>	
7.5.1 Do not use the <code>asm</code> declaration.	2 - Small differences	<a href="#">A7-4-1</a>	
8.1.1 Do not use multiple levels of pointer indirection.	3 - Significant differences	<a href="#">A5-0-3</a>	At most two levels of pointer indirection are allowed.
8.2.1 Make parameter names absent or identical in all declarations.	2 - Small differences	<a href="#">M3-9-1</a>	
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 <code>std::unique_ptr</code> by const reference.	2 - Small differences	<a href="#">A18-1-5</a>	
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. <a href="#">M8-3-1</a> .
8.3.4 Define =delete functions with parameters of type rvalue reference to const.	2 - Small differences	<a href="#">A13-3-1</a> , <a href="#">A18-9-3</a>	
8.4.1 Do not access an invalid object or an object with indeterminate value.	2 - Small differences	<a href="#">M8-5-1</a> , <a href="#">A12-8-3</a>	
8.4.2 Ensure that a braced aggregate initializer matches the layout of the aggregate object.	2 - Small differences	<a href="#">M8-5-2</a>	
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	<a href="#">M9-3-3</a>	
9.1.2 Make default arguments the same or absent when overriding a virtual function.	2 - Small differences	<a href="#">M8-3-1</a>	
9.1.3 Do not return non-const handles to class data from const member functions.	2 - Small differences	<a href="#">M9-3-1</a> , <a href="#">A9-3-1</a>	
9.1.4 Do not write member functions which return non-const handles to data less accessible than the member function.	3 - Significant differences	<a href="#">A9-3-1</a>	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	<a href="#">A10-3-3</a>	
9.2.1 Declare bit-fields with an explicitly unsigned integral or enumeration type.	2 - Small differences	<a href="#">A9-6-1</a>	
10.1.1 Ensure that access to base class subobjects does not require explicit disambiguation.	3 - Significant differences	<a href="#">A10-1-1</a>	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	<a href="#">A10-3-2</a>	
10.3.1 Ensure that a derived class has at most one base class which is not an interface class.	2 - Small differences	<a href="#">A10-1-1</a>	Note that the definition of an interface changed, see: <a href="#">Interface-Class</a> .
11.1.1 Declare all data members private.	2 - Small differences	<a href="#">M11-0-1</a>	
11.2.1 Do not use friend declarations.	2 - Small differences	<a href="#">A11-3-1</a>	

12.1.1 Do not declare implicit user defined conversions.	3 - Significant differences	<a href="#">A12-1-4</a>	
12.2.1 Declare virtual, private or protected the destructor of a type used as a base class.	3 - Significant differences	<a href="#">A12-4-1</a>	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.	4 - Rejected		There is no need for a new rule.
12.4.1 Do not use the dynamic type of an object unless the object is fully constructed.	2 - Small differences	<a href="#">M12-1-1</a>	
12.4.2 Ensure that a constructor initializes explicitly all base classes and non-static data members.	2 - Small differences	<a href="#">A12-1-1</a>	
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	<a href="#">A12-1-2</a>	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	<a href="#">A8-5-1</a>	
12.4.5 Use delegating constructors to reduce code duplication.	4 - Rejected		Non-generic design principle; There is no need for a new rule.
12.5.1 Define explicitly =default or =delete implicit special member functions of concrete classes.	3 - Significant differences	<a href="#">A12-0-1</a>	
12.5.2 Define special members =default if the behavior is equivalent.	2 - Small differences	<a href="#">A12-7-1</a>	
12.5.3 Ensure that a user defined move/copy constructor only moves/copies base and member objects.	2 - Small differences	<a href="#">A12-8-1</a>	
12.5.4 Declare noexcept the move constructor and move assignment operator.	3 - Significant differences	<a href="#">A15-5-1</a>	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	<a href="#">A12-8-1</a>	



12.5.6 Use an atomic, non-throwing swap operation to implement the copy and move assignment operators.	2 - Small differences	<a href="#">A12-8-2</a>	
12.5.7 Declare assignment operators with the ref-qualifier &.	2 - Small differences	<a href="#">A12-8-7</a>	
12.5.8 Make the copy assignment operator of an abstract class protected or define it =delete.	3 - Significant differences	<a href="#">A12-8-6</a>	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	<a href="#">M7-3-5</a>	
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	<a href="#">A13-3-1</a>	A function taking “forwarding reference” shall not be overloaded.
13.2.1 Do not overload operators with special semantics.	2 - Small differences	<a href="#">M5-2-11</a> , <a href="#">M5-3-3</a>	
13.2.2 Ensure that the return type of an overloaded binary operator matches the built-in counterparts.	2 - Small differences	<a href="#">A13-2-1</a> , <a href="#">A13-2-2</a> , <a href="#">A13-2-3</a>	
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	<a href="#">A13-5-1</a>	
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	<a href="#">A8-4-1</a>	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.	2 - Small differences	<a href="#">M14-7-3</a>	
14.2.2 Do not explicitly specialize a function template that is overloaded with other templates.	2 - Small differences	<a href="#">M14-8-1</a>	
14.2.3 Declare extern an explicitly instantiated template.	4 - Rejected		

15.1.1 Only use instances of <code>std::exception</code> for exceptions.	2 - Small differences	<a href="#">A15-1-1</a>	
15.2.1 Do not throw an exception from a destructor.	2 - Small differences	<a href="#">A15-5-1</a>	
15.3.1 Do not access non-static members from a catch handler of constructor/destructor function try block.	2 - Small differences	<a href="#">M15-3-3</a>	
15.3.2 Ensure that a program does not result in a call to <code>std::terminate</code> .	2 - Small differences	<a href="#">A15-5-2</a> , <a href="#">A15-5-3</a>	
16.1.1 Use the preprocessor only for implementing include guards, and including header files with include guards.	3 - Significant differences	<a href="#">A16-0-1</a>	Conditional and unconditional file inclusion is allowed.
16.1.2 Do not include a path specifier in filenames supplied in <code>#include</code> directives.	3 - Significant differences	<a href="#">A16-2-1</a>	Path specifier <code>/is</code> allowed to specify a path relative to path passed to the compiler.
16.1.3 Match the filename in a <code>#include</code> directive to the one on the file system.	4 - Rejected		
16.1.4 Use <code>&lt;&gt;</code> 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: <a href="#">A16-2-2</a> , <a href="#">A16-2-3</a> .
17.1.1 Do not use <code>std::vector&lt;bool&gt;</code> .	2 - Small differences	<a href="#">A18-1-2</a>	
17.2.1 Wrap use of the C Standard Library.	2 - Small differences	<a href="#">A17-1-1</a>	
17.3.1 Do not use <code>std::move</code> on objects declared with <code>const</code> or <code>const &amp;</code> type.	2 - Small differences	<a href="#">A18-9-3</a>	
17.3.2 Use <code>std::forward</code> to forward universal references.	2 - Small differences	<a href="#">A18-9-2</a>	
17.3.3 Do not subsequently use the argument to <code>std::forward</code> .	2 - Small differences	<a href="#">A18-9-4</a>	

<p>17.3.4 Do not create smart pointers of array type.</p>	<p>3 - Significant differences</p>	<p><a href="#">A18-1-4</a></p>	<p>This especially concerns <code>std::shared_ptr</code>, because <code>std::unique_ptr</code> provides partial specialization for array types.</p>
<p>17.3.5 Do not create an rvalue reference of <code>std::array</code>.</p>	<p>4 - Rejected</p>		<p>The rule is only a hint saying that passing <code>std::array</code> 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 <code>std::array</code> by rvalue reference.</p>
<p>17.4.1 Use <code>const</code> container calls when result is immediately converted to a <code>const</code> iterator.</p>	<p>2 - Small differences</p>	<p><a href="#">A23-0-1</a></p>	
<p>17.4.2 Use API calls that construct objects in place.</p>	<p>3 - Significant differences</p>	<p><a href="#">A18-5-2</a></p>	<p><a href="#">A18-5-2</a> prohibits explicit calls to <code>new</code> and <code>delete</code> operators, <code>std::make_shared</code>, <code>std::make_unique</code> and similar constructions are recommended.</p>
<p>17.5.1 Do not ignore the result of <code>std::remove</code>, <code>std::remove_if</code> or <code>std::unique</code>.</p>	<p>2 - Small differences</p>	<p><a href="#">A0-1-2</a></p>	
<p>18.1.1 Do not use platform specific multi-threading facilities.</p>	<p>5 - Not yet analyzed</p>		<p>The “Concurrency” chapter is not yet covered, this will be addressed in future.</p>
<p>18.2.1 Use <code>high_integrity::thread</code> in place of <code>std::thread</code>.</p>	<p>4 - Rejected</p>		<p>The <code>high_integrity::thread</code> is not part of the C++ Language Standard.</p>

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.	5 - Not yet analyzed		The “Concurrency” chapter is not yet covered, this will be addressed in future.
18.2.4 Use <code>std::call_once</code> 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 <code>std::recursive_mutex</code> .	5 - Not yet analyzed		The “Concurrency” chapter is not yet covered, this will be addressed in future.
18.3.4 Only use <code>std::unique_lock</code> when <code>std::lock_guard</code> 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 <code>std::mutex</code> 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 <code>std::condition_variable_any</code> on a <code>std::mutex</code>	5 - Not yet analyzed		The “Concurrency” chapter is not yet covered, this will be addressed in future.
---	----------------------	--	---

**Table A.2: HIC++ v4.0**

### A.3 Traceability to JSF

The following table demonstrates the traceability to Joint Strike Fighter Air Vehicle C++ Coding Standard [7]. 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	<a href="#">A1-1-1</a>	
AV Rule 9 Only those characters specified in the C++ basic source character set will be used. [...].	2 - Small differences	<a href="#">A2-2-1</a>	
AV Rule 10 Values of character types will be restricted to a defined and documented subset of ISO 10646-1.	5 - Not yet analyzed		
AV Rule 11 Trigraphs will not be used.	2 - Small differences	<a href="#">A2-5-1</a>	
AV Rule 12 The following digraphs will not be used [...].	2 - Small differences	<a href="#">A2-6-1</a>	
AV Rule 13 Multi-byte characters and wide string literals will not be used.	4 - Rejected		Agreed for <code>wchar_t</code> type only, <a href="#">A2-14-3</a> .
AV Rule 14 Literal suffixes shall use uppercase rather than lowercase letters.	2 - Small differences	<a href="#">M2-13-4</a>	
AV Rule 15 Provision shall be made for run-time checking (defensive programming).	2 - Small differences	<a href="#">M0-3-1</a>	
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 <code>errno</code> shall not be used.	2 - Small differences	<a href="#">M19-3-1</a>	
AV Rule 18 The macro <code>offsetof</code> , in library <code>&lt;stddef.h&gt;</code> , shall not be used.	2 - Small differences	<a href="#">M18-2-1</a>	
AV Rule 19 <code>&lt;locale.h&gt;</code> and the <code>setlocale</code> function shall not be used.	2 - Small differences	<a href="#">A18-0-3</a>	
AV Rule 20 The <code>setjmp</code> macro and the <code>longjmp</code> function shall not be used.	2 - Small differences	<a href="#">M17-0-5</a>	

AV Rule 21 The signal handling facilities of <signal.h> shall not be used.	2 - Small differences	<a href="#">M18-7-1</a>	
AV Rule 22 The input /output library <stdio.h> shall not be used.	2 - Small differences	<a href="#">M27-0-1</a>	
AV Rule 23 The library functions atof, atoi and atol from library <stdlib.h> shall not be used.	2 - Small differences	<a href="#">A18-0-2</a>	
AV Rule 24 The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used.	2 - Small differences	<a href="#">M18-0-3</a>	
AV Rule 25 The time handling functions of library <time.h> shall not be used.	2 - Small differences	<a href="#">M18-0-4</a>	
AV Rule 26 Only the following pre-processor directives shall be used: 1. #ifndef 2. #define 3. #endif 4. #include.	2 - Small differences	<a href="#">A16-0-1</a>	
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	<a href="#">A16-0-1</a> , <a href="#">M16-2-3</a>	
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	<a href="#">A16-0-1</a>	
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	<a href="#">A16-0-1</a>	
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	<a href="#">A16-0-1</a>	
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	<a href="#">A16-0-1</a>	
AV Rule 32 The #include pre-processor directive will only be used to include header (*.h) files.	2 - Small differences	<a href="#">A16-0-1</a>	
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	<a href="#">A3-3-1</a>	

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.	5 - Not yet analyzed		
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.	5 - Not yet analyzed		
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.	5 - Not yet analyzed		
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.	5 - Not yet analyzed		
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 “_”.	5 - Not yet analyzed		

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	<a href="#">A3-1-2</a>	
AV Rule 53.1 The following character sequences shall not appear in header file names: ‘, \, /*, //, or ”.	2 - Small differences	<a href="#">A16-2-1</a>	
AV Rule 54 Implementation files will always have a file name extension of “.cpp”.	2 - Small differences	<a href="#">A3-1-3</a>	
AV Rule 55 The name of a header file should reflect the logical entity for which it provides declarations.	5 - Not yet analyzed		
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.)	5 - Not yet analyzed		



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	<a href="#">M6-3-1</a>	
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	<a href="#">A11-0-2</a>	
AV Rule 66 A class should be used to model an entity that maintains an invariant.	3 - Significant differences	<a href="#">A11-0-1</a>	
AV Rule 67 Public and protected data should only be used in structs - not classes.	2 - Small differences	<a href="#">M11-0-1</a>	

AV Rule 68 Unneeded implicitly generated member functions shall be explicitly disallowed.	5 - Not yet analyzed		
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	<a href="#">M9-3-3</a>	
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: <a href="#">A11-3-1</a> .
AV Rule 70.1 An object shall not be improperly used before its lifetime begins or after its lifetime ends.	5 - Not yet analyzed		
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.	5 - Not yet analyzed		
AV Rule 71.1 A class's virtual functions shall not be invoked from its destructor or any of its constructors.	2 - Small differences	<a href="#">M12-1-1</a>	
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.	5 - Not yet analyzed		
AV Rule 73 Unnecessary default constructors shall not be defined.	5 - Not yet analyzed		
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	<a href="#">A12-6-1</a>	
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	<a href="#">A8-5-1</a>	
AV Rule 76 A copy constructor and an assignment operator shall be declared for classes that contain pointers to data items or nontrivial destructors.	5 - Not yet analyzed		
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).	5 - Not yet analyzed		

AV Rule 77.1 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).	5 - Not yet analyzed		
AV Rule 78 All base classes with a virtual function shall define a virtual destructor.	2 - Small differences	<a href="#">A12-4-1</a>	
AV Rule 79 All resources acquired by a class shall be released by the class's destructor.	5 - Not yet analyzed		
AV Rule 80 The default copy and assignment operators will be used for classes when those operators offer reasonable semantics.	5 - Not yet analyzed		
AV Rule 81 The assignment operator shall handle self-assignment correctly.	2 - Small differences	<a href="#">A12-8-5</a>	
AV Rule 82 An assignment operator shall return a reference to *this.	5 - Not yet analyzed		
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).	5 - Not yet analyzed		
AV Rule 84 Operator overloading will be used sparingly and in a conventional manner.	5 - Not yet analyzed		
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.	5 - Not yet analyzed		
AV Rule 86 Concrete types should be used to represent simple independent concepts.	5 - Not yet analyzed		
AV Rule 87 Hierarchies should be based on abstract classes.	5 - Not yet analyzed		
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	<a href="#">A10-1-1</a>	
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: <a href="#">M10-1-1</a> .
AV Rule 89 A base class shall not be both virtual and non-virtual in the same hierarchy.	2 - Small differences	<a href="#">M10-1-3</a>	
AV Rule 90 Heavily used interfaces should be minimal, general and abstract.	5 - Not yet analyzed		
AV Rule 91 Public inheritance will be used to implement "is-a" relationships.	5 - Not yet analyzed		

<p>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.</p>	5 - Not yet analyzed		
<p>AV Rule 93 “has-a” or “is-implemented-in-terms-of” relationships will be modeled through membership or non-public inheritance.</p>	5 - Not yet analyzed		
<p>AV Rule 94 An inherited nonvirtual function shall not be redefined in a derived class.</p>	2 - Small differences	<a href="#">A10-2-1</a>	
<p>AV Rule 95 An inherited default parameter shall never be redefined.</p>	2 - Small differences	<a href="#">M8-3-1</a>	
<p>AV Rule 97 Arrays shall not be used in interfaces. Instead, the Array class should be used.</p>	2 - Small differences	<a href="#">M5-2-12</a>	
<p>AV Rule 97.1 Neither operand of an equality operator (== or !=) shall be a pointer to a virtual member function.</p>	5 - Not yet analyzed		
<p>AV Rule 98 Every nonlocal name, except main(), should be placed in some namespace.</p>	2 - Small differences	<a href="#">M7-3-1</a>	
<p>AV Rule 99 Namespaces will not be nested more than two levels deep.</p>			To be discussed.
<p>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.</p>	3 - Significant differences	<a href="#">M7-3-4</a> , <a href="#">M7-3-6</a>	
<p>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.</p>	5 - Not yet analyzed		

AV Rule 102 Template tests shall be created to cover all actual template instantiations.	5 - Not yet analyzed		
AV Rule 103 Constraint checks should be applied to template arguments.	5 - Not yet analyzed		
AV Rule 104 A template specialization shall be declared before its use.	5 - Not yet analyzed		
AV Rule 105 A template definition's dependence on its instantiation contexts should be minimized.	5 - Not yet analyzed		
AV Rule 106 Specializations for pointer types should be made where appropriate.	5 - Not yet analyzed		
AV Rule 107 Functions shall always be declared at file scope.	2 - Small differences	<a href="#">M3-1-2</a>	
AV Rule 108 Functions with variable numbers of arguments shall not be used.	2 - Small differences	<a href="#">A8-4-1</a>	
AV Rule 109 A function definition should not be placed in a class specification unless the function is intended to be inlined.	5 - Not yet analyzed		
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	<a href="#">M7-5-2</a>	
AV Rule 112 Function return values should not obscure resource ownership.	5 - Not yet analyzed		
AV Rule 113 Functions will have a single exit point.	4 - Rejected		Multiple points of exit are permitted by AUTOSAR C++ Coding Guidelines.
AV Rule 114 All exit points of value-returning functions shall be through return statements.	2 - Small differences	<a href="#">A8-4-2</a>	
AV Rule 115 If a function returns error information, then that error information will be tested.	2 - Small differences	<a href="#">A8-4-2</a>	
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.	5 - Not yet analyzed		
AV Rule 117 Arguments should be passed by reference if NULL values are not possible.	5 - Not yet analyzed		

AV Rule 117.1 An object should be passed as const T& if the function should not change the value of the object.	5 - Not yet analyzed		
AV Rule 117.2 An object should be passed as T& if the function may change the value of the object.	5 - Not yet analyzed		
AV Rule 118 Arguments should be passed via pointers if NULL values are possible.	5 - Not yet analyzed		
AV Rule 118.1 An object should be passed as const T* if its value should not be modified.	5 - Not yet analyzed		
AV Rule 118.2 An object should be passed as T* if its value may be modified.	5 - Not yet analyzed		
AV Rule 119 Functions shall not call themselves, either directly or indirectly (i.e. recursion shall not be allowed).	2 - Small differences	<a href="#">A7-5-2</a>	
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.	5 - Not yet analyzed		
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.	5 - Not yet analyzed		
AV Rule 123 The number of accessor and mutator functions should be minimized.	5 - Not yet analyzed		
AV Rule 124 Trivial forwarding functions should be inlined.	5 - Not yet analyzed		
AV Rule 125 Unnecessary temporary objects should be avoided.	5 - Not yet analyzed		
AV Rule 126 Only valid C++ style comments (//) shall be used.	4 - Rejected		
AV Rule 127 Code that is not used (commented out) shall be deleted.	2 - Small differences	<a href="#">A2-8-2</a>	
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.	5 - Not yet analyzed		
AV Rule 129 Comments in header files should describe the externally visible behavior of the functions or classes being documented.	3 - Significant differences	<a href="#">A2-8-3</a>	

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	<a href="#">A2-8-3</a>	
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	<a href="#">A2-11-1</a>	
AV Rule 136 Declarations should be at the smallest feasible scope.	2 - Small differences	<a href="#">M3-4-1</a>	
AV Rule 137 All declarations at file scope should be static where possible.	3 - Significant differences	<a href="#">A3-3-1</a>	
AV Rule 138 Identifiers shall not simultaneously have both internal and external linkage in the same translation unit.	2 - Small differences	<a href="#">M3-3-2</a>	
AV Rule 139 External objects will not be declared in more than one file.	2 - Small differences	<a href="#">M3-2-3</a>	
AV Rule 140 The register storage class specifier shall not be used.	2 - Small differences	<a href="#">A7-1-4</a>	
AV Rule 141 A class, structure, or enumeration will not be declared in the definition of its type.	5 - Not yet analyzed		
AV Rule 142 All variables shall be initialized before use.	2 - Small differences	<a href="#">M8-5-1</a>	
AV Rule 143 Variables will not be introduced until they can be initialized with meaningful values.	2 - Small differences	<a href="#">M3-4-1</a>	
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	<a href="#">M8-5-2</a>	



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	<a href="#">A7-2-4</a>	
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	<a href="#">A0-4-1</a>	
AV Rule 147 The underlying bit representations of floating point numbers shall not be used in any way by the programmer.	2 - Small differences	<a href="#">M3-9-3</a>	
AV Rule 148 Enumeration types shall be used instead of integer types (and constants) to select from a limited series of choices.	5 - Not yet analyzed		
AV Rule 149 Octal constants (other than zero) shall not be used.	2 - Small differences	<a href="#">M2-13-2</a>	
AV Rule 150 Hexadecimal constants will be represented using all uppercase letters.	5 - Not yet analyzed		
AV Rule 151 Numeric values in code will not be used; symbolic values will be used instead.	2 - Small differences	<a href="#">A5-1-1</a>	
AV Rule 151.1 A string literal shall not be modified.	5 - Not yet analyzed		
AV Rule 152 Multiple variable declarations shall not be allowed on the same line.	5 - Not yet analyzed		
AV Rule 153 Unions shall not be used.	2 - Small differences	<a href="#">M9-5-1</a>	
AV Rule 154 Bit-fields shall have explicitly unsigned integral or enumeration types only.	2 - Small differences	<a href="#">A9-6-1</a>	
AV Rule 155 Bit-fields will not be used to pack data into a word for the sole purpose of saving space.	5 - Not yet analyzed		
AV Rule 156 All the members of a structure (or class) shall be named and shall only be accessed via their names.	5 - Not yet analyzed		
AV Rule 157 The right hand operand of a && or   operator shall not contain side effects.	2 - Small differences	<a href="#">M5-14-1</a>	
AV Rule 158 The operands of a logical && or   shall be parenthesized if the operands contain binary operators.	2 - Small differences	<a href="#">M5-2-1</a>	
AV Rule 159 Operators   , &&, and unary & shall not be overloaded.	2 - Small differences	<a href="#">M5-2-11</a> , <a href="#">M5-3-3</a>	
AV Rule 160 An assignment expression shall be used only as the expression in an expression statement.	2 - Small differences	<a href="#">M6-2-1</a>	



AV Rule 162 Signed and unsigned values shall not be mixed in arithmetic or comparison operations.	5 - Not yet analyzed		
AV Rule 163 Unsigned arithmetic shall not be used.	5 - Not yet analyzed		
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	<a href="#">M5-8-1</a>	
AV Rule 164.1 The left-hand operand of a right-shift operator shall not have a negative value.	5 - Not yet analyzed		
AV Rule 165 The unary minus operator shall not be applied to an unsigned expression.	2 - Small differences	<a href="#">M5-3-2</a>	
AV Rule 166 The sizeof operator will not be used on expressions that contain side effects.	2 - Small differences	<a href="#">M5-3-4</a>	
AV Rule 167 The implementation of integer division in the chosen compiler shall be determined, documented and taken into account.	3 - Significant differences	<a href="#">A0-4-2</a>	
AV Rule 168 The comma operator shall not be used.	2 - Small differences	<a href="#">M5-18-1</a>	
AV Rule 169 Pointers to pointers should be avoided when possible.	3 - Significant differences	<a href="#">A5-0-3</a>	
AV Rule 170 More than 2 levels of pointer indirection shall not be used.	3 - Significant differences	<a href="#">A5-0-3</a>	
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	<a href="#">M5-0-18</a>	
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	<a href="#">M7-5-2</a>	
AV Rule 174 The null pointer shall not be de-referenced.	5 - Not yet analyzed		
AV Rule 175 A pointer shall not be compared to NULL or be assigned NULL; use plain 0 instead.	4 - Rejected		Only <code>nullptr</code> constant shall be used, see: <a href="#">A4-10-1</a> .
AV Rule 176 A typedef will be used to simplify program syntax when declaring function pointers.	5 - Not yet analyzed		
AV Rule 177 User-defined conversion functions should be avoided.	5 - Not yet analyzed		

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)	5 - Not yet analyzed		
AV Rule 179 A pointer to a virtual base class shall not be converted to a pointer to a derived class.	5 - Not yet analyzed		
AV Rule 180 Implicit conversions that may result in a loss of information shall not be used.	5 - Not yet analyzed		
AV Rule 181 Redundant explicit casts will not be used.	5 - Not yet analyzed		
AV Rule 182 Type casting from any type to or from pointers shall not be used.	5 - Not yet analyzed		
AV Rule 183 Every possible measure should be taken to avoid type casting.	5 - Not yet analyzed		
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	<a href="#">M5-0-5</a>	
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	<a href="#">A5-2-2</a>	
AV Rule 186 There shall be no unreachable code.	2 - Small differences	<a href="#">M0-1-1</a>	
AV Rule 187 All non-null statements shall potentially have a side-effect.	2 - Small differences	<a href="#">M0-1-9</a>	
AV Rule 188 Labels will not be used, except in switch statements.	3 - Significant differences	<a href="#">A6-6-1</a>	
AV Rule 189 The goto statement shall not be used.	2 - Small differences	<a href="#">A6-6-1</a>	
AV Rule 190 The continue statement shall not be used.	4 - Rejected		The continue statement usage is allowed within for-loops, see: <a href="#">M6-6-3</a> .
AV Rule 191 The break statement shall not be used (except to terminate the cases of a switch statement).	2 - Small differences	<a href="#">M6-4-5</a>	
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	<a href="#">M6-4-2</a>	

AV Rule 193 Every non-empty case clause in a switch statement shall be terminated with a break statement.	2 - Small differences	<a href="#">M6-4-3</a> , <a href="#">M6-4-5</a>	
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	<a href="#">M6-4-6</a>	
AV Rule 195 A switch expression will not represent a Boolean value.	2 - Small differences	<a href="#">M6-4-7</a>	
AV Rule 196 Every switch statement will have at least two cases and a potential default.	2 - Small differences	<a href="#">A6-4-1</a>	
AV Rule 197 Floating point variables shall not be used as loop counters.	2 - Small differences	<a href="#">M6-5-2</a>	
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.	5 - Not yet analyzed		
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.	5 - Not yet analyzed		
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	<a href="#">A6-5-2</a>	
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	<a href="#">M6-5-3</a>	
AV Rule 202 Floating point variables shall not be tested for exact equality or inequality.	2 - Small differences	<a href="#">M6-2-2</a>	
AV Rule 203 Evaluation of expressions shall not lead to overflow/underflow (unless required algorithmically and then should be heavily documented).	2 - Small differences	<a href="#">M5-19-1</a> , <a href="#">A7-1-2</a>	
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.	5 - Not yet analyzed		
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	<a href="#">A5-0-1</a>	
AV Rule 205 The volatile keyword shall not be used unless directly interfacing with hardware.	5 - Not yet analyzed		

AV Rule 206 Allocation/deallocation from/to the free store (heap) shall not occur after initialization.	2 - Small differences	<a href="#">A18-5-3</a>	
AV Rule 207 Unencapsulated global data will be avoided.	5 - Not yet analyzed		
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	<a href="#">A3-9-1</a>	
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.)	5 - Not yet analyzed		
AV Rule 210.1 Algorithms shall not make assumptions concerning the order of allocation of nonstatic data members separated by an access specifier.	5 - Not yet analyzed		
AV Rule 211 Algorithms shall not assume that shorts, ints, longs, floats, doubles or long doubles begin at particular addresses.	5 - Not yet analyzed		
AV Rule 212 Underflow or overflow functioning shall not be depended on in any special way.	5 - Not yet analyzed		
AV Rule 213 No dependence shall be placed on C++'s operator precedence rules, below arithmetic operators, in expressions.	2 - Small differences	<a href="#">A5-0-1</a>	
AV Rule 214 Assuming that non-local static objects, in separate translation units, are initialized in a special order shall not be done.	5 - Not yet analyzed		
AV Rule 215 Pointer arithmetic will not be used.	3 - Significant differences	<a href="#">M5-0-15</a>	Pointer arithmetic may be used for array indexing.
AV Rule 216 Programmers should not attempt to prematurely optimize code.	5 - Not yet analyzed		
AV Rule 217 Compile-time and link-time errors should be preferred over run-time errors.	5 - Not yet analyzed		

AV Rule 218 Compiler warning levels will be set in compliance with project policies.	3 - Significant differences	<a href="#">A1-1-2</a>	
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.	5 - Not yet analyzed		
AV Rule 220 Structural coverage algorithms shall be applied against flattened classes.	5 - Not yet analyzed		
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.	5 - Not yet analyzed		

**Table A.3: JSF**

## A.4 Traceability to SEI CERT C++

The following table demonstrates the traceability to SEI CERT C++ Coding Standard [9]. 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	<a href="#">M7-5-2</a>	
DCL40-C. Do not create incompatible declarations of the same function or object.	2 - Small differences	<a href="#">M3-9-1</a>	
DCL50-CPP. Do not define a C-style variadic function.	2 - Small differences	<a href="#">A8-4-1</a>	
DCL51-CPP. Do not declare or define a reserved identifier.	2 - Small differences	<a href="#">A13-1-2</a> , <a href="#">A17-0-1</a>	
DCL52-CPP. Never qualify a reference type with const or volatile.	5 - Not yet analyzed		
DCL53-CPP. Do not write syntactically ambiguous declarations.	2 - Small differences	<a href="#">A8-5-2</a>	
DCL54-CPP. Overload allocation and deallocation functions as a pair in the same scope.	3 - Significant differences	<a href="#">A18-5-3</a> , <a href="#">A18-5-4</a>	
DCL55-CPP. Avoid information leakage when passing a class object across a trust boundary.	5 - Not yet analyzed		

DCL56-CPP. Avoid cycles during initialization of static objects.	5 - Not yet analyzed		
DCL57-CPP. Do not let exceptions escape from destructors or deallocation functions.	2 - Significant differences	<a href="#">A15-5-1</a>	AUTOSAR C++ Coding Guidelines specifies more functions that need to be noexcept.
DCL58-CPP. Do not modify the standard namespaces.	5 - Not yet analyzed		
DCL59-CPP. Do not define an unnamed namespace in a header file.	2 - Small differences	<a href="#">M7-3-3</a>	
DCL60-CPP. Obey the one-definition rule.	2 - Small differences	<a href="#">M3-2-2</a>	
EXP34-C. Do not dereference null pointers.	5 - Not yet analyzed		
EXP35-C. Do not modify objects with temporary lifetime.	5 - Not yet analyzed		
EXP36-C. Do not cast pointers into more strictly aligned pointer types.	5 - Not yet analyzed		
EXP37-C. Call functions with the correct number and type of arguments.	5 - Not yet analyzed		
EXP39-C. Do not access a variable through a pointer of an incompatible type.	5 - Not yet analyzed		
EXP42-C. Do not compare padding data.	5 - Not yet analyzed		
EXP45-C. Do not perform assignments in selection statements.	2 - Small differences	<a href="#">A5-0-2</a> , <a href="#">M6-2-1</a>	
EXP46-C. Do not use a bitwise operator with a Boolean-like operand.	4 - Rejected		Use of bitwise operators restricted to following cases: <a href="#">M5-0-10</a> , <a href="#">M5-0-20</a> , <a href="#">M5-0-21</a> .
EXP47-C. Do not call <code>va_arg</code> with an argument of the incorrect type.	4 - Rejected		Use of variable arguments are prohibited, see: <a href="#">A8-4-1</a> .
EXP50-CPP. Do not depend on the order of evaluation for side effects.	2 - Small differences	<a href="#">A5-0-1</a>	
EXP51-CPP. Do not delete an array through a pointer of the incorrect type.	5 - Not yet analyzed		
EXP52-CPP. Do not rely on side effects in unevaluated operands.	3 - Significant differences	<a href="#">M5-3-4</a> , <a href="#">A5-3-1</a>	
EXP53-CPP. Do not read uninitialized memory.	5 - Not yet analyzed		
EXP54-CPP. Do not access an object outside of its lifetime.	5 - Not yet analyzed		

EXP55-CPP. Do not access a cv-qualified object through a cv-unqualified type.	2 - Small differences	<a href="#">A5-2-3</a>	
EXP56-CPP. Do not call a function with a mismatched language linkage.	5 - Not yet analyzed		
EXP57-CPP. Do not cast or delete pointers to incomplete classes.	5 - Not yet analyzed		
EXP58-CPP. Pass an object of the correct type to va_start.	4 - Rejected		Use of variable arguments are prohibited, see: <a href="#">A8-4-1</a> .
EXP59-CPP. Use offsetof() on valid types and members.	4 - Rejected		Use of offsetof() is prohibited, see: <a href="#">M18-2-1</a> .
EXP60-CPP. Do not pass a nonstandard-layout type object across execution boundaries.	5 - Not yet analyzed		
EXP61-CPP. A lambda object must not outlive any of its reference captured objects.	2 - Small differences	<a href="#">A5-1-4</a>	
EXP62-CPP. Do not access the bits of an object representation that are not part of the object's value representation.	5 - Not yet analyzed		
EXP63-CPP. Do not rely on the value of a moved-from object.	2 - Small differences	<a href="#">A12-8-3</a>	
INT30-C. Ensure that unsigned integer operations do not wrap.	2 - Small differences	<a href="#">A4-7-1</a> , <a href="#">M5-19-1</a>	
INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data.	3 - Significant differences	<a href="#">A4-7-1</a> , <a href="#">M5-0-15</a>	
INT32-C. Ensure that operations on signed integers do not result in overflow.	2 - Small differences	<a href="#">A4-7-1</a>	
INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors.	2 - Small differences	<a href="#">A5-5-1</a>	
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	<a href="#">M5-8-1</a>	
INT35-C. Use correct integer precisions.	3 - Significant differences	<a href="#">A3-9-1</a>	
INT36-C. Converting a pointer to integer or integer to pointer.	2 - Small differences	<a href="#">M5-2-8</a> , <a href="#">M5-2-9</a>	
INT50-CPP. Do not cast to an out-of-range enumeration value.	2 - Small differences	<a href="#">A7-2-1</a>	
CTR50-CPP. Guarantee that container indices and iterators are within the valid range.	3 - Significant differences	<a href="#">A5-2-5</a>	



CTR51-CPP. Use valid references, pointers, and iterators to reference elements of a container.	3 - Significant differences	<a href="#">M5-0-16</a> , <a href="#">M5-0-17</a>	
CTR52-CPP. Guarantee that library functions do not overflow.	5 - Not yet analyzed		
CTR53-CPP. Use valid iterator ranges.	3 - Significant differences	<a href="#">M5-0-16</a> , <a href="#">M5-0-17</a>	
CTR54-CPP. Do not subtract iterators that do not refer to the same container.	3 - Significant differences	<a href="#">M5-0-16</a> , <a href="#">M5-0-17</a>	
CTR55-CPP. Do not use an additive operator on an iterator if the result would overflow.	3 - Significant differences	<a href="#">M5-0-16</a> , <a href="#">M5-0-17</a>	
CTR56-CPP. Do not use pointer arithmetic on polymorphic objects.	5 - Not yet analyzed		
CTR57-CPP. Provide a valid ordering predicate.	5 - Not yet analyzed		
CTR58-CPP. Predicate function objects should not be mutable.	5 - Not yet analyzed		
ARR30-C. Do not form or use out-of-bounds pointers or array subscripts.	3 - Significant differences	<a href="#">A5-2-5</a>	
ARR37-C. Do not add or subtract an integer to a pointer to a non-array object.	3 - Significant differences	<a href="#">M5-0-15</a>	
ARR38-C. Guarantee that library functions do not form invalid pointers.	5 - Not yet analyzed		
ARR39-C. Do not add or subtract a scaled integer to a pointer.	3 - Significant differences	<a href="#">M5-0-15</a>	
STR30-C. Do not attempt to modify string literals.	4 - Rejected		Use of C-style arrays, apart from static constexpr members, is prohibited. See: <a href="#">A18-1-1</a> .
STR32-C. Do not pass a non-null-terminated character sequence to a library function that expects a string	5 - Not yet analyzed		
STR34-C. Cast characters to unsigned char before converting to larger integer sizes.	5 - Not yet analyzed		
STR37-C. Arguments to character-handling functions must be representable as an unsigned char.	5 - Not yet analyzed		
STR38-C. Do not confuse narrow and wide character strings and functions.	5 - Not yet analyzed		
STR51-CPP. Do not attempt to create a <code>std::string</code> from a null pointer.	5 - Not yet analyzed		
STR52-CPP. Use valid references, pointers, and iterators to reference elements of a <code>basic_string</code> .	5 - Not yet analyzed		



STR53-CPP. Range check element access.	3 - Significant differences	<a href="#">A5-2-5</a>	The specific case of <a href="#">A5-2-5</a> .
MEM31-C. Free dynamically allocated memory when no longer needed.	5 - Not yet analyzed		
MEM34-C. Only free memory allocated dynamically.	5 - Not yet analyzed		
MEM35-C. Allocate sufficient memory for an object.	4 - Rejected		Use of malloc, calloc and realloc functions is prohibited, see: <a href="#">A18-5-1</a> .
MEM36-C. Do not modify the alignment of objects by calling realloc().	4 - Rejected		Use of malloc, calloc and realloc functions is prohibited, see: <a href="#">A18-5-1</a> .
MEM50-CPP. Do not access freed memory.	5 - Not yet analyzed		
MEM51-CPP. Properly deallocate dynamically allocated resources.	3 - Significant differences	<a href="#">A18-5-3</a>	Use of memory allocation and deallocation operators limited by <a href="#">A18-5-2</a> , <a href="#">A18-5-4</a> .
MEM52-CPP. Detect and handle memory allocation errors.	3 - Significant differences	<a href="#">A15-0-2</a> , <a href="#">A15-2-2</a> , <a href="#">A15-3-3</a> , <a href="#">A15-5-3</a>	
MEM53-CPP. Explicitly construct and destruct objects when manually managing object lifetime.	4 - Rejected	<a href="#">A18-5-2</a>	Explicit use of operators new and delete is prohibited. Managing object lifetime also covered by <a href="#">A18-5-1</a> , <a href="#">A18-5-3</a> .
MEM54-CPP. Provide placement new with properly aligned pointers to sufficient storage capacity.	5 - Not yet analyzed		
MEM55-CPP. Honor replacement dynamic storage management requirements.	5 - Not yet analyzed		
MEM56-CPP. Do not store an already-owned pointer value in an unrelated smart pointer.	5 - Not yet analyzed		
MEM57-CPP. Avoid using default operator new for over-aligned types.	5 - Not yet analyzed		

FIO50-CPP. Do not alternately input and output from a file stream without an intervening positioning call.	5 - Not yet analyzed		
FIO51-CPP. Close files when they are no longer needed.	5 - Not yet analyzed		
FIO30-C. Exclude user input from format strings.	2 - Small differences	<a href="#">A27-0-1</a>	
FIO32-C. Do not perform operations on devices that are only appropriate for files.	5 - Not yet analyzed		
FIO34-C. Distinguish between characters read from a file and EOF or WEOF.	5 - Not yet analyzed		
FIO37-C. Do not assume that fgets() or fgetws() returns a nonempty string when successful.	5 - Not yet analyzed		
FIO38-C. Do not copy a FILE object.	5 - Not yet analyzed		
FIO39-C. Do not alternately input and output from a stream without an intervening flush or positioning call.	5 - Not yet analyzed		
FIO40-C. Reset strings on fgets() or fgetws() failure.	5 - Not yet analyzed		
FIO41-C. Do not call getc(), putc(), getwc(), or putwc() with a stream argument that has side effects.	5 - Not yet analyzed		
FIO42-C. Close files when they are no longer needed.	5 - Not yet analyzed		
FIO44-C. Only use values for fsetpos() that are returned from fgetpos().	5 - Not yet analyzed		
FIO45-C. Avoid TOCTOU race conditions while accessing files.	5 - Not yet analyzed		
FIO46-C. Do not access a closed file.	5 - Not yet analyzed		
FIO47-C. Use valid format strings.	5 - Not yet analyzed		
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: <a href="#">M19-3-1</a> .
ERR32-C. Do not rely on indeterminate values of errno.	4 - Rejected		Use of the errno is prohibited, see: <a href="#">M19-3-1</a> .
ERR33-C. Detect and handle standard library errors.	3 - Small differences	<a href="#">M0-3-2</a> , <a href="#">A15-0-3</a>	
ERR50-CPP. Do not abruptly terminate the program.	2 - Small differences	<a href="#">A15-5-2</a> , <a href="#">A15-5-3</a>	
ERR51-CPP. Handle all exceptions.	2 - Small differences	<a href="#">A15-3-3</a> , <a href="#">A15-5-3</a>	
ERR52-CPP. Do not use setjmp() or longjmp().	2 - Small differences	<a href="#">M17-0-5</a>	

ERR53-CPP. Do not reference base classes or class data members in a constructor or destructor function-try-block handler.	3 - Significant differences	<a href="#">M15-3-3</a>	Use of function-try-blocks is anyway not recommended. See: <a href="#">A15-3-5</a> .
ERR54-CPP. Catch handlers should order their parameter types from most derived to least derived.	2 - Small differences	<a href="#">M15-3-6</a> , <a href="#">M15-3-7</a>	
ERR55-CPP. Honor exception specifications.	3 - Significant differences	<a href="#">A15-4-2</a>	Use of dynamic exception specification is prohibited, see: <a href="#">A15-4-1</a> . The noexcept specifier should be used instead.
ERR56-CPP. Guarantee exception safety.	2 - Small differences	<a href="#">A15-0-2</a>	
ERR57-CPP. Do not leak resources when handling exceptions.	3 - Significant differences	<a href="#">A15-0-2</a> , <a href="#">A15-1-2</a> , <a href="#">A15-1-4</a>	
ERR58-CPP. Handle all exceptions thrown before main() begins executing.	2 - Small differences	<a href="#">A15-2-1</a>	
ERR59-CPP. Do not throw an exception across execution boundaries.	2 - Small differences	<a href="#">A15-1-5</a>	
ERR60-CPP. Exception objects must be nothrow copy constructible.	3 - Significant differences	<a href="#">A15-5-3</a>	
ERR61-CPP. Catch exceptions by lvalue reference.	2 - Small differences	<a href="#">A15-3-5</a>	
ERR62-CPP. Detect errors when converting a string to a number.	5 - Not yet analyzed		
OOP50-CPP. Do not invoke virtual functions from constructors or destructors.	2 - Small differences	<a href="#">M12-1-1</a>	
OOP51-CPP. Do not slice derived objects.	2 - Small differences	<a href="#">A12-8-6</a> , <a href="#">A15-3-5</a>	
OOP52-CPP. Do not delete a polymorphic object without a virtual destructor.	2 - Small differences	<a href="#">A12-4-1</a> , <a href="#">A12-4-2</a>	
OOP53-CPP. Write constructor member initializers in the canonical order.	2 - Small differences	<a href="#">A8-5-1</a>	
OOP54-CPP. Gracefully handle self-copy assignment.	2 - Small differences	<a href="#">A12-8-5</a>	
OOP55-CPP. Do not use pointer-to-member operators to access nonexistent members.	5 - Not yet analyzed		
OOP56-CPP. Honor replacement handler requirements.	5 - Not yet analyzed		

OOP57-CPP. Prefer special member functions and overloaded operators to C Standard Library functions.	5 - Not yet analyzed		
OOP58-CPP. Copy operations must not mutate the source object.	2 - Small differences	<a href="#">A12-8-1</a>	
CON50-CPP. Do not destroy a mutex while it is locked.	5 - Not yet analyzed		
CON51-CPP. Ensure actively held locks are released on exceptional conditions.	5 - Not yet analyzed		
CON52-CPP. Prevent data races when accessing bit-fields from multiple threads.	5 - Not yet analyzed		
CON53-CPP. Avoid deadlock by locking in a predefined order.	5 - Not yet analyzed		
CON54-CPP. Wrap functions that can spuriously wake up in a loop.	5 - Not yet analyzed		
CON55-CPP. Preserve thread safety and liveness when using condition variables.	5 - Not yet analyzed		
CON56-CPP. Do not speculatively lock a non-recursive mutex that is already owned by the calling thread.	5 - Not yet analyzed		
CON33-C. Avoid race conditions when using library functions.	5 - Not yet analyzed		
CON37-C. Do not call signal() in a multithreaded program	4 - Rejected		Use of signal handling facilities of <code>&lt;csignal&gt;</code> is prohibited, see: <a href="#">M18-7-1</a> .
CON40-C. Do not refer to an atomic variable twice in an expression.	5 - Not yet analyzed		
CON41-C. Wrap functions that can fail spuriously in a loop.	5 - Not yet analyzed		
CON43-C. Do not allow data races in multithreaded code.	5 - Not yet analyzed		
MSC33-C. Do not pass invalid data to the <code>asctime()</code> function.	4 - Rejected		Use of time handling functions of <code>&lt;ctime&gt;</code> is prohibited, see: <a href="#">M18-0-4</a> .
MSC38-C. Do not treat a predefined identifier as an object if it might only be implemented as a macro.	4 - Rejected		Error indicator <code>errno</code> , <code>setjmp()</code> and variadic arguments shall not be used, see: <a href="#">M19-3-1</a> , <a href="#">M17-0-5</a> , <a href="#">A8-4-1</a> .

MSC39-C. Do not call <code>va_arg()</code> on a <code>va_list</code> that has an indeterminate value.	4 - Rejected		Use of variadic arguments is prohibited, see: <a href="#">A8-4-1</a> .
MSC40-C. Do not violate constraints.	5 - Not yet analyzed		
MSC50-CPP. Do not use <code>std::rand()</code> for generating pseudorandom numbers.	5 - Not yet analyzed		
MSC51-CPP. Ensure your random number generator is properly seeded.	5 - Not yet analyzed		
MSC52-CPP. Value-returning functions must return a value from all exit paths.	2 - Small differences	<a href="#">A8-4-2</a>	
MSC53-CPP. Do not return from a function declared <code>[[noreturn]]</code> .	5 - Not yet analyzed		
MSC54-CPP. A signal handler must be a plain old function.	4 - Rejected		Use of signal handling facilities of <code>&lt;csignal&gt;</code> is prohibited, see: <a href="#">M18-7-1</a> .
FLP30-C. Do not use floating-point variables as loop counters.	2 - Small differences	<a href="#">A6-5-2</a>	
FLP32-C. Prevent or detect domain and range errors in math functions.	5 - Not yet analyzed		
FLP34-C. Ensure that floating-point conversions are within range of the new type.	3 - Significant differences	<a href="#">M5-0-5</a> , <a href="#">M5-0-6</a> , <a href="#">M5-0-7</a>	
FLP36-C. Preserve precision when converting integral values to floating-point type.	3 - Significant differences	<a href="#">M5-0-5</a> , <a href="#">M5-0-6</a> , <a href="#">M5-0-7</a>	
FLP37-C. Do not use object representations to compare floating-point values.	2 - Small differences	<a href="#">M3-9-3</a>	
ENV30-C. Do not modify the object referenced by the return value of certain functions.	5 - Not yet analyzed		
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	<a href="#">A15-5-2</a> , <a href="#">A15-5-3</a>	
ENV33-C. Do not call <code>system()</code> .	5 - Not yet analyzed		
ENV34-C. Do not store pointers returned by certain functions.	3 - Significant differences	<a href="#">A18-0-3</a> , <a href="#">M19-3-1</a>	

SIG31-C. Do not access shared objects in signal handlers.	4 - Rejected		Use of signal handling facilities of <code>&lt;csignal&gt;</code> is prohibited, see: <a href="#">M18-7-1</a> .
SIG34-C. Do not call <code>signal()</code> from within interruptible signal handlers.	4 - Rejected		Use of signal handling facilities of <code>&lt;csignal&gt;</code> is prohibited, see: <a href="#">M18-7-1</a> .
SIG35-C. Do not return from a computational exception signal handler.	4 - Rejected		Use of signal handling facilities of <code>&lt;csignal&gt;</code> is prohibited, see: <a href="#">M18-7-1</a> .
PRE30-C. Do not create a universal character name through concatenation.	5 - Not yet analyzed		
PRE31-C. Avoid side effects in arguments to unsafe macros.	3 - Significant differences		Defining function-like macros is prohibited, see: <a href="#">A16-0-1</a> .
PRE32-C. Do not use preprocessor directives in invocations of function-like macros.			Defining function-like macros is prohibited, see: <a href="#">A16-0-1</a> .

**Table A.4: SEI CERT C++**

## A.5 Traceability to C++ Core Guidelines

The following table demonstrates the traceability to C++ Core Guidelines [10]. 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	<a href="#">A0-4-3</a>	
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: <a href="#">A5-2-1</a> , <a href="#">A5-2-2</a> , <a href="#">A5-2-4</a> , <a href="#">M5-2-12</a> , <a href="#">A8-5-2</a> , <a href="#">M9-5-1</a>
P.5: Prefer compile-time checking to run-time checking.	3 - Significant differences	<a href="#">M0-3-1</a>	
P.6: What cannot be checked at compile time should be checkable at run time.	3 - Significant differences	<a href="#">A0-1-2</a> , <a href="#">M0-3-2</a>	
P.7: Catch run-time errors early.	3 - Significant differences	<a href="#">A0-1-2</a> , <a href="#">M0-3-2</a> , <a href="#">A5-2-5</a> , <a href="#">A15-0-4</a> , <a href="#">A15-0-5</a>	
P.8: Don't leak any resources.	3 - Significant differences	<a href="#">A18-5-1</a> , <a href="#">A18-5-2</a> , <a href="#">A15-1-4</a>	
P.9: Don't waste time or space.	3 - Significant differences	<a href="#">M0-1-1</a> , <a href="#">A0-1-1</a> , <a href="#">M0-1-8</a> , <a href="#">M0-1-9</a>	
P.10: Prefer immutable data to mutable data.	2 - Small differences	<a href="#">A7-1-1</a>	
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.	2 - Small differences	<a href="#">A3-3-2</a>	
I.3: Avoid singletons.	2 - Small differences	<a href="#">A3-3-2</a>	
I.4: Make interfaces precisely and strongly typed.	5 - Not yet analyzed	-	-
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	<a href="#">A14-1-1</a>	
I.10: Use exceptions to signal a failure to perform a required task.	2 - Small differences	<a href="#">A15-0-1</a>	
I.11: Never transfer ownership by a raw pointer (T*).	5 - Not yet analyzed	-	-
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	<a href="#">M5-2-12</a>	
I.22: Avoid complex initialization of global objects.	3 - Significant differences	<a href="#">A3-3-2</a>	
I.23: Keep the number of function arguments low.	4 - Rejected		AUTOSAR C++ Coding Guidelines does not define code metrics, see: <a href="#">A1-4-1</a> .
I.24: Avoid adjacent unrelated parameters of the same type.	5 - Not yet analyzed	-	-
I.25: Prefer abstract classes as interfaces to class hierarchies.	3 - Significant differences	<a href="#">A10-1-1</a>	Multiple inheritance limited to only one base class, but multiple interface classes can be inherited.
I.26: If you want a cross-compiler ABI, use a C-style subset.	5 - Not yet analyzed	-	-
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: <a href="#">A1-4-1</a> .
F.4: If a function may have to be evaluated at compile time, declare it constexpr.	2 - Small differences	<a href="#">A7-1-2</a>	
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: <a href="#">A1-4-1</a> .
F.6: If your function may not throw, declare it noexcept.	2 - Small differences	<a href="#">A15-4-4</a>	
F.7: For general use, take T* or T& arguments rather than smart pointers.	5 - Not yet analyzed	-	-
F.8: Prefer pure functions.	5 - Not yet analyzed	-	-
F.15: Prefer simple and conventional ways of passing information.	5 - Not yet analyzed	-	-
F.16: For “in” parameters, pass cheaply-copied types by value and others by reference to const.	5 - Not yet analyzed	-	-
F.17: For “in-out” parameters, pass by reference to non-const.	5 - Not yet analyzed	-	-



F.18: For “consume” parameters, pass by X&& and std::move the parameter.	5 - Not yet analyzed	-	-
F.19: For “forward” parameters, pass by TP&& and only std::forward the parameter.	5 - Not yet analyzed	-	-
F.20: For “out” output values, prefer return values to output parameters.	5 - Not yet analyzed	-	-
F.21: To return multiple “out” values, prefer returning a tuple or struct.	5 - Not yet analyzed	-	-
F.22: Use T* or owner<T*> to designate a single object.	3 - Significant differences	<a href="#">M5-2-12</a>	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.	4 - R	-	-
F.27: Use a shared_ptr<T> to share ownership.	5 - Not yet analyzed	-	-
F.60: Prefer T* over T& when no argument is a valid option.	5 - Not yet analyzed	-	-
F.42: Return a T* to indicate a position (only).	5 - Not yet analyzed	-	-
F.43: Never (directly or indirectly) return a pointer or a reference to a local object.	2 - Small differences	<a href="#">M7-5-2</a>	
F.44: Return a T& when copy is undesirable and returning no object isn’t needed.	5 - Not yet analyzed	-	-
F.45: Don’t return a T&&.	5 - Not yet analyzed	-	-
F.46: int is the return type for main().	5 - Not yet analyzed	-	-
F.47: Return T& from assignment operators.	5 - Not yet analyzed	-	-

F.50: Use a lambda when a function won't do (to capture local variables, or to write a local function).	5 - Not yet analyzed	-	-
F.51: Where there is a choice, prefer default arguments over overloading.	5 - Not yet analyzed	-	-
F.52: Prefer capturing by reference in lambdas that will be used locally, including passed to algorithms.	2 - Small differences	<a href="#">A5-1-5</a>	
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	<a href="#">A5-1-4</a>	
F.54: If you capture this, capture all variables explicitly (no default capture).	3 - Significant differences	<a href="#">A5-1-2</a>	AUTOSAR C++ Coding Guidelines prohibits implicit variables capturing into a lambda expression.
C.1: Organize related data into structures (structs or classes).	5 - Not yet analyzed	-	-
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: <a href="#">A11-0-1</a> ), and a struct for types defined in <a href="#">A11-0-2</a> .
C.3: Represent the distinction between an interface and an implementation using a class.	5 - Not yet analyzed	-	-
C.4: Make a function a member only if it needs direct access to the representation of a class.	2 - Small differences	<a href="#">M9-3-3</a>	
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.	5 - Not yet analyzed	-	-
C.8: Use class rather than struct if any member is non-public.	2 - Small differences	<a href="#">M11-0-1</a> , <a href="#">A11-0-1</a>	
C.9: Minimize exposure of members.	3 - Significant differences	<a href="#">M9-3-1</a> , <a href="#">A9-3-1</a> , <a href="#">M11-0-1</a>	
C.10 Prefer concrete types over class hierarchies.	5 - Not yet analyzed	-	-
C.11: Make concrete types regular.	5 - Not yet analyzed	-	-
C.20: If you can avoid defining default operations, do.	2 - Small differences	<a href="#">A12-0-1</a>	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	<a href="#">A12-0-1</a>	
C.22: Make default operations consistent.	2 - Small differences	<a href="#">A12-1-1</a> , <a href="#">A12-8-1</a>	
C.30: Define a destructor if a class needs an explicit action at object destruction.	5 - Not yet analyzed	-	-
C.31: All resources acquired by a class must be released by the class's destructor.	5 - Not yet analyzed	-	-
C.32: If a class has a raw pointer (T*) or reference (T&), consider whether it might be owning.	4 - Rejected		Non-generic design principle; There is no need for a new rule.
C.33: If a class has an owning pointer member, define a destructor.	4 - Rejected		The rule is vague.
C.34: If a class has an owning reference member, define a destructor.	4 - Rejected		The rule is vague.
C.35: A base class destructor should be either public and virtual, or protected and nonvirtual.	2 - Small differences	<a href="#">A12-4-1</a>	
C.36: A destructor may not fail.	2 - Small differences	<a href="#">A15-5-1</a>	
C.37: Make destructors noexcept.	2 - Small differences	<a href="#">A15-5-1</a>	
C.40: Define a constructor if a class has an invariant.	5 - Not yet analyzed	-	-
C.41: A constructor should create a fully initialized object.	2 - Small differences	<a href="#">A15-2-2</a>	
C.42: If a constructor cannot construct a valid object, throw an exception.	2 - Small differences	<a href="#">A15-2-2</a>	
C.43: Ensure that a class has a default constructor.	5 - Not yet analyzed	-	-
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	<a href="#">A12-1-3</a> , <a href="#">A12-7-1</a>	
C.46: By default, declare single-argument constructors explicit.	2 - Small differences	<a href="#">A12-1-4</a>	
C.47: Define and initialize member variables in the order of member declaration.	2 - Small differences	<a href="#">A8-5-1</a>	

C.48: Prefer in-class initializers to member initializers in constructors for constant initializers.	3 - Significant differences	<a href="#">A12-1-3</a>	AUTOSAR C++ Coding Guidelines states that NSDMI shall not be mixed with member initializer list of constructors, see: <a href="#">A12-1-2</a> .
C.49: Prefer initialization to assignment in constructors.	2 - Small differences	<a href="#">A12-6-1</a>	
C.50: Use a factory function if you need “virtual behavior” during initialization.	4 - Rejected		Non-generic design principle; There is no need for a new rule.
C.51: Use delegating constructors to represent common actions for all constructors of a class.	5 - Not yet analyzed	-	-
C.52: Use inheriting constructors to import constructors into a derived class that does not need further explicit initialization.	5 - Not yet analyzed	-	-
C.60: Make copy assignment non-virtual, take the parameter by const&, and return by non-const&.	2 - Small differences	<a href="#">A10-3-5</a> , <a href="#">A13-2-1</a>	
C.61: A copy operation should copy.	2 - Small differences	<a href="#">A12-8-1</a> , <a href="#">A12-8-2</a>	
C.62: Make copy assignment safe for self-assignment.	2 - Small differences	<a href="#">A12-8-5</a>	
C.63: Make move assignment non-virtual, take the parameter by &&, and return by non-const&.	2 - Small differences	<a href="#">A10-3-5</a> , <a href="#">A13-2-1</a>	
C.64: A move operation should move and leave its source in a valid state.	2 - Small differences	<a href="#">A12-8-1</a> , <a href="#">A12-8-4</a>	
C.65: Make move assignment safe for self-assignment.	2 - Small differences	<a href="#">A12-8-5</a>	
C.66: Make move operations noexcept.	2 - Small differences	<a href="#">A15-5-1</a>	
C.67: A base class should suppress copying, and provide a virtual clone instead if copying” is desired.	2 - Small differences	<a href="#">A12-8-6</a>	
C.80: Use =default if you have to be explicit about using the default semantics.	2 - Small differences	<a href="#">A12-7-1</a>	
C.81: Use =delete when you want to disable default behavior (without wanting an alternative).	5 - Not yet analyzed	-	-
C.82: Don’t call virtual functions in constructors and destructors.	2 - Small differences	<a href="#">M12-1-1</a>	

C.83: For value-like types, consider providing a noexcept swap function.	3 - Significant differences	<a href="#">A12-8-2</a>	The swap function is explicitly recommended for copy and move assignment operators only.
C.84: A swap function may not fail.	2 - Small differences	<a href="#">A15-5-1</a>	
C.85: Make swap noexcept.	2 - Small differences	<a href="#">A15-5-1</a>	
C.86: Make == symmetric with respect to operand types and noexcept.	5 - Not yet analyzed	-	-
C.87: Beware of == on base classes.	5 - Not yet analyzed	-	-
C.89: Make a hash noexcept.	5 - Not yet analyzed	-	-
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: <a href="#">Interface-Class</a> .
C.122: Use abstract classes as interfaces when complete separation of interface and implementation is needed.	4 - Rejected		Non-generic design principle; There is no need for a new rule.
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	<a href="#">A12-4-1</a>	
C.128: Virtual functions should specify exactly one of virtual, override, or final.	2 - Small differences	<a href="#">A10-3-1</a>	
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	<a href="#">A12-8-6</a>	
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	<a href="#">M11-0-1</a>	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	<a href="#">M11-0-1</a> , <a href="#">A11-0-2</a>	
C.135: Use multiple inheritance to represent multiple distinct interfaces.	2 - Small differences	<a href="#">A10-1-1</a>	
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: <a href="#">M10-1-1</a> , <a href="#">M10-1-2</a> .
C.138: Create an overload set for a derived class and its bases with using.	5 - Not yet analyzed	-	-
C.139: Use final sparingly.	4 - Rejected		Non-generic design principle; There is no need for a new rule.
C.140: Do not provide different default arguments for a virtual function and an overrider.	2 - Small differences	<a href="#">M8-3-1</a>	
C.145: Access polymorphic objects through pointers and references.	5 - Not yet analyzed	-	-
C.146: Use <code>dynamic_cast</code> where class hierarchy navigation is unavoidable.	2 - Small differences	<a href="#">A5-2-1</a>	

C.147: Use <code>dynamic_cast</code> to a reference type when failure to find the required class is considered an error.	4 - Rejected		The <code>dynamic_cast</code> should not be used, see: <a href="#">A5-2-1</a> .
C.148: Use <code>dynamic_cast</code> to a pointer type when failure to find the required class is considered a valid alternative.	4 - Rejected		The <code>dynamic_cast</code> should not be used, see: <a href="#">A5-2-1</a> .
C.149: Use <code>unique_ptr</code> or <code>shared_ptr</code> to avoid forgetting to delete objects created using <code>new</code> .	2 - Small differences	<a href="#">A18-5-2</a>	
C.150: Use <code>make_unique()</code> to construct objects owned by <code>unique_ptr</code> s.	2 - Small differences	<a href="#">A18-5-2</a>	
C.151: Use <code>make_shared()</code> to construct objects owned by <code>shared_ptr</code> s.	2 - Small differences	<a href="#">A18-5-2</a>	
C.152: Never assign a pointer to an array of derived class objects to a pointer to its base.	5 - Not yet analyzed	-	-
C.160: Define operators primarily to mimic conventional usage.	5 - Not yet analyzed	-	-
C.161: Use nonmember functions for symmetric operators.	5 - Not yet analyzed	-	-
C.162: Overload operations that are roughly equivalent.	5 - Not yet analyzed	-	-
C.163: Overload only for operations that are roughly equivalent.	5 - Not yet analyzed	-	-
C.164: Avoid conversion operators.	5 - Not yet analyzed	-	-
C.165: Use <code>using</code> for customization points.	5 - Not yet analyzed	-	-
C.166: Overload unary <code>&amp;</code> only as part of a system of smart pointers and references.	3 - Significant differences	<a href="#">M5-3-3</a>	The unary <code>&amp;</code> operator shall not be overloaded.
C.168: Define overloaded operators in the namespace of their operands.	5 - Not yet analyzed	-	-
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.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: <a href="#">A7-1-5</a> .

C.180: Use unions to save memory.	4 - Rejected		Unions shall not be used, see: <a href="#">M9-5-1</a> .
C.181: Avoid “naked” unions.	2 - Small differences	<a href="#">M9-5-1</a>	
C.182: Use anonymous unions to implement tagged unions.	5 - Not yet analyzed	-	-
C.183: Don’t use a union for type punning.	4 - Rejected		Unions shall not be used, see: <a href="#">M9-5-1</a> .
Enum.1: Prefer enumerations over macros.	3 - Significant differences	<a href="#">A16-0-1</a>	Usage of macros is prohibited.
Enum.2: Use enumerations to represent sets of related named constants.	4 - Rejected		Design principle; There is no need for a new rule.
Enum.3: Prefer class enums over “plain” enums.	2 - Small differences	<a href="#">A7-2-3</a>	
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	<a href="#">A7-2-3</a>	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: <a href="#">A3-9-1</a> .



Enum.8: Specify enumerator values only when necessary.	3 - Significant differences	<a href="#">A7-2-4</a>	It is defined how enumerators values should be specified.
R.1: Manage resources automatically using resource handles and RAI (Resource Acquisition Is Initialization).	4 - Rejected		Design principle; There is no need for a new rule.
R.2: In interfaces, use raw pointers to denote individual objects (only).	4 - Rejected		Design principle; There is no need for a new rule.
R.3: A raw pointer (a T*) is non-owning.	4 - Rejected		Design principle; There is no need for a new rule.
R.4: A raw reference (a T&) is non-owning.	4 - Rejected		Design principle; There is no need for a new rule.
R.5: Don't heap-allocate unnecessarily.	4 - Rejected		Design principle; There is no need for a new rule.
R.6: Avoid non-const global variables.	3 - Significant differences	<a href="#">A3-3-2</a>	There shall be no non-POD type objects with static storage duration besides static constexpr variables.
R.10: Avoid malloc() and free().	2 - Small differences	<a href="#">A18-5-1</a>	
R.11: Avoid calling new and delete explicitly.	2 - Small differences	<a href="#">A18-5-2</a>	
R.12: Immediately give the result of an explicit resource allocation to a manager object.	2 - Small differences	<a href="#">A18-5-2</a>	
R.13: Perform at most one explicit resource allocation in a single expression statement.	5 - Not yet analyzed	-	-
R.14: ??? array vs. pointer parameter.	2 - Small differences	<a href="#">M5-2-12</a>	
R.15: Always overload matched allocation/deallocation pairs.	5 - Not yet analyzed	-	-

R.20: Use <code>unique_ptr</code> or <code>shared_ptr</code> to represent ownership.	4 - Rejected		Design principle; There is no need for a new rule.
R.21: Prefer <code>unique_ptr</code> over <code>shared_ptr</code> unless you need to share ownership.	4 - Rejected		Design principle; There is no need for a new rule.
R.22: Use <code>make_shared()</code> to make <code>shared_ptr</code> s.	5 - Not yet analyzed	-	-
R.23: Use <code>make_unique()</code> to make <code>unique_ptr</code> s.	5 - Not yet analyzed	-	-
R.24: Use <code>std::weak_ptr</code> to break cycles of <code>shared_ptr</code> s.	4 - Rejected		Design principle; There is no need for a new rule.
R.30: Take smart pointers as parameters only to explicitly express lifetime semantics.	4 - Rejected		Design principle; There is no need for a new rule.
R.31: If you have non-std smart pointers, follow the basic pattern from <code>std</code> .	4 - Rejected		Design principle; There is no need for a new rule.
R.32: Take a <code>unique_ptr&lt;widget&gt;</code> parameter to express that a function assumes ownership of a widget.	4 - Rejected		Design principle; There is no need for a new rule.
R.33: Take a <code>unique_ptr&lt;widget&gt;&amp;</code> parameter to express that a function reseats the widget.	4 - Rejected		Design principle; There is no need for a new rule.
R.34: Take a <code>shared_ptr&lt;widget&gt;</code> parameter to express that a function is part owner.	4 - Rejected		Design principle; There is no need for a new rule.
R.35: Take a <code>shared_ptr&lt;widget&gt;&amp;</code> parameter to express that a function might reseat the shared pointer.	4 - Rejected		Design principle; There is no need for a new rule.-
R.36: Take a <code>const shared_ptr&lt;widget&gt;&amp;</code> parameter to express that it might retain a reference count to the object ???.	4 - Rejected		Design principle; There is no need for a new rule.

R.37: Do not pass a pointer or reference obtained from an aliased smart pointer.	4 - Rejected		Design principle; There is no need for a new rule.
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	<a href="#">M3-4-1</a>	
ES.6: Declare names in for-statement initializers and conditions to limit scope.	2 - Small differences	<a href="#">M3-4-1</a>	As an exception from the <a href="#">A7-1-7</a> , 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	<a href="#">M2-10-1</a>	
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	<a href="#">A7-1-7</a>	
ES.11: Use auto to avoid redundant repetition of type names.	3 - Significant differences	<a href="#">A7-1-5</a>	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	<a href="#">A2-11-1</a>	
ES.20: Always initialize an object.	2 - Small differences	<a href="#">M8-5-1</a>	
ES.21: Don't introduce a variable (or constant) before you need to use it.	2 - Small differences	<a href="#">M3-4-1</a>	

ES.22: Don't declare a variable until you have a value to initialize it with.	2 - Small differences	<a href="#">M3-4-1</a> , <a href="#">M8-5-1</a>	
ES.23: Prefer the {} initializer syntax.	2 - Small differences	<a href="#">A8-5-2</a>	
ES.24: Use a <code>unique_ptr&lt;T&gt;</code> to hold pointers.	3 - Significant differences	<a href="#">A18-5-2</a> , <a href="#">A15-1-4</a> , <a href="#">A18-1-3</a>	AUTOSAR C++ Coding Guidelines does not force a programmer to use <code>std::unique_ptr</code> , it is just highly recommended within examples and rationales.
ES.25: Declare an object <code>const</code> or <code>constexpr</code> unless you want to modify its value later on.	2 - Small differences	<a href="#">A7-1-1</a>	
ES.26: Don't use a variable for two unrelated purposes.	4 - Rejected		Implementation principle; There is no need for a new rule.
ES.27: Use <code>std::array</code> or <code>stack_array</code> for arrays on the stack.	3 - Significant differences	<a href="#">A18-1-1</a>	C-style arrays shall not be used, and it is recommended to use <code>std::array</code> instead.
ES.28: Use lambdas for complex initialization, especially of <code>const</code> variables.	5 - Not yet analyzed	-	-
ES.30: Don't use macros for program text manipulation.	2 - Small differences	<a href="#">A16-0-1</a>	Usage of macros is prohibited.
ES.31: Don't use macros for constants or "functions".	2 - Small differences	<a href="#">A16-0-1</a>	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	<a href="#">M2-10-1</a>	
ES.34: Don't define a (C-style) variadic function.	2 - Small differences	<a href="#">A8-4-1</a>	

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: <a href="#">A6-4-1</a> .
ES.71: Prefer a range-for-statement to a for-statement when there is a choice.	3 - Significant differences	<a href="#">A6-5-1</a>	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.	5 - Not yet analyzed	-	-
ES.73: Prefer a while-statement to a for-statement when there is no obvious loop variable.	3 - Significant differences	<a href="#">A6-5-2</a>	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	<a href="#">M3-4-1</a>	It is required that each identifier is defined in a block that minimizes its visibility.
ES.75: Avoid do-statements.	5 - Not yet analyzed	-	-
ES.76: Avoid goto.	2 - Small differences	<a href="#">A6-6-1</a>	
ES.78: Always end a non-empty case with a break.	2 - Small differences	<a href="#">M6-4-5</a>	
ES.85: Make empty statements visible.	2 - Small differences	<a href="#">M6-3-1</a> , <a href="#">M6-4-1</a> , <a href="#">M6-4-1</a>	
ES.86: Avoid modifying loop control variables inside the body of raw for-loops.	2 - Small differences	<a href="#">M6-5-3</a>	
ES.40: Avoid complicated expressions.	4 - Rejected		
ES.41: If in doubt about operator precedence, parenthesize.	5 - Not yet analyzed	-	-
ES.42: Keep use of pointers simple and straightforward.	4 - Rejected		
ES.43: Avoid expressions with undefined order of evaluation.	2 - Small differences	<a href="#">A5-0-1</a>	
ES.44: Don't depend on order of evaluation of function arguments.	2 - Small differences	<a href="#">A5-0-1</a>	
ES.45: Avoid "magic constants"; use symbolic constants.	2 - Small differences	<a href="#">A5-1-1</a>	
ES.46: Avoid lossy (narrowing, truncating) arithmetic conversions.	2 - Small differences	<a href="#">A4-7-1</a> , <a href="#">M5-0-6</a>	
ES.47: Use nullptr rather than 0 or NULL.	2 - Small differences	<a href="#">A4-10-1</a>	

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.	5 - Not yet analyzed	-	-
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.100: Don't mix signed and unsigned arithmetic.	5 - Not yet analyzed	-	-
ES.101: Use unsigned types for bit manipulation.	2 - Small differences	M5-0-21	
ES.102: Use signed types for arithmetic.	5 - Not yet analyzed	-	-
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	A5-5-1	
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.19: Access memory predictably.	5 - Not yet analyzed	-	-
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.	55 - Not yet analyzed		The "Concurrency and Parallelism" chapter is not yet covered, this will be addressed in future.
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 <code>gsl::detached_thread</code> over <code>std::thread</code> if you plan to <code>detach()</code> .	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
CP.27: Use plain <code>std::thread</code> 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 <code>detach()</code> 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- <code>rai_threads</code> .	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 <code>shared_ptr</code> .	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.	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
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.	5 - Not yet analyzed		The "Concurrency and Parallelism" chapter is not yet covered, this will be addressed in future.
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	<a href="#">A15-0-1</a>	
E.3: Use exceptions for error handling only.	2 - Small differences	<a href="#">A15-0-1</a>	

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	<a href="#">A15-2-2</a>	
E.6: Use RAll 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	<a href="#">A15-4-4</a>	
E.13: Never throw while being the direct owner of an object.	3 - Significant differences	<a href="#">A15-1-4</a>	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	<a href="#">A15-1-1</a>	It is required that user-defined exceptions inherit from std::exception class.
E.15: Catch exceptions from a hierarchy by reference.	2 - Small differences	<a href="#">A15-3-5</a>	
E.16: Destructors, deallocation, and swap must never fail.	2 - Small differences	<a href="#">A15-5-1</a>	

E.17: Don't try to catch every exception in every function.	2 - Small differences	<a href="#">A15-3-1</a> , <a href="#">A15-3-2</a>	AUTOSAR C++ Coding Guidelines introduces checked and unchecked exceptions. Whether they should be propagated or caught, It depends on the type of an exception.
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 <a href="#">A15-1-4</a> .
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: <a href="#">M0-3-2</a> .
E.28: Avoid error handling based on global state (e.g. errno).	2 - Small differences	<a href="#">M19-3-1</a>	
Con.1: By default, make objects immutable.	5 - Not yet analyzed	-	-
Con.2: By default, make member functions const.	2 - Small differences	<a href="#">M9-3-3</a>	

Con.3: By default, pass pointers and references to consts.	5 - Not yet analyzed	-	-
Con.4: Use const to define objects with values that do not change after construction.	5 - Not yet analyzed	-	-
Con.5: Use constexpr for values that can be computed at compile time.	2 - Small differences	<a href="#">A7-1-2</a>	
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.	5 - Not yet analyzed	-	-
T.43: Prefer using over typedef for defining aliases.	2 - Small differences	<a href="#">A7-1-6</a>	
T.44: Use function templates to deduce class template argument types (where feasible).	5 - Not yet analyzed	-	-

T.46: Require template arguments to be at least Regular or SemiRegular.	5 - Not yet analyzed	-	-
T.47: Avoid highly visible unconstrained templates with common names.	5 - Not yet analyzed	-	-
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.	5 - Not yet analyzed	-	-
T.60: Minimize a template's context dependencies.	5 - Not yet analyzed	-	-
T.61: Do not over-parameterize members (SCARY).	2 - Small differences	<a href="#">A14-1-1</a> , <a href="#">A14-7-1</a>	
T.62: Place non-dependent class template members in a non-templated base class.	5 - Not yet analyzed	-	-
T.65: Use tag dispatch to provide alternative implementations of a function.	5 - Not yet analyzed	-	-
T.68: Use < > rather than () within templates to avoid ambiguities.	2 - Small differences	<a href="#">A8-5-2</a>	
T.69: Inside a template, don't make an unqualified nonmember function call unless you intend it to be a customization point.	5 - Not yet analyzed	-	-
T.80: Do not naively templatize a class hierarchy.	5 - Not yet analyzed	-	-
T.81: Do not mix hierarchies and arrays.	5 - Not yet analyzed	-	-
T.83: Do not declare a member function template virtual.	5 - Not yet analyzed	-	-
T.84: Use a non-template core implementation to provide an ABI-stable interface.	5 - Not yet analyzed	-	-
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	<a href="#">A8-4-1</a>	
T.103: Don't use variadic templates for homogeneous argument lists.	5 - Not yet analyzed	-	-
T.120: Use template metaprogramming only when you really need to.	5 - Not yet analyzed	-	-
T.121: Use template metaprogramming primarily to emulate concepts.	5 - Not yet analyzed	-	-
T.122: Use templates (usually template aliases) to compute types at compile time.	5 - Not yet analyzed	-	-
T.123: Use constexpr functions to compute values at compile time.	5 - Not yet analyzed	-	-



T.124: Prefer to use standard-library TMP facilities.	5 - Not yet analyzed	-	-
T.125: If you need to go beyond the standard-library TMP facilities, use an existing library.	5 - Not yet analyzed	-	-
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.	5 - Not yet analyzed	-	-
T.142?: Use template variables to simplify notation.	5 - Not yet analyzed	-	-
T.143: Don't write unintentionally nongeneric code.	5 - Not yet analyzed	-	-
T.144: Don't specialize function templates.	2 - Small differences	<a href="#">M14-8-1</a> , <a href="#">A14-8-1</a>	
T.150: Check that a class matches a concept using <code>static_assert</code> .	2 - Small differences	<a href="#">A14-1-1</a>	
CPL.1: Prefer C++ to C.	2 - Small differences	<a href="#">A17-1-1</a> , <a href="#">A18-0-1</a>	
CPL.2: If you must use C, use the common subset of C and C++, and compile the C code as C++.	5 - Not yet analyzed	-	-
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 <code>.cpp</code> suffix for code files and <code>.h</code> for interface files if your project doesn't already follow another convention.	3 - Significant differences	<a href="#">A3-1-2</a> , <a href="#">A3-1-3</a>	For header file names, AUTOSAR C++ Coding Guidelines allows either <code>.h</code> , <code>.hpp</code> or <code>.hxx</code> extension.
SF.2: A <code>.h</code> file may not contain object definitions or non-inline function definitions.	2 - Small differences	<a href="#">A3-1-1</a>	
SF.3: Use <code>.h</code> files for all declarations used in multiple source files.	2 - Small differences	<a href="#">M3-2-2</a> , <a href="#">A3-3-1</a>	
SF.4: Include <code>.h</code> files before other declarations in a file.	2 - Small differences	<a href="#">M16-0-1</a>	
SF.5: A <code>.cpp</code> file must include the <code>.h</code> file(s) that defines its interface.	5 - Not yet analyzed	-	-
SF.7: Don't write using namespace in a header file.	2 - Small differences	<a href="#">M7-3-6</a>	
SF.8: Use <code>#include</code> guards for all <code>.h</code> files.	2 - Small differences	<a href="#">M16-2-3</a>	
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	<a href="#">M7-3-3</a>	
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.con.1: Prefer using STL array or vector instead of a C array.	2 - Small differences	<a href="#">A18-1-1</a>	
SL.con.2: Prefer using STL vector by default unless you have a reason to use a different container.	5 - Not yet analyzed		Not yet analyzed, this rule will be address later.
SL.io.50: Avoid endl.	5 - Not yet analyzed	-	-
Type.1: Don't use reinterpret_cast.	2 - Small differences	<a href="#">A5-2-4</a>	
Type.2: Don't use static_cast downcasts. Use dynamic_cast instead.	2 - Small differences	<a href="#">M5-2-2</a>	
Type.3: Don't use const_cast to cast away const (i.e., at all).	2 - Small differences	<a href="#">A5-2-3</a>	
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	<a href="#">A5-2-2</a>	
Type.5: Don't use a variable before it has been initialized.	2 - Small differences	<a href="#">M8-5-1</a>	
Type.6: Always initialize a member variable.	2 - Small differences	<a href="#">M8-5-1</a>	
Type.7: Avoid accessing members of raw unions. Prefer variant instead.	2 - Small differences	<a href="#">M9-5-1</a>	
Type.8: Avoid reading from varargs or passing vararg arguments. Prefer variadic template parameters instead.	2 - Small differences	<a href="#">A8-4-1</a>	
Bounds.1: Don't use pointer arithmetic. Use span instead.	2 - Small differences	<a href="#">M5-0-15</a>	
Bounds.2: Only index into arrays using constant expressions.	2 - Small differences	<a href="#">A5-2-5</a>	
Bounds.3: No array-to-pointer decay.	2 - Small differences	<a href="#">M5-2-12</a>	
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	<a href="#">M2-10-1</a>	
NL.20: Don't place two statements on the same line.	3 - Significant differences	<a href="#">A7-1-7</a>	It is required for declarations only.

NL.21: Declare one name (only) per declaration.	2 - Small differences	<a href="#">A7-1-7</a>	
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 A.5: ISOCPP**

## B 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.

<p>One definition rule</p>	<p>The rule states that:</p> <ul style="list-style-type: none"> <li>• 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.</li> <li>• There shall be one and only one definition of every non-inline function or variable that is odr-used in the entire program.</li> <li>• An inline function definition is required in every translation unit where it is odr-used.</li> <li>• 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.</li> <li>• 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:             <ul style="list-style-type: none"> <li>– each definition consists of the same sequence of tokens (typically, appears in the same header file)</li> <li>– 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.</li> <li>– overloaded operators, including conversion, allocation, and deallocation functions refer to the same function from each definition (unless referring to one defined within the definition)</li> <li>– the language linkage is the same (e.g. the include file isn't inside an extern "C" block)</li> <li>– the three rules above apply to every default argument used in each definition</li> <li>– 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</li> <li>– 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</li> </ul> </li> </ul> <p>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.</p>
<p>ODR-use</p>	<p>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.</p>

<p>POD Type</p>	<p>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.</p>
<p>Trivially Copyable Class</p>	<p>A class that:</p> <ul style="list-style-type: none"> <li>• has no non-trivial default constructors</li> <li>• has no non-trivial copy and move constructors</li> <li>• has no non-trivial copy and move assignment operators</li> <li>• has no virtual functions</li> <li>• has no virtual base classes</li> <li>• has a trivial destructor</li> <li>• has a default constructor</li> </ul>
<p>Standard-Layout Class</p>	<p>A class that:</p> <ul style="list-style-type: none"> <li>• has no non-static data members of type non-standard-layout class (or array of such types) or reference</li> <li>• has no virtual functions and no virtual base classes</li> <li>• has the same access control for all non-static data members</li> <li>• has no non-standard-layout base classes</li> <li>• has at most one base class subobject of any given type</li> <li>• has all non-static data members and bit-fields in the class and its base classes first declared in the same class</li> <li>• has no element of the set <math>M(X)</math> of types as a base class where <math>M(X)</math> is defined as follows: <ul style="list-style-type: none"> <li>– If <math>X</math> is a non-union class type, the set <math>M(X)</math> is empty if <math>X</math> has no (possibly inherited) non-static data members; otherwise, it consists of the type of the first non-static data member of <math>X</math> (where said member may be an anonymous union), <math>X0</math>, and the elements of <math>M(X0)</math>.</li> <li>– If <math>X</math> is a union type, the set <math>M(X)</math> is the union of all <math>M(U_i)</math> and the set containing all <math>U_i</math>, where each <math>U_i</math> is the type of the <math>i</math>-th non-static data member of <math>X</math>.</li> <li>– If <math>X</math> is a non-class type, the set <math>M(X)</math> is empty.</li> </ul> </li> </ul>



Dataflow Anomaly	<p>The state of a variable at a point in a program can be described using the following terms:</p> <ul style="list-style-type: none"> <li>• Undefined (U): The value of the variable is indeterminate.</li> <li>• Referenced (R): The variable is used in some way (e.g. in an expression).</li> <li>• Defined (D): The variable is explicitly initialized or assigned a value.</li> </ul> <p>Given the above, the following dataflow anomalies can be defined:</p> <ul style="list-style-type: none"> <li>• UR dataflow anomaly: Variable not assigned a value before the specified use.</li> <li>• DU dataflow anomaly: Variable is assigned a value that is never subsequently used.</li> <li>• DD dataflow anomaly: Variable is assigned a value twice with no intermediate use.</li> </ul>
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 overridden 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	<p>An interface class is a class which has following properties:</p> <ul style="list-style-type: none"> <li>• if there are any, all methods are public pure virtual</li> <li>• if there are any, all data members are public static constexpr</li> </ul>
Extended precision format	<p>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.</p> <p>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.</p>
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		A scalar type is a type that provides built-in functionality for the addition operator without overloads. Following types are scalar types: <ul style="list-style-type: none"> <li>• integral types</li> <li>• floating point types</li> <li>• pointers</li> <li>• scoped and unscoped enumerations</li> <li>• <code>std::nullptr_t</code></li> </ul>
glvalue		A glvalue is an lvalue or an xvalue.
rvalue		An rvalue is an xvalue or a prvalue.
lvalue		An lvalue represents an object that occupies identifiable location in memory.
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 rvalue that is not an xvalue, e.g. a literal (such as <code>true</code> , <code>nullptr</code> , etc.) or the result of calling a function whose return type is not a reference is a prvalue.
Implicitly-defined constructor	default	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 constructor	copy	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 constructor	move	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 assignment operator	copy	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 assignment operator	move	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.
-------------------------------	--

**Table B.1: table:acronyms**