<table>
<thead>
<tr>
<th>Document Title</th>
<th>Guidelines for the use of the C++14 language in critical and safety-related systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document Owner</td>
<td>AUTOSAR</td>
</tr>
<tr>
<td>Document Responsibility</td>
<td>AUTOSAR</td>
</tr>
<tr>
<td>Document Identification No</td>
<td>839</td>
</tr>
<tr>
<td>Document Status</td>
<td>Final</td>
</tr>
<tr>
<td>Part of AUTOSAR Standard</td>
<td>Adaptive Platform</td>
</tr>
<tr>
<td>Part of Standard Release</td>
<td>17-03</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Date</th>
<th>Release</th>
<th>Changed by</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017-03-31</td>
<td>17-03</td>
<td>AUTOSAR Release Management</td>
<td>Initial release</td>
</tr>
</tbody>
</table>
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
6.10.2 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:

2. High Integrity C++ Coding Standard Version 4.0 [8]
3. CERT C++ Coding Standard [9]

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.

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

### 6.8 Declarators

<table>
<thead>
<tr>
<th>Feature</th>
<th>C++11</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default arguments</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Variadic arguments</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>List initialization</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

### 6.9 Classes

<table>
<thead>
<tr>
<th>Feature</th>
<th>C++11</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unions</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Bit-fields</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

### 6.10 Derived Classes

<table>
<thead>
<tr>
<th>Feature</th>
<th>C++11</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Multiple inheritance</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Virtual functions</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>override specifier</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>final specifier</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

### 6.11 Member Access Control

<table>
<thead>
<tr>
<th>Feature</th>
<th>C++11</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>friend declaration</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

### 6.12 Special Member Functions

<table>
<thead>
<tr>
<th>Feature</th>
<th>C++11</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defaulted and deleted functions</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Delegating constructors</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Member initializer lists</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Non-static data member initializer</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>explicit specifier</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Move semantics</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

### 6.13 Overloading

<table>
<thead>
<tr>
<th>Feature</th>
<th>C++11</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>User-defined literals</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Digit sequences separators ’</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

### 6.14 Templates

<table>
<thead>
<tr>
<th>Feature</th>
<th>C++11</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variadic templates</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Variable templates</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

### 6.15 Exception Handling

<table>
<thead>
<tr>
<th>Feature</th>
<th>C++11</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exceptions</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Function-try-blocks</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Dynamic exception specification</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>noexcept specifier</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

### 6.16 Preprocessing Directives

<table>
<thead>
<tr>
<th>Feature</th>
<th>C++11</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static assertion</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Implementation defined behavior control (#pragma directive)</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

**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

```cpp
//% $Id: A0-1-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
#include <array>
#include <cstdint>

std::uint8_t fn1(std::uint8_t param) noexcept
{
    std::int32_t x{0}; // Non-compliant - DU data flow anomaly; Variable defined,
                      // but not used
    if (param > 0)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

std::int32_t fn2() noexcept
{
    std::int8_t x{10U}; // Compliant - variable defined and will be used
    std::int8_t y{20U}; // Compliant - variable defined and will be used
    std::int16_t result = x + y; // x and y variables used
    x = 0; // Non-compliant - DU data flow anomaly; Variable defined, but x is
            // not subsequently used and goes out of scope
    y = 0; // Non-compliant - DU data flow anomaly; Variable defined, but y is
            // not subsequently used and goes out of scope
    return result;
}

std::int32_t fn3(std::int32_t param) noexcept
{
    std::int32_t x{param + 1}; // Compliant - variable defined, and will be used in
                                // one of the branches
                             // However, scope of x variable could be reduced
    if (param > 20)
    {
        return x;
    }
    return 0;
}

std::int32_t fn4(std::int32_t param) noexcept
```
std::int32_t x{param + 1}; // Compliant - variable defined, and will be used in some of the branches
if (param > 20)
{
    return x + 1;
}
else if (param > 10)
{
    return x;
}
else
{
    return 0;
}
}

void fn5() noexcept
{
    std::array<std::int32_t, 100> arr{};
    arr.fill(1);
    constexpr std::uint8_t limit{100U};
    std::int8_t x{0};
    for (std::uint8_t i{0U}; i < limit; ++i) // Compliant by exception - on the final loop, value of i defined will not be used
    {
        arr[i] = arr[x];
        ++x; // Non-compliant - DU data flow anomaly on the final loop, value defined and not used
    }
}

See also

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

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

**Rationale**

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

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

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

Example

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

See also

- MISRA C++ 2008 [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 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**

```cpp
1 //% $Id: A0-1-3.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
2 #include <cstdint>
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     protected:
24     void m1ProtectedImpl(std::int32_t j) // Compliant, never used but declared
25         // as protected
26         {
27             x = j;
28         }
29 ```

private:
    std::int32_t x;

void m1PrivateImpl(
    std::int32_t j) // Non-compliant, private method never used
{
    x = j;
}

int main(int, char**)
{
    f1();
    C c;
    c.m1(1);
    return 0;
}

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
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**

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

See also
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].

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

```cpp
//% $Id: A1-1-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
#include <cstdint>
#include <stdexcept>

void f(std::int32_t i)
{
    std::int32_t* a = nullptr;
    // __try // Non-compliant - __try is a part of Visual Studio extension
    try // Compliant - try keyword is a part of C++ Language Standard
    {
        a = new std::int32_t[i];
        // ...
    }
    // __finally // Non-compliant - __finally is a part of Visual Studio
    // extension
    catch (std::exception&)
    {
        delete[] a;
        a = nullptr;
    }
}
```

See also


- 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 <cstdint>
3 void fn() noexcept
4 {
5     std::int32_t sum = 0; // Compliant
6     // std::int32_t £_value = 10; // Non-compliant
7     // sum += £_value; // Non-compliant
8     // Variable sum stores £ pounds // Non-compliant
9 }

See also
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

```cpp
#include <iostream>

void fn1()
{
    std::cout << "Enter date ??/??/??"; // Non-compliant, ??/??/?? becomes \??
    // after trigraph translation
}

void fn2()
{
    std::cout << "Enter date dd/mm/yy"; // Compliant
}
```

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
class A
{
public:
    void f2() {}
};

// void fn1(A* a<:10:>) // Non-compliant
// <%
// a<:0:>->f2();
// %>
void fn2(A* a[10]) // Compliant, equivalent to the above
{
    a[0]->f2();
}

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

```cpp
#include <cstdint>

void fn() noexcept
{
    std::int8_t idx = 0;
    // Incrementing idx before the loop starts // Requirement X.X.X \ 
    ++idx; // Non-compliant - ++idx was unexpectedly commented-out because of \ character occurrence in the end of C++ comment
   constexpr std::int8_t limit = 10;
    for (; idx <= limit; ++idx)
    {
        // ...
    }
```
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**

```cpp
// $Id: A2-8-2.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
#include <cstdint>

void fn1() noexcept
{
    std::int32_t i = 0;
    // /*
    //   * ++i; /* incrementing the variable i */
    // */ // Non-compliant - C-style comments nesting is not supported,
    // compilation error
    for (; i < 10; ++i)
    {
        // ...
    }
}

void fn2() noexcept
{
    std::int32_t i = 0;
    // ++i; // Incrementing the variable i // Non-compliant - code should not
    // be commented-out
    for (; i < 10; ++i)
    {
        // ...
    }
}

void fn3() noexcept
{
    std::int32_t i = 0;
```
++i; // Incrementing the variable i using ++i syntax // Compliant - code
// is not commented-out, but ++i occurs in a
// comment too
for (; i < 10; ++i)
{
    // ...
}

See also

- MISRA C++ 2008 [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

```cpp
void f1(std::int32_t) noexcept; // Non-compliant documentation
std::int32_t f2(std::int16_t input1,
std::int32_t input2); // Non-compliant documentation

/// @brief Function description
/// @param input1 input1 parameter description
/// @param input2 input2 parameter description
/// @throw std::runtime_error conditions to runtime_error occur
///
/// @return return value description
std::int32_t f3(
```
Guidelines for the use of the C++14 language in critical and safety-related systems

AUTOSAR AP Release 17-03

---

```cpp
std::int16_t input1,
std::int16_t input2) noexcept(false); // Compliant documentation

/// @brief Class responsibility
class C // Compliant documentation
{
  public:
    /// @brief Constructor description
    /// @param input1 input1 parameter description
    /// @param input2 input2 parameter description
    C(std::int32_t input1, float input2) : x(input1), y(input2) {}

  private:
    /// @brief Data member description
    std::int x;
    /// @brief Data member description
    float y;
};

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
#include <cstdint>

void fn(bool b1, bool b2) noexcept
{
  // Introduces x of type std::int16_t // Compliant
  std::int16_t x = 0;
  // std::int16_t x = 0; // commented out temporarily, type too small //
  // Compliant
  if (b1)
  {
    if (b2)
    
```
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
```cpp
//% $Id: A2-11-1.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
#include <cstdint>
std::int32_t sum = 0;
namespace
{
  std::int32_t sum; // Non-compliant, hides sum in outer scope
}
class C1
{
  std::int32_t sum; // Compliant, does not hide sum in outer scope
};
namespace n1
{
  std::int32_t sum; // Compliant, does not hide sum in outer scope
}
void f1(std::int32_t);
void f2()
{
  std::int32_t max = 5;
  for (std::int32_t idx = 0; idx < max;
```
```cpp
++idx) // Non-compliant, hides idx in outer scope
{
        for (std::int32_t idx = 0; idx < max;
            ++idx) // Non-compliant, hides idx in outer scope
                {
                        }
                }

            void f3()
            {
                std::int32_t i = 0;
                std::int32_t j = 0;
                auto lambda = [i] () {
                    std::int32_t j =
                    10; // Compliant - j was not captured, so it does not hide
                        // j in outer scope
                    return i + j;
                };
            }

See also

- MISRA C++ 2008 [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

```
using func = void (*)(std::int32_t, std::int32_t);

void f1()
{
    using func = void (*)(void); // Non-compliant, reuses func identifier declared in the same namespace
}

namespace n2
{
    using func = void (*)(std::int32_t, std::int32_t); // Compliant, reuses func identifier but in another namespace
}

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**

```cpp
class Type
{
};
// struct Type { }; // Non-compliant, Type name reused
// 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.
Guidelines for the use of the C++14 language in critical and safety-related systems

AUTOSAR AP Release 17-03

- 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**
```cpp
// f1.cpp
namespace ns1
{
  static std::int32_t globalvariable = 0;
}

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

namespace ns2
{
  static std::int32_t globalvariable = 0; // Compliant - identifier reused, but in another namespace
}

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

**See also**
- MISRA C++ 2008 [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
```
#include <cstdint>

// f1.cpp
namespace n_s1
{
  static std::int32_t globalvariable = 0;
}

static std::int32_t filevariable = 5; // Compliant - identifier not reused
static void globalfunction();

// f2.cpp
namespace n_s1
{
  // static std::int32_t globalvariable = 0; // Non-compliant - identifier reused
  static std::int16_t modulevariable = 10; // Compliant - identifier not reused
}
namespace n_s2
{
  static std::int16_t modulevariable2 = 20;
}

static void globalfunction(); // Non-compliant - identifier reused
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**

```cpp

```char16_t``` nArray[] =
  u"Hello"
  u"World"; // Compliant, "u" stands for char16_t type

```char32_t``` nArray2[] =
  U"Hello"
  U"World"; // Compliant, "U" stands for char32_t type

```wchar_t``` wArray[] =
  L"Hello"
  L"World"; // Compliant, "L" stands for wchar_t type - violates A2-14-3
  // rule.

```wchar_t``` mixed1[] =
  "Hello"
  L"World"; // Compliant

```char32_t``` mixed2[] =
  "Hello"
  U"World"; // Compliant

```char16_t``` mixed3[] =
  "Hello"
  u"World"; // Compliant

// wchar_t mixed1[] = u"Hello" L"World"; // Non-compliant - compilation error
// 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**
```cpp
//% $Id: A2-14-3.cpp 266557 2017-02-07 13:08:19Z piotr.tanski $
char16_t string1[] = u"ABC"; // Compliant
char32_t string2[] = U"DEF"; // Compliant
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**
```cpp
//% $Id: A3-1-1.hpp 271715 2017-03-23 10:13:51Z piotr.tanski $ 
#include <cstdint>
void f1(); // Compliant
extern void f2(); // Compliant
void f3()
{
} // Non-compliant
static inline void f4()
{
}
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

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

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

See also

- MISRA C++ 2008 [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

```cpp
//% $Id: A3-3-1.hpp 271715 2017-03-23 10:13:51Z piotr.tanski $
#include <cstdint>
extern std::int32_t a1;
extern void f4();
namespace n
{
    void f2();
    std::int32_t a5; // Compliant, external linkage
}

//% $Id: A3-3-1.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
#include "A3-3-1.hpp"
std::int32_t a1 = 0; // Compliant, external linkage
std::int32_t a2 = 0; // Non-compliant, static keyword not used
```
static std::int32_t a3 = 0;  // Compliant, internal linkage
namespace
{
    std::int32_t a4 = 0;  // Compliant by exception
    void f1()   // Compliant by exception
    {
    }
}
namespace n
{
    void f2()  // Compliant, external linkage
    {
    }
    std::int32_t a6 = 0;  // Non-compliant, external linkage
}
extern std::int32_t a7;  // Non-compliant, extern object declared in .cpp file
static void f3()   // Compliant, static keyword used
{
}
void f4()  // Compliant, external linkage
{
    a1 = 1;
    a2 = 1;
    a3 = 1;
    a4 = 1;
    n::a5 = 1;
    n::a6 = 1;
    a7 = 1;
}
void f5()  // Non-compliant, static keyword not used
{
    a1 = 2;
    a2 = 2;
    a3 = 2;
    a4 = 2;
    n::a5 = 2;
    n::a6 = 2;
    a7 = 2;
}
int main(int, char**)  // Compliant by exception
{
    f1();
    n::f2();
    f3();
    f4();
    f5();
}

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

```cpp
// $Id: A3-3-2.cpp 271927 2017-03-24 12:01:35Z piotr.tanski $
#include <cstdint>
#include <limits>
#include <string>
class A // Non-POD type
{
    public:
        static std::uint8_t instanceId; // Compliant - static variable of POD type
        static float const pi; // Compliant - static variable of POD type
        static std::string const separator; // Non-compliant - static variable of nonPOD type
    
    // Implementation
};
std::uint8_t A::instanceId = 0;
float const A::pi = 3.14159265359;
std::string const A::separator = "==========";

class B
{
    public:
        // Implementation
```
```cpp
private:
    static A a; // Non-compliant - static variable of non-POD type
};

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

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

    A instance{}; // Non-compliant - static variable of non-POD type

class C
    constexpr C; // Compliant by exception - constexpr static variable of non-POD type
} // namespace

void fn() noexcept
{
    static A a{}; // Non-compliant
    static std::int32_t counter{0}; // Compliant
}

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

    private:
        static B* instance; // Compliant - static variable of non-POD type, because it is a raw pointer
};

B* D::instance = nullptr;

See also

- HIC++ v4.0 [8]: 3.3.1 Do not use variables with static storage duration.
```
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**

```cpp
//% $Id: A3-9-1.cpp 271389 2017-03-21 14:41:05Z piotr.tanski $
#include <cstdint>
void f()
{
    std::int32_t i1 = 5; // Compliant
    int i2 = 10; // Non-compliant
    std::int64_t i3 = 250; // Compliant
    long int i4 = 50; // Non-compliant
    std::int8_t i5 = 16; // Compliant
    char i6 = 23; // Non-compliant
}
```

**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 `[ ]`,

---

52 of 371 Document ID 839: AUTOSAR_RS_CPP14Guidelines

--- AUTOSAR CONFIDENTIAL ---
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

```cpp
// $Id: A4-5-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
#include <cstdint>
enum Colour : std::uint8_t
{
    Red,
    Green,
    Blue,
    ColoursCount
};
void f1() noexcept(false)
{
    Colour colour = Red;
    if (colour == Green) // Compliant
    {
    }
    if (colour == (Red + Blue)) // Non-compliant
    {
    }
    if (colour < ColoursCount) // Compliant
    {
    }
}
enum class Car : std::uint8_t
{
    Model1,
    Model2,
    Model3,
    ModelsCount
};
void f2() noexcept(false)
{
    Car car = Car::Model1;
```
if (car != Car::Model2) // Compliant
{
}

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

// if (car == (Car::Model1 + Car::Model2)) // Non-compliant -
// operator+ not provided for Car enum class, compilation error
//}}
if (car < Car::ModelsCount) // Compliant
{
}

Car operator+(Car lhs, Car rhs)
{
    return Car::Model3;
}

void f3() noexcept(false)
{
    Car car = Car::Model3;
    if (car == (Car::Model1 + Car::Model2)) // Non-compliant - overloaded
        // operator+ provided, no
        // compilation error
{
}

enum Team : std::uint8_t
{
    TeamMember1 = 0,
    TeamMember2 = 1,
    TeamMember3 = 2,
    TeamMember4 = 3,
    TeamMembersStart = TeamMember1,
    TeamMembersEnd = TeamMember2,
    TeamMembersCount = 4
};
void f4(const char* teamMember)
{
    // Implementation
}
void f5()
{
    const char* team[TeamMembersCount]; // Compliant
    // ...
    f4(team[TeamMember2]); // Compliant
}
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

```cpp
// $Id: A4-7-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
#include <cstdint>
#include <stdexcept>
std::int8_t fn1(std::int8_t x, std::int8_t y) noexcept
{
    return (x + y); // Non-compliant - may lead to overflow
}
std::int8_t fn2(std::int8_t x, std::int8_t y)
{
    if (x > 100 || y > 100) // Range check
    {
        throw std::logic_error("Preconditions check error");
    }
    return (x + y); // Compliant - ranges of x and y checked before the arithmetic operation
}
std::int16_t fn3(std::int8_t x, std::int8_t y) noexcept
{
    return (static_cast<std::int16_t>(x) + y); // Compliant - std::int16_t type is enough for this arithmetic operation
}
```
```cpp
std::uint8_t fn4(std::uint8_t x, std::uint8_t y) noexcept
{
    return (x * y); // Non-compliant - may lead to wrap-around
}

std::int8_t fn5(std::int16_t x)
{
    return static_cast<std::int8_t>(x); // Non-compliant - data loss
}

std::int8_t fn6(std::int16_t x)
{
    return x; // Non-compliant - data loss by implicit conversion
}

void f()
{
    std::int8_t x1 = fn1(5, 10); // Compliant - overflow will not occur for these values
    std::int8_t x2 = fn1(250, 250); // Non-compliant - Overflow occurs
    try
    {
        std::int8_t x3 = fn2(250, 250); // Compliant - No overflow, range checks
        // inside fn2() function
    }
    catch (std::logic_error&)
    {
        // Handle an error
    }
    std::int16_t x4 = fn3(250, 250); // Compliant - No overflow, arithmetic
    // operation underlying type is wider than
    // std::int8_t
    std::uint8_t x5 = fn4(50, 10); // Non-compliant - Wrap-around occurs
    std::int8_t x6 = fn5(100); // Compliant - data loss will not occur
    std::int8_t x7 = fn5(300); // Non-compliant - Data loss occurs
    std::int8_t x8 = fn6(300); // Non-compliant - Data loss occurs
    std::int8_t x9 = 150;
    std::int16_t x10 = static_cast<std::int16_t>(x9 + x9); // Non-compliant
    x10 = x9 + x9; // Non-compliant
    x10 = static_cast<std::int16_t>(x9) + x9; // Compliant
    std::int8_t x11 = x9 << 5; // Non-compliant
    ++x12; // Non-compliant
    ++x13; // Non-compliant
}
```

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**

```cpp
//% $Id: A4-10-1.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
#include <cstddef>
#include <cstdint>

void f1(std::int32_t);
void f2(std::int32_t*);
void f3()
{
    f1(0);      // Compliant
    f1(NULL);   // Non-compliant - NULL used as an integer,
    // compilable
    // f1(nullptr); // Non-compliant - nullptr used as an integer
    // compilation error
    f2(0);      // Non-compliant - 0 used as the null pointer constant
    f2(NULL);   // Non-compliant - NULL used as the null pointer constant
    f2(nullptr); // Compliant
}
void f4(std::int32_t*);
```
template <class F, class A>

void f5(F f, A a)
{
    f4(a);
}

void f6()
{
    // f5(f4, NULL); // Non-compliant - function f4(std::int32_t) not declared
    f5(f4, nullptr); // Compliant
}

See also
• HIC++ v4.0 [8]: 2.5.3 Use nullptr for 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 <cstdint>
3 #include <stack>
4 // The following notes give some guidance on how dependence on order of
5 // evaluation may occur, and therefore may assist in adopting the rule.
// 1) Increment or decrement operators
// As an example of what can go wrong, consider
void f1(std::uint8_t (&arr)[10], std::uint8_t idx) noexcept(false)
{
    std::uint16_t x = arr[idx] + idx++;
}
// This will give different results depending on whether arr[idx] is evaluated
// before idx++ or vice versa. The problem could be avoided by putting the
// increment operation in a separate statement. For example:
void f2(std::uint8_t (&arr)[10], std::uint8_t idx) noexcept(false)
{
    std::uint8_t x = arr[idx] + idx;
    idx++;
}

// 2) Function arguments
// The order of evaluation of function arguments is unspecified.
extern std::uint8_t func(std::uint8_t x, std::uint8_t y);
void f3() noexcept(false)
{
    std::uint8_t i = 0;
    std::uint8_t x = func(i++, i);
}
// This will give different results depending on which of the functions two
// parameters is evaluated first.

// 3) Function pointers
// If a function is called via a function pointer there shall be no
// dependence
// on the order in which function-designator and function arguments are
// evaluated.
struct S
{
    void taskStartFn(S* obj) noexcept(false);
};
void f4(S* p) noexcept(false)
{
    p->taskStartFn(p++);
}

// 4) Function calls
// Functions may have additional effects when they are called (e.g. modifying
// some global data). Dependence on order of evaluation could be avoided by
// invoking the function prior to the expression that uses it, making use of a
// temporary variable for the value. For example:
extern std::uint16_t g(std::uint8_t) noexcept(false);
extern std::uint16_t z(std::uint8_t) noexcept(false);
void f5(std::uint8_t a) noexcept(false)
{
    std::uint16_t x = g(a) + z(a);
// could be written as
void f6(std::uint8_t a) noexcept(false) {
    std::uint16_t x = g(a);
    x += z(a);
}

// As an example of what can go wrong, consider an expression to take two values
// off a stack, subtract the second from the first, and push the result back on
// the stack:
std::int32_t pop(std::stack<std::int32_t>& s) {
    std::int32_t ret = s.top();
    s.pop();
    return ret;
}

void f7(std::stack<std::int32_t>& s) {
    s.push(pop(s) - pop(s));
}

// This will give different results depending on which of the pop() function
// calls is evaluated first (because pop() has side effects).

// 5) Nested assignment statements
// Assignments nested within expressions cause additional side effects. The best
// way to avoid any possibility of this leading to a dependence on order of
// evaluation is not to embed assignments within expressions. For example, the
// following is not recommended:
void f8(std::int32_t& x) noexcept(false) {
    std::int32_t y = 4;
    x = y = y++; // It is undefined whether the final value of y is 4 or 5
}

// 6) Accessing a volatile
// The volatile type qualifier is provided in C++ to denote objects whose value
// can change independently of the execution of the program (for example an
// input register). If an object of volatile qualified type is accessed this may
// change its value. C++ compilers will not optimize out reads of a volatile. In
// addition, as far as a C++ program is concerned, a read of a volatile has a
// side effect (changing the value of the volatile). It will usually be
// necessary to access volatile data as part of an expression, which then means
// there may be dependence on order of evaluation. Where possible, though, it is
// recommended that volatiles only be accessed in simple assignment statements,
// such as the following:
void f9(std::uint16_t& x) noexcept(false) {
    volatile std::uint16_t v;
    // ...  
    x = v;
}
// The rule addresses the order of evaluation problem with side effects. Note
// that there may also be an issue with the number of times a sub-expression is
// evaluated, which is not covered by this rule. This can be a problem with
// function invocations where the function is implemented as a macro. For
// example, consider the following function-like macro and its invocation:
#define MAX(a, b) (((a) > (b)) ? (a) : (b))
// ...
void f10(std::uint32_t& i, std::uint32_t j)
{
    std::uint32_t z = MAX(i++, j);
}
// The definition evaluates the first parameter twice if a > b but only once if
// a = b. The macro invocation may thus increment i either once or twice,
// depending on the values of i and j.
// It should be noted that magnitude-dependent effects, such as those due to
// floating-point rounding, are also not addressed by this rule. Although
// order in which side effects occur is undefined, the result of an operation is
// otherwise well-defined and is controlled by the structure of the expression.
// In the following example, f1 and f2 are floating-point variables; F3, F4
// and
// F5 denote expressions with floating-point types.

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

See also

- MISRA C++ 2008 [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:

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

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

See MISRA C++ 2008 [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**

```cpp
1 // $Id: A5-0-2.cpp 271927 2017-03-24 12:01:35Z piotr.tanski $
2 #include <cstdint>
3 #include <memory>
```
extern std::int32_t* fn();
extern std::int32_t fn2();
extern bool fn3();
void f() noexcept(false)
{
    std::int32_t* ptr = nullptr;

    while ((ptr = fn()) != nullptr) // Compliant
    {
        // Code
    }

    // The following is a cumbersome but compliant example
    do
    {
        std::int32_t* ptr = fn();
        if (nullptr == ptr)
        {
            break;
        }
        // Code
    } while (true); // Compliant

    std::unique_ptr<std::int32_t> uptr;
    if (!uptr) // Compliant - std::unique_ptr defines an explicit conversion to
               // type bool.
    {
        // Code
    }

    while (std::int32_t length = fn2()) // Compliant by exception
    {
        // Code
    }

    while (bool flag = fn3()) // Compliant
    {
        // Code
    }

    if (std::int32_t* ptr = fn())
    { // Compliant by exception
    }

    if (std::int32_t length = fn2())
    { // Compliant by exception
    }

    if (bool flag = fn3())
    { // Compliant
    }

}
Guidelines for the use of the C++14 language in critical and safety-related systems

AUTOSAR AP Release 17-03

std::uint8_t u = 8;

if (u)            // Non-compliant
    ;

bool boolean1 = false;
bool boolean2 = true;

if (u && (boolean1 <= boolean2))        // Non-compliant
    ;

for (std::int32_t x = 10; x; --x)       // Non-compliant
    ;

}

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
```cpp
#include <cstdint>

using IntPtr = std::int8_t*;

struct S {
    std::int8_t* s1; // Compliant
    std::int8_t** s2; // Compliant
    std::int8_t*** s3; // Non-compliant
};

S* ps1; // Compliant
S** ps2; // Compliant
S*** ps3; // Non-compliant

std::int8_t** (*pfunc1)(); // Compliant
std::int8_t** (**pfunc2)(); // Compliant
std::int8_t** (***pfunc3)(); // Non-compliant
std::int8_t*** (**pfunc4)(); // Non-compliant

void fn(std::int8_t* par1, // Compliant
        std::int8_t** par2, // Compliant
        std::int8_t*** par3, // Non-compliant
        IntPtr* par4, // Compliant
        IntPtr* const* const par5 = nullptr; // Non-compliant
        std::int8_t* par6[], // Compliant
        std::int8_t** par7[]) // Non-compliant
{
    std::int8_t* ptr1; // Compliant
    std::int8_t** ptr2; // Compliant
    std::int8_t*** ptr3; // Non-compliant
    IntPtr* ptr4; // Compliant
    IntPtr* const* const ptr5 = nullptr; // Non-compliant
    std::int8_t* ptr6[10]; // Compliant
    std::int8_t** ptr7[10]; // Compliant
}
```

// Explanation of types
// 1) par1 and ptr1 are of type pointer to std::int8_t.
// 2) par2 and ptr2 are of type pointer to pointer to std::int8_t.
// 3) par3 and ptr3 are of type pointer to a pointer to a pointer
// to std::int8_t.
// This is three levels and is non-compliant.
// 4) par4 and ptr4 are expanded to a type of pointer to a pointer to
// std::int8_t.
// 5) par5 and ptr5 are expanded to a type of const pointer to a const
// pointer
// to a pointer to std::int8_t. This is three levels and is non-compliant.
// 6) par6 is of type pointer to pointer to std::int8_t because arrays
// are converted
// to a pointer to the initial element of the array.
// 7) ptr6 is of type pointer to array of std::int8_t.
// 8) par7 is of type pointer to pointer to pointer to
// std::int8_t because arrays are
// converted to a pointer to the initial element of the array. This is
// three
// levels and is non-compliant.
// 9) ptr7 is of type array of pointer to pointer to std::int8_t. This
// 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**

```cpp
#include <array>
#include <cstdint>
#include <iostream>
#include <stdexcept>
namespace
{
    const std::int32_t maxIterations = 10; // Compliant - assignment
    const char* const loopIterStr = "iter "; // Compliant - assignment
    const char separator = ":"; // Compliant - assignment
}

void f1() noexcept
{
    for (std::int32_t i = 0; i < 10; ++i) // Non-compliant
    {
        std::cout << "iter " << i << ":" << std::endl; // Compliant by exception
    }
}

void f2()
{
    // ...
    throw std::logic_error("Logic Error"); // Compliant
    // initialization of exception object
}

class C
{
public:
    C() : x(0), y(nullptr) // Compliant - initialization
    {
    }
    C(std::int8_t num, std::int32_t* ptr) : x(num), y(ptr) {}
private:
    std::int8_t x;
```
std::int32_t* y;
};
static std::int32_t* globalPointer = nullptr; // Compliant - assignment
void f3() noexcept
{
    C c1;
    // ...
    C c2(0, globalPointer); // Compliant - initialization of C object
}
std::int32_t f4(std::int32_t x, std::int32_t y) noexcept
{
    return x + y;
}
void f5() noexcept
{
    std::int32_t ret = f4(2, 5); // Non-compliant
    // ...
    std::int32_t x = 2;
    std::int32_t y = 5;
    ret = f4(x, y); // Compliant
    std::array<std::int8_t, 5> arr{{1, 2, 3, 4, 5}}; // Compliant
}

See also
• HIC++ v4.0 [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**
```cpp
void fn1(const std::vector<std::int32_t>& v)
{
    std::uint64_t sum = 0;
    std::for_each(v.begin(), v.end(), [&] (std::int32_t lhs) {
        sum += lhs;
```
Guidelines for the use of the C++14 language in critical and safety-related systems

AUTOSAR AP Release 17-03

```cpp
}; // Non-compliant

sum = 0;
std::for_each(v.begin(), v.end(), [&sum](std::int32_t lhs) {
    sum += lhs;
}); // Compliant

} // Non-compliant

void fn2()
{
    constexpr std::uint8_t n = 10;
    static std::int32_t j = 0;
    [n]() {
        std::int32_t array[n]; // Compliant
        j += 1; // Compliant by exception
    }
}
```

See also

- HIC++ v4.0 [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
```cpp
// $Id: A5-1-3.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
#include <cstdint>
void fn()
{
    std::int32_t i = 0;
    std::int32_t j = 0;
    auto lambda1 = [&i, &j] { ++i, ++j; }; // Non-compliant
    auto lambda2 = [&i, &j]() { ++i;
        ++j;
    }; // Compliant
}
```

See also

- HIC++ v4.0 [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

```cpp
// $Id: A5-1-4.cpp 272338 2017-03-28 08:15:01Z piotr.tanski $
#include <cstdint>
#include <functional>
std::function<std::int32_t()> f()
{
    std::int32_t i = 12;
    return ([&i]() -> std::int32_t {
        i = 100;
        return i;
    }); // Non-compliant
}
std::function<std::int32_t()> g()
{
    std::int32_t i = 12;
    return ([i]()->mutable -> std::int32_t { return ++i; }); // Compliant
}
void fn()
{
    auto lambda1 = f();
    std::int32_t i = lambda1(); // Undefined behavior
    auto lambda2 = g();
    i = lambda2(); // lambda2() returns 13
}
```

See also

- SEI CERT C++ [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. * sizeof(std::uint32_t)).

Example

```cpp
#include <cstdint>
#include <functional>
namespace {
constexpr std::int32_t bufferMax = 1024;
constexpr std::int8_t receiversMax = 10;
}  
class UDPClient {
// Implementation - size of UDPClient class exceeds 16 bytes
};

void f1() noexcept(false) {
    UDPClient client;
    std::uint8_t buffer[bufferMax];

    auto lambda1 = [client, buffer] () // Non-compliant - it is inefficient to capture
                     // UDPClient and buffer objects by copy in lambda
    {
        // Code
    };
    lambda1(); // lambda1 used locally only

    auto lambda2 = [&client, &buffer] () // Compliant - be aware that this construct
       // may introduce data races in parallel calls.
    {
        // Code
    };
    lambda2(); // lambda2 used locally only

    std::uint32_t number1 = 10;
    std::uint32_t number2 = 20;

    auto lambda3 = [number1, number2] () // Compliant by exception - the size of
                                         // std::uint32_t is 4 bytes (lesser or
                                         // equal to the size of a pointer -
                                         // depending on architecture)
    {
        // Code
    };
    lambda3(); // lambda3 used locally only
}
void f2(std::int8_t currentReceiver) noexcept(false) {
    std::function<void()> receiver;
```
if (currentReceiver < receiversMax)
{
    UDPClient client;
    receiver =
        [&client]() // Non-compliant - lambda is not used locally, client
            // object will go out of scope
        {
            // Code
        };
}

// ...
receiver(); // Undefined behavior, client object went out of scope

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 <cstdint>
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?
Guidelines for the use of the C++14 language in critical and safety-related systems
AUTOSAR AP Release 17-03

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

```cpp
#include <cstdint>
#include <functional>
#include <vector>

void fn()
{
    auto lambda1 = []() -> std::int8_t { return 1; };
    auto lambda2 = []() -> std::int8_t { return 1; };

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

    std::vector<decltype=lambda1)> v; // Non-compliant
    // v.push_back([]() { return 1; }); // Compilation error, type of pushed
    // lambda is different than decltype=lambda1)
    // using mylambda_t = decltype([](){ return 1; }); // Non-compliant -
    // compilation error
    auto lambda3 = []() { return 2; };
    using lambda3_t = decltype=lambda3); // Non-compliant - lambda3_t type can
    // not be used for lambda expression
    // declarations
    // lambda3_t lambda4 = []() { return 2; }; // Conversion error at
    // compile-time
    std::function<std::int32_t>() f1 = []() { return 3; };
    std::function<std::int32_t>() f2 = []() { return 3; };

    if (typeid=f1) == typeid=f2)) // Compliant - types are equal
    {
        // ...
    }
}
```
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
```
#include <cstdint>

void fn1()
{
    std::int16_t x = 0;
    auto f1 = [&x] () {
    
        auto f2 = []() {}; // Non-compliant
        f2();
    
        auto f4 = []() {}; // Non-compliant
        f4();
    
    }; // Non-compliant

    f1();
}

void fn2()
{
    auto f5 = []() {
        // Implementation
        // Compliant
        f5();
    }
}
```

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
// $Id: A5-2-1.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
class A
{
  public:
    virtual void f() noexcept;
};
class B : public A
{
  public:
    void f() noexcept override {}
};
void fn(A* aptr) noexcept
{
  // ...
  B* bptr = dynamic_cast<B*>(aptr); // Non-compliant
  if (bptr != nullptr)
  {
    // Use B class interface
  }
  else
  {
    // ...
  }
Guidelines for the use of the C++14 language in critical and safety-related systems

AUTOSAR AP Release 17-03

// Use A class interface
}
}

See also

- C++ Core Guidelines [10]: C.146: Use dynamic_cast where class hierarchy navigation is unavoidable.
- 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

// $Id: A5-2-2.cpp 271927 2017-03-24 12:01:35Z piotr.tanski $
#include <cstdint>

class C
{
  public:
    explicit C(std::int32_t) {}
    virtual void fn() noexcept {}
};

class D : public C
{
  public:
    void fn() noexcept override {}
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**

```cpp
1 // $Id: A5-2-3.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
```
Guidelines for the use of the C++14 language in critical and safety-related systems

AUTOSAR AP Release 17-03

#include <cstdint>

void f1(const char* str) noexcept(false)
{
  *(const_cast<char*>(str)) = '\0'; // Non-compliant - const qualification removed
}

class C
{
  public:
    explicit C(std::int32_t) {}
};

void f2() noexcept(false)
{
  C const a1 = C(10);
  C* a2 = const_cast<C*>(a1); // Non-compliant - const qualification removed
  C* a3 = (C*)&a1; // Non-compliant - const qualification removed
}

extern volatile std::int32_t* f3() noexcept;

void f4() noexcept
{
  volatile std::int32_t* ptr1 = f3();
  // ...
  std::int32_t* ptr2 = const_cast<std::int32_t*>(ptr1); // Non-compliant - volatile qualification removed
  // ...
  std::int32_t* ptr3 = (std::int32_t*)ptr1; // Non-compliant - volatile qualification removed
}

See also

• MISRA C++ 2008 [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
// $Id: A5-2-4.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
#include <cstdint>
#include <string>
void f1() noexcept
{
    std::string str = "Hello";
    std::int32_t* ptr = reinterpret_cast<std::int32_t*>(&str); // Non-compliant
}

struct A
{
    std::int32_t x;
    std::int32_t y;
};

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

class C : public B
{
}

class D : public B
{
}

void f2(A* ptr) noexcept
{
    B* b1 = reinterpret_cast<B*>(ptr); // Non-compliant
    std::int32_t num = 0;
    A* a1 = reinterpret_cast<A*>(num); // Non-compliant
    A* a2 = (A*)num; // Compliant with this rule, but non-compliant with Rule A5-2-2.
    num; // Compliant with this rule, but non-compliant with Rule A5-2-2. Cross-cast.
    B* b2 = reinterpret_cast<B*>(num); // Non-compliant
    D d;
    C* c1 = reinterpret_cast<C*>(&d); // Non-compliant - cross cast
    C* c2 = (C*)&d; // Compliant with this rule, but non-compliant with Rule A5-2-2. Cross-cast.
    B* b3 = &d; // Compliant - class D is a subclass of class B
}

See also

- MISRA C++ 2008 [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>
```cpp
#include <cstdint>
#include <iostream>

void fn1() noexcept
{
    constexpr std::int32_t arraySize = 16;
    std::int32_t array[arraySize][0];

    std::int32_t elem1 = array[0]; // Compliant - access with constant literal that
    // is less than ArraySize
    std::int32_t elem2 = array[12]; // Compliant - access with constant literal that
    // is less than ArraySize
    for (std::int32_t idx = 0; idx < 20; ++idx)
    {
        std::int32_t elem3 = array[idx]; // Non-compliant - access beyond ArraySize
        // bounds, which has 16 elements
    }

    std::int32_t shift = 25;
    std::int32_t elem4 = *(array + shift); // Non-compliant - access beyond ArraySize bounds

    std::int32_t index = 40;
    std::int32_t elem2 = array[index]; // Non-compliant - index may exceed the ArraySize bounds
    if (index < arraySize)
    {
        std::int32_t elem6 = array[index]; // Compliant - range check coded
    }
}

void fn2() noexcept
{
    constexpr std::int32_t arraySize = 32;
    std::array<std::int32_t, arraySize> array;
    array.fill(0);

    std::int32_t elem1 = array[10]; // Compliant - access with constant literal that
    // is less than ArraySize
    std::int32_t index = 40;
    std::int32_t elem2 = array[index]; // Non-compliant - access beyond ArraySize bounds
    try
    {
        std::int32_t elem3 = array.at(50); // Compliant - at() method provides a
        // range check, throwing an exception if
        // input exceeds the bounds
    }
    catch (std::out_of_range& e)
    {
        std::cerr << e.what() << std::endl;
    }
}
```
```cpp
catch (std::out_of_range&) {
    // Handle an error
}

for (auto&& e : array) // The std::array provides a possibility to iterate
    // over its elements with range-based loop
    // Iterate over all elements
```

---

### 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

```cpp
// $Id: A5-3-1.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
#include <typeinfo>
bool sideEffects() noexcept
{
    // Implementation
    return true;
}

class A
{
    public:
        static A& f1() noexcept { return a; }
    virtual ~A() {};
    private:
        static A a;
    ;
    A A::a;
    void f2() noexcept(false)
    {
        typeid(sideEffects()); // Non-compliant - sideEffects() function not called
        typeid(A::f1()); // Non-compliant - A::f1() functions called to determine
        // the polymorphic type
    }
}
```

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
```
// $Id: A5-5-1.cpp 272338 2017-03-28 08:15:01Z piotr.tanski $
#include <cstdint>
#include <stdexcept>
std::int32_t division1(std::int32_t a, std::int32_t b) noexcept
{
    return (a / b); // Non-compliant - value of b could be zero
}
std::int32_t division2(std::int32_t a, std::int32_t b)
{
    if (b == 0)
    {
        throw std::runtime_error("Division by zero error");
    }
    return (a / b); // Compliant - value of b checked before division
}
double fn()
{
    std::int32_t x = 20 / 0; // Non-compliant - undefined behavior
    x = division1(20, 0); // Undefined behavior
    x = division2(20,
    0); // Preconditions check will throw a runtime_error from
    // division2() function
    std::int32_t remainder = 20 % 0; // Non-compliant - undefined behavior
}
```

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**
```cpp
// $Id: A5-10-1.cpp 271927 2017-03-24 12:01:35Z piotr.tanski $

class A {
  public:
    virtual ~A() = default;
    void f1() noexcept {}  // Compliant
    void f2() noexcept {}  // Compliant
    virtual void f3() noexcept {}  // Compliant
};

void fn() {
  bool b1 = (&A::f1 == &A::f2);  // Compliant
  bool b2 = (&A::f1 == nullptr);  // Compliant
  bool b3 = (&A::f3 == nullptr);  // Compliant
  bool b4 = (&A::f3 != nullptr);  // Compliant
  bool b5 = (&A::f3 == &A::f1);  // Non-compliant
}
```

**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**

```cpp
// $Id: A5-16-1.cpp 271927 2017-03-24 12:01:35Z piotr.tanski $
#include <cstdint>
constexpr bool fn1(std::int32_t x)
{
    return (x > 0); // Compliant
}

std::int32_t fn2(std::int32_t x)
{
    std::int32_t i = (x >= 0 ? x : 0); // Compliant

    std::int32_t j =
    x + (i == 0 ? (x >= 0 ? x : -x) : i); // Non-compliant - nested conditional operator
    // and used as a sub-expression

    return (i > 0
            ? (j > 0 ? i + j : i)
            : (j > 0 ? j : 0)); // Non-compliant - nested conditional operator
}
```

**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

```cpp
#include <cstdint>

void f1(std::uint8_t choice) noexcept
{
    switch (choice)
    {
        default:
        break;
    } // Non-compliant, the switch statement is redundant
}

void f2(std::uint8_t choice) noexcept
{
    switch (choice)
    {
        case 0:
        // ...
        break;
        default:
        // ...
        break;
    } // Non-compliant, only 1 case-clause

    if (choice == 0) // Compliant, an equivalent if statement
    {
        // ...
    }
    else
    {
```

90 of 371
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**

```cpp
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};
```
for (std::int8_t idx = 0; idx < arraySize; ++idx) // Compliant
{
    array[idx] = idx;
}

for (std::int8_t idx = 0; idx < arraySize / 2; ++idx) // Compliant - for does not loop though all elements
{
    // ...
}

for (std::uint32_t* iter = std::begin(array); iter != std::end(array); ++iter) // Non-compliant
{
    // ...
}

for (std::int8_t idx = 0; idx < arraySize; ++idx) // Non-compliant
{
    // ...
}

for (std::uint32_t value : array) // Compliant - equivalent to non-compliant loops above
{
    // ...
}

for (std::int8_t idx = 0; idx < arraySize; ++idx) // Compliant
{
    if ((idx % 2) == 0)
    {
        // ...
    }
}

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

```cpp
#include <cstdint>

namespace {
  constexpr std::int32_t xlimit = 20;
  constexpr std::int32_t ylimit = 15;
  constexpr float zlimit = 2.5F;
  constexpr std::int32_t ilimit = 100;
}

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

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

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

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

See also

- MISRA C++ 2008 [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 \(\leq, <, >\) or \(\geq\).

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**
# Guidelines for the use of the C++14 language in critical and safety-related systems

AUTOSAR AP Release 17-03

```cpp
#include <cstdint>

namespace {
    constexpr std::int32_t loopLimit = 100;
}

void f1(std::int32_t n) noexcept {
    if (n < 0) {
        // goto exit; // Non-compliant - jumping to exit from here crosses ptr
        // pointer initialization, compilation
        // error
    }
}

std::int32_t* ptr = new std::int32_t(n);
// ...
exit:
delete ptr;
}

void f2() noexcept {
    // ...
    goto error; // Non-compliant
    // ...
    error:; // Error handling and cleanup
}

void f3() noexcept {
    for (std::int32_t i = 0; i < loopLimit; ++i) {
        for (std::int32_t j = 0; j < loopLimit; ++j) {
            for (std::int32_t k = 0; k < loopLimit; ++k) {
                if ((i == j) && (j == k)) {
                    // ...
                    goto loop_break; // Non-compliant
                }
            }
        }
    }
    loop_break:; // ...
}
```

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

```c++
#include <cstdint>
#include <limits>

void fn()
{
    const std::int16_t x1 = 5; // Compliant
    constexpr std::int16_t x2 = 5; // Compliant
    std::int16_t x3 = 5; // Non-compliant - x3 is not modified but not declared as const
                         // constant (const or constexpr)
}
```
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**

```cpp
//% $Id: A7-1-2.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
#include <cstdint>
std::int32_t pow1(std::int32_t number)
{
    return (number * number);
}
constexpr std::int32_t pow2(
    std::int32_t number) // Possible compile-time computing
    // because of constexpr specifier
    {
        return (number * number);
    }
void fn()
{
    constexpr std::int16_t i1 = 20; // Compliant, evaluated at compile-time
    const std::int16_t i2 = 20; // Non-compliant, possible run-time evaluation
    std::int32_t twoSquare =
        pow1(2); // Non-compliant, possible run-time evaluation
    const std::int32_t threeSquare =
        pow1(3); // Non-compliant, possible run-time evaluation
    // static_assert(threeSquare == 9, "pow1(3) did not succeed."); // Value
    // can not be static_assert-ed
    constexpr std::int32_t fiveSquare =
        pow2(5); // Compliant, evaluated at compile time
    static_assert(fiveSquare == 25,
                  "pow2(5) did not succeed."); // Compliant, constexpr
    // evaluated at compile time
    // constexpr std::int32_t int32Max =
```
class A
{
    public:
        static constexpr double pi = 3.14159265; // Compliant - value of PI can be
determined in compile time
        // constexpr double e = 2.71828182; // Non-compliant - constexprs need
        // to be static members, compilation error
        constexpr A() = default; // Compliant
};

See also

- C++ Core Guidelines [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
// $Id: A7-1-3.cpp 271687 2017-03-23 08:57:35Z piotr.tanski$
#include <cstdint>
using IntPtr = std::int32_t*;
using IntConstPtr = std::int32_t* const;
using ConstIntPtr = const std::int32_t*;
void fn(const std::uint8_t& input) // Compliant
{
    std::int32_t value1 = 10;
    std::int32_t value2 = 20;
    const IntPtr ptr1 =
&value1; // Non-compliant - deduced type is std::int32_t *
    // const, not const std::int32_t *

    // ptr1 = &value2; // Compilation error, ptr1 is read-only variable
IntPtr const ptr2 =
    &value1; // Compliant - deduced type is std::int32_t * const

    // ptr2 = &value2; // Compilation error, ptr2 is read-only variable
IntConstPtr ptr3 = &value1; // Compliant - type is std::int32_t * const, no
    // additional qualifiers needed

    // ptr3 = &value2; // Compilation error, ptr3 is read-only variable
ConstIntPtr ptr4 = &value1; // Compliant - type is const std::int32_t *

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

See also

• HIC++ v4.0 [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
// $Id: A7-1-4.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
#include <cstdint>
std::int32_t f1(register std::int16_t number) noexcept // Non-compliant
{
    return ((number * number) + number);
}
void f2(std::int16_t number) noexcept // Compliant
{
    register std::int8_t x = 10; // Non-compliant
    std::int32_t result = f1(number); // Compliant
See also

- JSF December 2005 [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

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

class A {
};

void f1() noexcept {
    auto x1 = 5;  // Non-compliant - initializer is of fundamental type
    auto x2 = 0.3F;  // Non-compliant - initializer is of fundamental type
    auto x3 = {8};  // Non-compliant - initializer is of fundamental type
    std::vector<std::int32_t> v;
    auto x4 = v.size();  // Compliant with case (1) - x4 is of size_t type that
    // is returned from v.size() method
```
auto a = A{}; // Compliant with case (2)

auto lambda1 = []() -> std::uint16_t {
    return 5U;
}; // Compliant with case (2) - lambda1 is of non-fundamental lambda
    // expression type
auto x5 = lambda1(); // Compliant with case (1) - x5 is of
    // std::uint16_t type

void f2() noexcept {
    auto lambda1 = [](auto x, auto y) -> decltype(x + y) {
        return (x + y);
    }; // Compliant with cases (2) and (3)
    auto y1 = lambda1(5.0, 10); // Compliant with case (1)
}

template <typename T, typename U>
auto f3(T t, U u) noexcept -> decltype(t + u) // Compliant with case (4)
{
    return (t + u);
}

template <typename T>

class B
{
    public:
    T fn(T t);
};

template <typename T>
auto B<T>::fn(T t) -> T // Compliant with case (4)
{
    // ...
    return t;
}

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

```cpp
// $Id: A7-1-6.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
#include <cstdint>
#include <type_traits>

typedef std::int32_t (*fPointer1)(std::int32_t); // Non-compliant

using fPointer2 = std::int32_t (*)(std::int32_t); // Compliant

// template<typename T>
// typedef std::int32_t (*fPointer3)(T); // Non-compliant - compilation error

template<typename T>
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

```cpp
// $Id: A7-1-7.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
#include <cstdint>
#include <vector>

typedef std::int32_t* ptr; // Compliant
typedef std::int32_t *pointer, value; // Non-compliant
```
void fn1() noexcept {
    std::int32_t x = 0; // Compliant
    std::int32_t y = 7, *p1 = nullptr; // Non-compliant
    std::int32_t const *p2, z = 1; // Non-compliant
}

void fn2() {
    std::vector<std::int32_t> v{1, 2, 3, 4, 5};
    for (auto iter=v.begin(), end=v.end(); iter != end;
        ++iter) // Compliant by exception
    {
        // ...
    }
}

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**

```cpp
#include <cstdint>

typedef std::int32_t int1; // Compliant
std::int32_t typedef int2; // Non-compliant

class C {
public:
    virtual inline void f1(); // Compliant
    inline virtual void f2(); // Compliant
    void virtual inline f3(); // Non-compliant
private:
```
std::int32_t mutable x; // Non-compliant
mutable std::int32_t y; // Compliant
}

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**

```cpp
#include <cstdint>
enum class E : std::uint8_t
{
    Ok = 0,
    Repeat,
    Error
};
E convert1(std::uint8_t number) noexcept
{
    E result = E::Ok; // Compliant
    switch (number)
    {
        case 0:
        {
            result = E::Ok; // Compliant
            break;
        }
        case 1:
        {
            result = E::Repeat; // Compliant
```


```cpp
break;
}
case 2:
{
    result = E::Error; // Compliant
    break;
}
case 3:
{
    constexpr std::int8_t val = 3;
    result = static_cast<E>(val); // Non-compliant - value 3 does not
    // correspond to any of E’s
    // enumerators
    break;
}
default:
{
    result = static_cast<E>(0); // Compliant - value 0 corresponds to E::Ok
    break;
}
} return result;
}
E convert2(std::uint8_t userInput) noexcept
{
    E result = static_cast<E>(userInput); // Non-compliant - the range of
    // userInput may not correspond to
    // any of E’s enumerators
    return result;
}
E convert3(std::uint8_t userInput) noexcept
{
    E result = E::Error;
    if (userInput < 3)
    {
        result = static_cast<E>(userInput); // Compliant - the range of
        // userInput checked before casting
        // it to E enumerator
    }
    return result;
}
```

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

```cpp
#include <cstdint>

enum class E1 // Non-compliant
{
    E10,
    E11,
    E12
};

enum class E2 : std::uint8_t // Compliant
{
    E20,
    E21,
    E22
};

enum E3 // Non-compliant
{
    E30,
    E31,
    E32
};

enum E4 : std::uint8_t // Compliant - violating another rule
{
    E40,
    E41,
    E42
};

enum class E5 : std::uint8_t // Non-compliant - will not compile
{
    E50 = 255,
    // E5_1, // E5_1 = 256 which is outside of range of underlying type
    // std::uint8_t
    // - compilation error
    // E5_2 // E5_2 = 257 which is outside of range of underlying type
    // std::uint8_t
    // - compilation error
};
```

See also

- HIC++ v4.0 [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

```cpp
#include <cstdint>

enum E1 : std::int32_t // Non-compliant
{
    E10,
    E11,
    E12
};

enum class E2 : std::int32_t // Compliant
{
    E20,
    E21,
    E22
};

// static std::int32_t E1_0 = 5; // E1_0 symbol redeclaration, compilation error

static std::int32_t E20 = 5; // No redeclarations, no compilation error

extern void f1(std::int32_t number)
{
}

void f2()
{
    f1(0);
    f1(E11); // Implicit conversion from enum to std::int32_t type
    // f1(E2::E2_1); // Implicit conversion not possible, compilation error
    f1(static_cast<std::int32_t>(E2::E21)); // Only explicit conversion allows to
```
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**

```cpp
//% $Id: A7-2-4.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
#include <cstdint>
enum class Enum1 : std::uint32_t
{
    One,
    Two = 2, // Non-compliant
    Three
};
enum class Enum2 : std::uint32_t // Compliant (none)
{
    One,
    Two,
    Three
};
enum class Enum3 : std::uint32_t // Compliant (the first)
{
    One = 1,
    Two,
    Three
};
enum class Enum4 : std::uint32_t // Compliant (all)
{
    One = 1,
    Two = 2,
    Three = 3
};
```

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 <cstdint>
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

```cpp
#include <cstdint>

class A
{
    explicit A(std::uint8_t n) : number(n) {}

    ~A() { number = 0U; }

    // Implementation

    private:
```
std::uint8_t number;

const A& fn1(const A& ref) noexcept // Non-compliant - the function returns a
  // reference to const reference parameter
  // which may bind to temporary objects.
  // According to C++14 Language Standard, it
  // is undefined whether a temporary object is introduced for const
  // reference
  // parameter
{
  // ...
  return ref;
}

const A& fn2(A& ref) noexcept // Compliant - non-const reference parameter does
  // not bind to temporary objects, it is allowed
  // that the function returns a reference to such
  // a parameter
{
  // ...
  return ref;
}

const A* fn3(const A& ref) noexcept // Non-compliant - the function returns a
  // pointer to const reference parameter
  // which may bind to temporary objects.
  // According to C++14 Language Standard, it
  // is undefined whether a temporary object is introduced for const
  // reference
  // parameter
{
  // ...
  return &ref;
}

template <typename T>
T& fn4(T& v) // Compliant - the function will not bind to temporary objects
{
  // ...
  return v;
}

void f() noexcept
{
  A a(5);
  const A& ref1 = fn1(a); // fn1 called with an lvalue parameter from an
                          // outer scope, ref1 refers to valid object
  const A& ref2 = fn2(a); // fn2 called with an lvalue parameter from an
                          // outer scope, ref2 refers to valid object
  const A* ptr1 = fn3(a); // fn3 called with an lvalue parameter from an
                          // outer scope, ptr1 refers to valid object
  const A& ref3 = fn4(a); // fn4 called with T = A, an lvalue parameter from
                          // an outer scope, ref3 refers to valid object
  const A& ref4 = fn1(A{10}); // fn1 called with an rvalue parameter
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**

```cpp
#include <cstdint>

static std::int32_t fn1(std::int32_t number);
static std::int32_t fn2(std::int32_t number);
static std::int32_t fn3(std::int32_t number);
static std::int32_t fn4(std::int32_t number);

std::int32_t fn1(std::int32_t number)
{
    if (number > 1)
    {
        number = number * fn1(number - 1); // Non-compliant
    }
    return number;
}

std::int32_t fn2(std::int32_t number)
```
for (std::int32_t n = number; n > 1; --n) // Compliant
{
    number = number * (n - 1);
}

return number;
std::int32_t fn3(std::int32_t number)
{
    if (number > 1)
    {
        number = number * fn3(number - 1); // Non-compliant
    }

    return number;
}

std::int32_t fn4(std::int32_t number)
{
    if (number == 1)
    {
        number = number * fn3(number - 1); // Non-compliant
    }

    return number;
}
template <typename T>
T fn5(T value)
{
    return value;
}

template <typename T, typename... Args>
T fn5(T first, Args... args)
{
    return first + fn5(args...); // Compliant by exception - all of the
    // arguments are known during compile time
}

std::int32_t fn6() noexcept
{
    std::int32_t sum = fn5<std::int32_t, std::uint8_t, float, double>(
        10, 5, 2.5, 3.5); // An example call to variadic template function
    // ...
    return sum;
}
constexpr std::int32_t fn7(std::int32_t x, std::int8_t n)
{
    if (n >= 0)
    {
        x += x;
        return fn5(x, --n); // Compliant by exception - recursion evaluated at
        // compile time
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**

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

template <typename T>
typename A<T>::Type A<T>::f(T const&) noexcept // Non-compliant
{
    // Implementation
}

template <typename T>
auto A<T>::g(T const&) noexcept -> Type // Compliant
{
    // Implementation
}

See also

- HIC++ v4.0 [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

// $Id: A8-4-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
void print1(char* format, ...) // Non-compliant - variadic arguments are used
{
  // ...}

template <typename First, typename... Rest>
void print2(const First& first, const Rest&... args) // Compliant
{
  // ...}

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

// $Id: A8-4-2.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
#include <cstdint>
#include <stdexcept>
std::int32_t f1() noexcept // Non-compliant
{
  // ...
} std::int32_t f2(std::int32_t x) noexcept(false)
if (x > 100)
{
    throw std::logic_error("Logic Error"); // Compliant by exception
}

return x; // Compliant

std::int32_t f3(std::int32_t x, std::int32_t y)
{
    if (x > 100 || y > 100)
    {
        throw std::logic_error("Logic Error"); // Compliant by exception
    }
    if (y > x)
    {
        return (y - x); // Compliant
    }
    return (x - y); // Compliant
}

See also

- MISRA C++ 2008 [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 Initializers

**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.
Guidelines for the use of the C++14 language in critical and safety-related systems

AUTOSAR AP Release 17-03

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

```cpp
// $Id: A8-5-1.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
#include <cstdint>
#include <string>
class A {
};
class B {
};
class C : public virtual B, public A {
  public:
    C() : B(), A(), s() {} // Compliant
    // C() : A(), B() { } // Non-compliant - incorrect order of initialization
  private:
    std::string s;
};
class D {
};
class E {
};
class F : public virtual A, public B, public virtual D, public E {
  public:
    F() : A(), D(), B(), E(), number1(0), number2(0U) {} // Compliant
    F(F const& oth)
    : B(), E(), A(), D(), number1(oth.number1), number2(oth.number2)
    {
    } // Non-compliant - incorrect
    // order of initialization
```
private:
  std::int32_t number1;
  std::uint8_t number2;
};

See also

- HIC++ v4.0 [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**

```cpp
#include <cstdint>
#include <initializer_list>
void f1() noexcept
```
5  std::int32_t x1 =
6      7.9; // Non-compliant - x1 becomes 7 without compilation error
7 // std::int32_t y {7.9}; // Compliant - compilation error, narrowing
8  std::int8_t x2{50}; // Compliant
9  std::int8_t x3 = {50}; // Non-compliant - std::int8_t x3 50  is equivalent
10    // and more readable
11  std::int8_t x4 =
12      1.0; // Non-compliant - implicit conversion from double to std::int8_t
13  std::int8_t x5 = 300; // Non-compliant - narrowing occurs implicitly
14  std::int8_t x6(x5); // Non-compliant
15 }
16 class A
17 {
18   public:
19      A(std::int32_t first, std::int32_t second) : x{first}, y{second} {}
20   }
21 private:
22      std::int32_t x;
23      std::int32_t y;
24   }
25 struct B
26 {
27      std::int16_t x;
28      std::int16_t y;
29   }
30 class C
31 {
32   public:
33      C(std::int32_t first, std::int32_t second) : x{first}, y{second} {}
34      C(std::initializer_list<std::int32_t> list) : x{0}, y{0} {}
35   }
36 private:
37      std::int32_t x;
38      std::int32_t y;
39   }
40 void f2() noexcept
41 {
42   A a1{1, 5}; // Compliant - calls constructor of class A
43   A a2 = {1, 5}; // Non-compliant - calls a default constructor of class A
44   // and not copy constructor or assignment operator.
45   A a3{1, 5}; // Non-compliant
46   B b1{5, 0}; // Compliant - struct members initialization
47   C c1{2, 2}; // Compliant - C(std::initializer_list<std::int32_t>)
48      // constructor is
49      // called
50   C c2{2, 2}; // Compliant by exception - this is the only way to call
51      // C(std::int32_t, std::int32_t) constructor
52   C c3{{}}; // Compliant - C(std::initializer_list<std::int32_t>) constructor
53      // is
54      // called with an empty initializer_list
C c4({2, 2}); // Compliant by exception -
    // C(std::initializer_list<std::int32_t>)
    // constructor is called
};
template <typename T, typename U>
void f1(T t, U u) noexcept(false)
{
    std::int32_t x = 0;
    T v1(x); // Non-compliant
    T v2(x); // Compliant - v2 is a variable
    // auto y = T(u); // Non-compliant - is it construction or cast?
    // Compilation error
};
void f3() noexcept
{
    f1(0, "abcd"); // Compile-time error, cast from const char* to int
}

See also

• C++ Core Guidelines [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

```cpp
#include <cstdint>
#include <initializer_list>
void fn() noexcept
{
    auto x1(10); // Compliant - the auto-declared variable is of type int, but
    // not compliant with A8-5-2.
```
auto x2{10}; // Non-compliant - according to C++14 standard the auto-declared variable is of type std::initializer_list.
// However, it can behave differently on different compilers.
auto x3 = 10; // Compliant - the auto-declared variable is of type int, but non-compliant with A8-5-2.
auto x4 = {10}; // Non-compliant - the auto-declared variable is of type std::initializer_list, non-compliant with A8-5-2.
std::int8_t x5{10}; // Compliant
}

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

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

class A
{
public:
    A() : x(0), y(0) {}
    A(std::int32_t first, std::int32_t second) : x(first), y(second) {}
    A(std::initializer_list<std::int32_t> list)
        : x(0), y(0) // Non-compliant - class A does not store a collection of objects
        // of objects
    {
        // ...
    }

private:
```
```cpp
std::int32_t x;
std::int32_t y;
};
class B
{
public:
B() : collection() {}
B(std::int32_t size, std::int32_t value) : collection(size, value) {}
B(std::initializer_list<std::int32_t> list)
 : collection{
     list }; // Compliant - class B stores a collection of objects
private:
    std::vector<std::int32_t> collection;
};
class C
{
public:
C() : array{0} {}
C(std::initializer_list<std::int32_t> list)
 : array{0} // Compliant - class C stores a collection of objects
{
    std::copy(list.begin(), list.end(), array);
}
private:
    static constexpr std::int32_t size = 100;
std::int32_t array[size];
};
class D : public C
{
public:
D() : C() {}
D(std::initializer_list<std::int32_t> list)
 : C(list) // Compliant - class D inherits a collection of objects
     // from class C
{
}
};
class E
{
public:
E() : container() {}
E(std::initializer_list<std::int32_t> list)
 : container(list) // Compliant - class E stores class C which
     // stores a collection of objects
{
}
```
private:
    C container;
};

void f1() noexcept
{
    A a1{};  // Calls A::A()
    A a2({});  // Calls A::A(std::initializer_list<std::int32_t>)
    A a3(0, 1);  // Calls A::A(std::initializer_list<std::int32_t>)
    A a4(0, 1);  // Calls A::A(std::int32_t, std::int32_t)
}

void f2() noexcept
{
    B b1{};  // Calls B::B()
    B b2({});  // Calls B::B(std::initializer_list<std::int32_t>)
    B b3(1, 2);  // Calls B::B(std::initializer_list<std::int32_t>)
    B b4(10, 0);  // Calls B::B(std::int32_t, std::int32_t)
}

void f3() noexcept
{
    C c1{};  // Calls C::C()
    C c2({});  // Calls C::C(std::initializer_list<std::int32_t>)
    C c3(1, 2, 3);  // Calls C::C(std::initializer_list<std::int32_t>)
}

void f4() noexcept
{
    D d1{};  // Calls D::D()
    D d2({});  // Calls D::D(std::initializer_list<std::int32_t>)
    D d3(1, 2, 3);  // Calls D::D(std::initializer_list<std::int32_t>)
}

void f5() noexcept
{
    E e1{};  // Calls E::E()
    E e2({});  // Calls E::E(std::initializer_list<std::int32_t>)
    E e3(1, 2, 3);  // Calls E::E(std::initializer_list<std::int32_t>)
}

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**

```cpp
#include <cstdint>
#include <memory>
#include <utility>

class A
{
public:
    explicit A(std::int32_t number) : x(number) {}
    // Implementation
    std::int32_t& getX() noexcept // Non-compliant - x is a resource owned by the A class
    {
        return x;
    }

private:
    std::int32_t x;
};

void fn1() noexcept
{
    A a(10);
    std::int32_t& number = a.getX();
    number = 15; // External modification of private class data
}
```
```cpp
class B
{
    public:
        explicit B(std::shared_ptr<std::int32_t> ptr) : sharedptr(std::move(ptr)) {}
        // Implementation
    std::shared_ptr<std::int32_t> getSharedPtr() const noexcept // Compliant - sharedptr is a variable being shared between // instances
    {
        return sharedptr;
    }

    private:
        std::shared_ptr<std::int32_t> sharedptr;
};

void fn2() noexcept
{
    std::shared_ptr<std::int32_t> ptr = std::make_shared<std::int32_t>(10);
    B b1{ptr};
    B b2{ptr};
    *ptr = 50; // External modification of ptr which shared between b1 and b2 // instances
    auto shared = b1.getSharedPtr();
    *shared = 100; // External modification of ptr which shared between b1 and // b2 instances
}

class C
{
    public:
        explicit C(std::int32_t number)
            : ownedptr{std::make_unique<std::int32_t>(number)}
    {
    }
    // Implementation
    const std::unique_ptr<std::int32_t>& getOwnedPtr() const noexcept // Non-compliant - only unique_ptr is const, the object that // it is pointing to is modifiable
    {
        return ownedptr;
    }
    const std::int32_t& getData() const noexcept // Compliant
    {
        return *ownedptr;
    }

    private:
        std::unique_ptr<std::int32_t> ownedptr;
};

void fn3() noexcept
{
    C c{10};
}
```
const std::int32_t& data = c.getData();
// data = 20; // Can not modify data, it is a const reference
const std::unique_ptr<std::int32_t>& ptr = c.getOwnedPtr();
*ptr = 20; // Internal data of class C modified

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**

```cpp
// $Id: A9-6-1.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
#include <cstdint>
enum class E1 : std::uint8_t
{
    E11,
    E12,
    E13
};
enum class E2 : std::int16_t
{
    E21,
    E22,
    E23
};
enum class E3
{
    E31,
    E32,
    E33
};
enum E4
{
    E41,
    E42,
    E43
};
class C
{
    public:
    std::int32_t a : 2; // Non-compliant - signed integral type
    std::uint8_t b : 2U; // Compliant
    bool c : 1; // Non-compliant - it is implementation-defined whether bool is
                 // signed or unsigned
    char d : 2; // Non-compliant
    wchar_t e : 2; // Non-compliant
    E1 f1 : 2; // Compliant
    E2 f2 : 2; // Non-compliant - E2 enum class underlying type is signed
    // int
    E3 f3 : 2; // Non-compliant - E3 enum class does not explicitly define
                 // underlying type
    E4 f4 : 2; // Non-compliant - E4 enum does not explicitly define underlying
                 // type
};
void fn() noexcept
{
    C c;
    c.f1 = E1::E11;
}
```
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

```cpp
// $Id: A10-1-1.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
#include <cstdint>

class A
{
  public:
    void f1() noexcept(false) {}

  private:
    std::int32_t x{0};
    std::int32_t y{0};
};

class B
{
  public:
    void f2() noexcept(false) {}

  private:
    std::int32_t x{0};
};
class C : public A, public B // Non-compliant - A and B are both not interface classes
{
}
```
class D
{
  public:
    virtual ~D() = 0;
    virtual void f3() noexcept = 0;
    virtual void f4() noexcept = 0;
};
class E
{
  public:
    static constexpr std::int32_t value{10};
    virtual ~E() = 0;
    virtual void f5() noexcept = 0;
};
class F : public A, public B, public D, public E // Non-compliant - A and B are both not interface classes
{
};
class G : public A, public D, public E // Compliant - D and E are interface classes
{
};

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

```cpp
// $Id: A10-2-1.cpp 271927 2017-03-24 12:01:35Z piotr.tanski$
class A {
  public:
    virtual ~A() = default;
    void f() noexcept {}
    virtual void g() noexcept {}
};

class B : public A {
  public:
    void f() noexcept // Non-compliant - f() function from A class hidden by B class
    {
    }
    void g() noexcept override // Compliant - g() function from A class
    // overridden by B class
    {
    }
};
```
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**

```cpp
// $Id: A10-3-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
class A
{
public:
    virtual ~A() {} // Compliant
    virtual void f() noexcept = 0; // Compliant
    virtual void g() noexcept final = 0; // Non-compliant - virtual final pure function is redundant
    virtual void h() noexcept final // Non-compliant - function is virtual and final
    {
    }
    virtual void k() noexcept // Compliant
    {
```
virtual void j() noexcept {}
virtual void m() noexcept // Compliant
{
}

virtual void z() noexcept // Compliant
{
}

virtual A& operator+=(A const& rhs) noexcept // Compliant
{
    // ...
    return *this;
}

class B : public A
{
public:
    -B() override {} // Compliant
    virtual void f() noexcept override // Non-compliant - function is specified // with virtual and override
    {
    }

    void k() noexcept override
    final // Non-compliant - function is specified with override and final
    {
    }

    virtual void m() noexcept // Compliant - violates A10-3-2
    {
    }

    void z() noexcept override // Compliant
    {
    }

    void j() noexcept // Non-compliant - virtual function but not marked as // overrider
    {
    }

    A& operator+=(A const& rhs) noexcept override // Compliant - to override // the operator correctly, // its signature needs to be // the same as in the base // class
    {
        // ...
        return *this;
    }
};

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 overrides, too.

Also, note that this rule applies to a pure virtual function which overrides another pure virtual function.

Example
```cpp
// $Id: A10-3-2.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
class A
{
  public:
    virtual ~A() {} // Compliant
    virtual void f() noexcept = 0;
    virtual void g() noexcept {} // Non-compliant
    virtual void z() noexcept {} // Non-compliant
    virtual A& operator+=(A const& oth) = 0;
};
class B : public A
{
  public:
    ~B() override {} // Compliant
    void f() noexcept // Non-compliant
    {
    }
    virtual void g() noexcept // Non-compliant
    {
    }
    void z() noexcept override // Compliant
    {
    }
    B& operator+=(A const& oth) override // Compliant
    {
      return *this;
    }
};
class C : public A
{
  public:
    ~C() {} // Non-compliant
    void f() noexcept override // Compliant
    {
    }
    void g() noexcept override // Compliant
```
```cpp
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**
```cpp
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**

```cpp
    // $Id: A10-3-4.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
    class A
    {
        public:
            virtual A& operator=(A const& oth) = 0; // Non-compliant
            virtual A& operator+=(A const& rhs) = 0; // Non-compliant
    }
    class B : public A
    {
        public:
            B& operator=(A const& oth) override // It needs to take an argument of type A& in order to override
            {
                return *this;
            }
            B& operator+=(A const& oth) override // It needs to take an argument of type A& in order to override
            {
                return *this;
            }
            B& operator-=(B const& oth) // Compliant
            {
                return *this;
            }
    }
    class C : public A
    {
        public:
            C& operator=(A const& oth) override // It needs to take an argument of type A& in order to override
            {
                return *this;
            }
```
C& operator+=(A const& oth) override // It needs to take an argument of type A& in order to override
{
    return *this;
}

C& operator-=(C const& oth) // Compliant
{
    return *this;
}

// class D : public A
{//
// public:
// D& operator=(D const& oth) override // Compile time error - this method
does not override because of different signature
// {
//     return *this;
// }
// D& operator+=(D const& oth) override // Compile time error - this method
does not override because of different signature
// {
//     return *this;
// }
//};
void fn() noexcept
{
    B b;
    C c;
    b = c; // Calls B::operator= and accepts an argument of type C
    b += c; // Calls B::operator+= and accepts an argument of type C
    c = b; // Calls C::operator= and accepts an argument of type B
    c += b; // Calls C::operator+= and accepts an argument of type B
    // b -= c; // Compilation error, because of types mismatch. Expected behavior
    // c -= b; // Compilation error, because of types mismatch. Expected behavior
    B b2;
    C c2;
    b -= b2;
    c -= c2;
}

See also

- none
Rule M10-3-3 (required, implementation, automated)
A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.

See MISRA C++ 2008 [6]
See: A10-3-2 for pure virtual function overriders 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

```cpp
// $Id: A11-0-1.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
#include <cstdint>
#include <limits>

class A // Compliant - A provides user-defined constructors, invariant and
    // interface
{
    std::int32_t x; // Data member is private by default

    public:
        static constexpr std::int32_t maxValue =
            std::numeric_limits<std::int32_t>::max();
        A() : x(maxValue) {};
        explicit A(std::int32_t number) : x(number) {};
        A(A const&) = default;
        A(A&&) = default;
        A& operator=(A const&) = default;
        A& operator=(A&&) = default;
```
std::int32_t getX() const noexcept { return x; }
void setX(std::int32_t number) noexcept { x = number; }

struct B // Non-compliant - non-POD type defined as struct
{
    public:
        static constexpr std::int32_t maxValue =
            std::numeric_limits<std::int32_t>::max();
        B() : x(maxValue) {}  
        explicit B(std::int32_t number) : x(number) {}  
        B(B const&) = default; 
        B(B&&) = default;
        B& operator=(B const&) = default;
        B& operator=(B&&) = default;
  
        std::int32_t getX() const noexcept { return x; }
        void setX(std::int32_t number) noexcept { x = number; }

    private:
        std::int32_t x;  // Need to provide private access specifier for x member
};

struct C // Compliant - POD type defined as struct
{
    std::int32_t x;
    std::int32_t y;
};

class D // Compliant - POD type defined as class, but not compliant with
        // M11-0-1
{
    public:
        std::int32_t x;
        std::int32_t y;
};

See also

- C++ Core Guidelines [10]: C.2: Use class if the class has an invariant; use struct if the data members can vary independently.
- stackoverflow.com [16]: 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

```cpp
// $Id: A11-0-2.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
#include <cstdint>

struct A // Compliant
{
  std::int32_t x;
  double y;
};

struct B // Compliant
{
  std::uint8_t x;
  A a;
};

struct C // Compliant
{
  float x = 0.0f;
  std::int32_t y = 0;
  std::uint8_t z = 0U;
};

struct D // Non-compliant
{
  public:
    std::int32_t x;

  protected:
    std::int32_t y;

  private:
    std::int32_t z;
};

struct E // Non-compliant
{
  public:
    std::int32_t x;

  void fn() noexcept {}

  private:
    void f1() noexcept(false) {}
};

struct F : public D // Non-compliant
{ }
};
```
See also

- stackoverflow.com [16]: 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**

```c++
// $Id: A11-3-1.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
class A
{
  public:
  A& operator+=(A const& oth);
  friend A const operator+(A const& lhs, A const& rhs); // Non-compliant
};
class B
{
  public:
  B& operator+=(B const& oth);
};
B const operator+(B const& lhs, B const& rhs) // Compliant
{
  // Implementation
}
```

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

```cpp
#include <cstdint>

class A // Compliant - the class A follow the "Rule of six" rule
{
  public:
    A(); // Non-default constructor
    ~A() = default;
    A(A const&) = default;
    A& operator=(A const&) = default;
    A(A&&) = delete;
    A& operator=(A&&) = delete;
};

class B // Non-compliant - some special functions are defined but not all of
  // them
{
  public:
    B();
    ~B();
  private:
    std::int32_t* pointer;
};

struct C // Compliant - no special functions are defined
{
  std::int32_t number;
};
```

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
Guidelines for the use of the C++14 language in critical and safety-related systems

AUTOSAR AP Release 17-03

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

```
// $Id: A12-1-1.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
#include <cstdint>
class Base {
    // Implementation
};
class VirtualBase {
};
class A : public virtual VirtualBase, public Base {
public:
    A() : VirtualBase{}, Base{}, i{0}, j{0} // Compliant
    {
    }
    A(A const& oth) : Base{}, j{0} // Non-compliant - VirtualBase base class and member
    {
        // i not initialized
    }
private:
    std::int32_t i;
    std::int32_t j;
    static std::int32_t k;
};
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**

```cpp
// $Id: A12-1-2.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
#include <cstdint>
#include <utility>
class A
{
  public:
  A() : i1(0), i2(0) // Compliant - i1 and i2 are initialized by the
  // constructor only. Not compliant with A12-1-3
  {
  }
  // Implementation

  private:
  std::int32_t i1;
  std::int32_t i2;
};
class B
{
  public:
  // Implementation

  private:
  std::int32_t i1(0);
```
std::int32_t i2 = 0; // Compliant - both i1 and i2 are initialized by NSDMI only

class C
{
public:
    C() : i2(0) // Non-compliant - i1 is initialized by NSDMI, i2 is in
    // member in member initializer list
    {
    }
    C(C const& oth) : i1(oth.i1), i2(oth.i2) // Compliant by exception
    {
    }
    C(C&& oth)
    : i1{std::move(oth.i1)},
      i2{std::move(oth.i2)} // Compliant by exception
    {
    }
    // Implementation

private:
    std::int32_t i1;
    std::int32_t i2;
};

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
// $Id: A12-1-3.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
#include <cstdint>
#include <string>
class A
{
public:
    A() : x(0), y(0.0F), str() // Non-compliant
    {
    }
private:
std::int32_t x;
float y;
std::string str;
};
class B
{
public:
// ...
private:
std::int32_t x = 0;  // Compliant
float y = 0.0F;  // Compliant
std::string str = "";  // Compliant
};
class C
{
public:
C() : x(0), y(0.0F), str()  // Compliant
{
}
C(std::int32_t i, float f, std::string s) : x(i), y(f), str(s)  // Compliant
{
}
// ...
private:
std::int32_t x = 0;  // Non-compliant - there’s a constructor that initializes C
// class with user input
float y = 0.0F;  // Non-compliant - there’s a constructor that initializes C
// class with user input
std::string str = "";  // Non-compliant - there’s a constructor that
// initializes C class with user input
};

See also

- C++ Core Guidelines [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

```cpp
// $Id: A12-1-4.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
#include <cstdint>

class A {
  public:
    explicit A(std::int32_t number) : x(number) {} // Compliant
    A(A const&) = default;
    A(A&&) = default;
    A& operator=(A const&) = default;
    A& operator=(A&&) = default;

  private:
    std::int32_t x;
};
class B {
  public:
    B(std::int32_t number) : x(number) {} // Non-compliant
    B(B const&) = default;
    B(B&&) = default;
    B& operator=(B const&) = default;
    B& operator=(B&&) = default;

  private:
    std::int32_t x;
};
void f1(A a) noexcept {
}
void f2(B b) noexcept {
}
void f3() noexcept {
  f1(A(10));
  // f1(10); // Compilation error - because of explicit constructor it is not
  // possible to implicitly convert integer
  // to type of class A
  f2(B(20));
  f2(20); // No compilation error - implicit conversion occurs
}

See also

- MISRA C++ 2008 [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

```cpp
// $Id: A12-4-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
class A
{
    public:
    ~A() // Non-compliant
};
class B : public A
{
};
class C
{
    public:
    virtual ~C() // Compliant
    {
    }
};
class D : public C
{
};
class E
{
    protected:
    ~E(); // Compliant
};
class F : public E
{
};
void f1(A* obj1, C* obj2)
{
    // ...
    delete obj1; // Only destructor of class A will be invoked
    delete obj2; // Both destructors of D and C will be invoked
}
```
void f2()
{
    A* a = new B;
    C* c = new D;
    f1(a, c);
}

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

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

class C // Compliant - class C is not final, and its destructor is virtual. It
  // can be used as a base class
{
  public:
    C() = default;
    C(C const&) = default;
    C(C&&) = default;
    C& operator=(C const&) = default;
    C& operator=(C&&) = default;
    virtual ~C() = default; // Public virtual destructor
};

class AA : public A
{
};

// class BA : public B // Compilation error - can not derive from final base
// class B
//{
//};
class CA : public C
{
};

void fn() noexcept
{
  AA obj1;
  CA obj2;
  A& ref1 = obj1;
  C& ref2 = obj2;

  ref1.~A(); // Calls A:~A() only
  ref2.~C(); // Calls both CA:~CA() and C:~C()
}

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**

```cpp
// $Id: A12-6-1.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
#include <cstdint>
#include <string>
class A
{
    public:
        A(std::int32_t n, std::string s) : number{n}, str{s} // Compliant
    { 
    }
    // Implementation

    private:
        std::int32_t number;
        std::string str;
    }

class B
{
    public:
        B(std::int32_t n, std::string s) // Non-compliant - no member initializers
    {
            number = n;
            str = s;
        }
    // Implementation

    private:
        std::int32_t number;
        std::string str;
    }

class C
{
    public:
        C(std::int32_t n, std::string s) : number{n}, str{s} // Compliant
    { 
            n += 1; // This does not violate the rule
            str.erase(str.begin(),
                   str.begin() + 1); // This does not violate the rule
        }
    // Implementation

    private:
        std::int32_t number;
```
Guidelines for the use of the C++14 language in critical and safety-related systems

AUTOSAR AP Release 17-03

std::string str;

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.


Example

// $Id: A12-7-1.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
#include <cstdint>
#include <utility>
class A
{
    public:
        A() : x(0), y(0) {} // Compliant
        A(std::int32_t first, std::int32_t second) : x(first), y(second) {} // Compliant
    // -
    // anyway, such
    // a constructor
    // cannot be
    // defaulted.
        A(const A& oth)
            : x(oth.x),
            y(oth.y) // Non-compliant - equivalent to the implicitly defined copy constructor
        { }
A(A&& oth)  
  : x(std::move(oth.x)),  
    y(std::move(  
      oth.y)) // Non-compliant - equivalent to the implicitly  
    // defined move constructor  
{  
}  
~A() // Non-compliant - equivalent to the implicitly defined destructor  
{  
}  
private:  
  std::int32_t x;  
  std::int32_t y;  
};  
class B  
{  
public:  
  B() {} // Non-compliant - x and y are not initialized  
    // should be replaced with: B() : x(0), y(0) {}  
  B(std::int32_t first, std::int32_t second) : x(first), y(second) {} //  
    Compliant  
  B(const B&) = default; // Compliant - equivalent to the copy constructor of class A  
  B(B&&) = default; // Compliant - equivalent to the move constructor of class A  
  ~B() = default; // Compliant - equivalent to the destructor of class A  
private:  
  std::int32_t x;  
  std::int32_t y;  
};  
class C  
{  
public:  
  C() = default; // Compliant  
  C(const C&) = default; // Compliant  
  C(C&&) = default; // Compliant  
};  
class D  
{  
public:  
  D() : ptr(nullptr) {} // Compliant - this is not equivalent to what the  
    // implicitly defined default constructor would do  
  D(C* p) : ptr(p) {} // Compliant  
  D(const D&) = default; // Shallow copy will be performed, user-defined copy  
    // constructor is needed to perform deep copy on ptr variable  
  D(D&&) = default; // ptr variable will be moved, so ptr will still point to  
    // the same object  
  ~D() = default; // ptr will not be deleted, the user-defined destructor is  
    // needed to delete allocated memory
private:
    C* ptr;
};

class E // Compliant - special member functions definitions are not needed as
    // class E uses only implicit definitions
{
};

See also

- HIC++ v4.0 [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

```cpp
#include <cstdint>
#include <utility>
class A
{
    public:
        // Implementation
        A(A const& oth) : x(oth.x) // Compliant
        {
        }
    private:
        std::int32_t x;
};
class B
```
public:
   // Implementation
   B(B&& oth) : ptr(std::move(oth.ptr)) // Compliant
   {
      oth.ptr = nullptr; // Compliant - this is not a side-effect, in this
      // case it is essential to leave moved-from object
      // in a valid state, otherwise double deletion will
      // occur.
   }
   ~B() { delete ptr; }
private:
   std::int32_t* ptr;
};

class C
{
   public:
      // Implementation
      C(C const& oth) : x(oth.x)
      {
         // ...
         x = x % 2; // Non-compliant - unrelated side-effect
      }
   private:
      std::int32_t x;
};

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
// $Id: A12-8-2.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
#include <cstdint>
#include <utility>
class A
{
  public:
    A(const A& oth) // ...
    {
    }
    A(A&& oth) noexcept // ...
    {
    }
    A& operator=(const A& oth) & // Compliant
    {
      A tmp(oth);
      swap(*this, tmp);
      return *this;
    }
    A& operator=(A&& oth) & noexcept // Compliant
    {
      A tmp(std::move(oth));
      swap(*this, tmp);
      return *this;
    }
    static void swap(A& lhs, A& rhs) noexcept
    {
      std::swap(lhs.ptr1, rhs.ptr1);
      std::swap(lhs.ptr2, rhs.ptr2);
    }
  private:
    std::int32_t* ptr1;
    std::int32_t* ptr2;
};

class B
{
  public:
    B& operator=(const B& oth) & // Non-compliant
    {
      if (this != &oth)
      {
        ptr1 = new std::int32_t(*oth.ptr1);
        ptr2 = new std::int32_t(*oth.ptr2); // Exception thrown here results in
        // a memory leak of ptr1
      }
      return *this;
    }
    B& operator=(B&& oth) & noexcept // Non-compliant
    {
      if (this != &oth)
private:
    std::int32_t* ptr1;
    std::int32_t* ptr2;
};

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**

```cpp
#include <cstdint>
#include <iostream>
#include <memory>
#include <string>

void f1()
{
    std::string s1 { "string" };
    std::string s2 { std::move (s1) };
    // ...
    std::cout << s1 << "\n";  // Non-compliant - s1 does not contain "string"
    // value after move operation
}

void f2()
{
    std::unique_ptr<std::int32_t> ptr1 = std::make_unique<std::int32_t>(0);
    std::unique_ptr<std::int32_t> ptr2 { std::move (ptr1) };
    std::cout << ptr1.get () << std::endl;  // Compliant by exception - move
    // construction of std::unique_ptr
    // leaves moved-from object in a
    // well-specified state
}
```

**See also**

- SEI CERT C++ [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

```cpp
// $Id: A12-8-4.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
#include <cstdint>
#include <string>
class A
{
    public:
        // ...
        A(A&& oth) :
            x(std::move(oth.x)), // Compliant
            s(std::move(oth.s)) // Compliant
        {
        }

    private:
        std::int32_t x;
        std::string s;
};
class B
{
    public:
        // ...
        B(B&& oth) :
            x(oth.x), // Compliant by exception, std::int32_t is of scalar
            s(oth.s) // Non-compliant
        {
        }

    private:
        std::int32_t x;
        std::string s;
};
class C
{
    public:
        // ...
        C(C&& oth) :
            x(oth.x), // Compliant by exception
            s(std::move(oth.s)) // Compliant
        {
        }

    private:
        std::int32_t x = 0;
```
std::string s = "Default string";

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**

```cpp
#include <cstdint>
#include <stdexcept>
struct A {
    std::int32_t number;
    std::int32_t* ptr;
    // Implementation
};

class B {
public:
    // ...
    B& operator=(B const& oth) // Non-compliant
    {
        i = oth.i;
        delete aPtr;
        try
        {
            aPtr = new A(*oth.aPtr); // If this is the self-copy case, then
            // the oth.a_ptr is already deleted
        }
        catch (std::bad_alloc&)
        {
```
```cpp
private:
    std::int16_t i = 0;
    A* aPtr = nullptr;
};

class C
{
    public:
        C& operator=(C const& oth) // Compliant
        {
            if (this != &oth)
            {
                A* tmpPtr = new A(*oth.aPtr);
                i = oth.i;
                delete aPtr;
                aPtr = tmpPtr;
            }
            return *this;
        }

        C& operator=(C&& oth) // Compliant
        {
            if (this != &oth)
            {
                A* tmpPtr = new A(std::move(*oth.aPtr));
                i = oth.i;
                delete aPtr;
                aPtr = tmpPtr;
            }
            return *this;
        }

    private:
        std::int16_t i = 0;
        A* aPtr = nullptr;
};

See also

- SEI CERT C++ [9]: OOP54-CPP Gracefully handle 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
```cpp
// $Id: A12-8-6.cpp 271927 2017-03-24 12:01:35Z piotr.tanski $
#include <memory>
#include <utility>
#include <vector>
class A // Abstract base class
{
public:
    A() = default;
    A(A const&) = default; // Non-compliant
    A(A&&) = default; // Non-compliant
    virtual ~A() = 0;
    A& operator=(A const&) = default; // Non-compliant
    A& operator=(A&&) = default; // Non-compliant
};
class B : public A
{
};
class C // Abstract base class
{
public:
    C() = default;
    virtual ~C() = 0;

protected:
    C(C const&) = default; // Compliant
    C(C&&) = default; // Compliant
    C& operator=(C const&) = default; // Compliant
    C& operator=(C&&) = default; // Compliant
};
class D : public C
{
};
class E // Abstract base class
{
public:
    E() = default;
    virtual ~E() = 0;
    E(E const&) = delete; // Compliant
    E(E&&) = delete; // Compliant
    E& operator=(E const&) = delete; // Compliant
```
E& operator=(E&&) = delete; // Compliant
}
class F : public E
{
class G // Non-abstract base class
{
  public:
  G() = default;
  virtual ~G() = default;
  G(G const&) = default; // Non-compliant
  G(G&&) = default; // Non-compliant
  G& operator=(G const&) = default; // Non-compliant
  G& operator=(G&&) = default; // Non-compliant
};
class H : public G
{
void fn1() noexcept
{
  B obj1;
  B obj2;
  A* ptr1 = &obj1;
  A* ptr2 = &obj2;
  *ptr1 = *ptr2; // Partial assignment only
  *ptr1 = std::move(*ptr2); // Partial move only
  D obj3;
  D obj4;
  C* ptr3 = &obj3;
  C* ptr4 = &obj4;
  *ptr3 = *ptr4; // Compilation error - copy assignment operator of class C
  // is protected
  *ptr3 = std::move(*ptr4); // Compilation error - move assignment operator
  // of class C is protected
  F obj5;
  F obj6;
  E* ptr5 = &obj5;
  E* ptr6 = &obj6;
  *ptr5 = *ptr6; // Compilation error - use of deleted copy assignment
  // operator
  *ptr5 = std::move(*ptr6); // Compilation error - use of deleted move
  // assignment operator
  H obj7;
  H obj8;
  G* ptr7 = &obj7;
  G* ptr8 = &obj8;
  *ptr7 = *ptr8; // Partial assignment only
  *ptr7 = std::move(*ptr8); // Partial move only
};
class I // Non-abstract base class
{
public:
I() = default;
~I() = default;

protected:
I(I const&) = default;  // Compliant
I(I&&) = default;       // Compliant
I& operator=(I const&) = default; // Compliant
I& operator=(I&&) = default;   // Compliant
};

class J : public I
{
public:
J() = default;
~J() = default;
J(J const&) = default;
J(J&&) = default;
J& operator=(J const&) = default;
J& operator=(J&&) = default;
};

void fn2() noexcept
{
std::vector<I> v1;
    // v1.push_back(J()); // Compilation-error on calling a deleted move
    // constructor of I class, slicing does not occur
    // v1.push_back(I()); // Compilation-error on calling a deleted move
    // constructor of I class

std::vector<J> v2;
v2.push_back(J()); // No compilation error

std::vector<std::unique_ptr<I>> v3;
v3.push_back(std::unique_ptr<I>{}); // No compilation error
v3.push_back(std::make_unique<I>()); // No compilation error
v3.push_back(std::make_unique<J>()); // No compilation error
v3.emplace_back();                  // No compilation error
}

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

```cpp
#include <cstdint>
class A {
public:
  A() = default;
  A& operator*=(std::int32_t i) // Non-compliant
  {
    // ...
    return *this;
  }
};
A f1() noexcept {
  return A{};
}
class B {
public:
  B() = default;
  B& operator*=(std::int32_t) & // Compliant
  {
    // ...
    return *this;
  }
};
B f2() noexcept {
  return B{};
}
std::int32_t f3() noexcept {
  return 1;
}
int main(int, char**) {
  f1() *= 10; // Temporary result of f1() multiplied by 10. No compile-time error.
  
  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 <cstddef> 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
**Guidelines for the use of the C++14 language in critical and safety-related systems**

**AUTOSAR AP Release 17-03**

```cpp
#include <cstdint>

constexpr unsigned long long int operator""_ft(unsigned long long int feet) // Non-compliant
{
    // Implementation
    return feet;
}

custom std::uint64_t operator""_m(std::uint64_t meters) // Non-compliant
{
    // Implementation
    return meters;
}

void fn() noexcept
{
    unsigned long long int feets = 200_ft;
    // std::uint64_t meters = 300_m;  // Compilation error - uint64_t has
    // invalid
    // type, not compatible with user-defined literals. On 64-bit machines,
    // uint64_t is defined as unsigned long, and not unsigned long long
}
```

**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**

```cpp
constexpr long double operator""_m(long double meters) // Compliant
{
    // Implementation
    return meters;
}

custom long double operator""_kg(long double kilograms) // Compliant
{
    // Implementation
    return kilograms;
}

custom long double operator"" m(long double meters) // Non-compliant
{
    // Implementation
    return meters;
}
```
constexpr long double operator"" kilograms(long double kilograms) // Non-compliant
{
    // Implementation
    return kilograms;
}

void fn()
{
    long double weight = 20.0_kg;
    long double distance = 204.8_m;
}

See also
• none

Rule A13-1-3 (required, implementation, automated)
User defined literals operators shall only perform conversion of passed parameters.

Rationale
It is expected behavior that the user-defined literals operators are only used to convert passed parameters to the type of declared return value. User-defined literals are not supposed to provide any other side-effects.

Example
 // $Id: A13-1-3.cpp 271927 2017-03-24 12:01:35Z piotr.tanski $
#include <cstdint>
#include <iostream>
struct Cube
{
    unsigned long long int volume;
    constexpr explicit Cube(unsigned long long int v) : volume(v) {} 
};
constexpr Cube operator"" _m3(unsigned long long int volume)
{
    return Cube(volume); // Compliant
}
struct Temperature
{
    unsigned long long int kelvins;
    constexpr explicit Temperature(unsigned long long int k) : kelvins(k) {} 
};
constexpr Temperature operator"" _K(unsigned long long int kelvins)
{
    return Temperature(kelvins); // Compliant
}
static void sumDistances(std::int32_t distance)
static std::int32_t overallDistance = 0;
overallDistance += distance;
}

struct Distance
{
    long double kilometers;
    explicit Distance(long double kms) : kilometers(kms) {}
};
Distance operator""_m(long double meters)
{
    sumDistances(meters); // Non-compliant - function has a side-effect
    return Distance(meters / 1000);
}
void operator""_print(const char* str)
{
    std::cout << str << '
'; // Non-compliant - user-defined literal operator
    // does not perform conversion and has a
    // side-effect
}

See also

• none

6.13.2 Declaration matching

Rule A13-2-1 (required, implementation, automated)
An assignment operator shall return a reference to “this”.

Rationale

Returning a type “T&” from an assignment operator is consistent with the C++ Standard Library.

Example

// $Id: A13-2-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
class A
{
    public:
        // ...
        A& operator=(const A&) & // Compliant
        {
            // ...
            return *this;
        }
};
class B
{
public:
  // ...
const B& operator=(const B&) & // Non-compliant - violating consistency
    // with standard types
{
  // ...
  return *this;
}

class C
{
  public:
  // ...
  C operator=(const C&) & // Non-compliant
  {
    // ...
    return *this;
  }
};

class D
{
  public:
  // ...
  D* operator=(const D&) & // Non-compliant
  {
    // ...
    return this;
  }
};

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**

// $Id: A13-2-2.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
Guidelines for the use of the C++14 language in critical and safety-related systems

AUTOSAR AP Release 17-03

#include <cstdint>

class A {
};

A operator+(A const&, A const&) noexcept // Compliant
{
    return A{};
}

std::int32_t operator/(A const&, A const&) noexcept // Compliant
{
    return 0;
}

A operator&(A const&, A const&)noexcept // Compliant
{
    return A{};
}

const A operator-(A const&, std::int32_t) noexcept // Non-compliant
{
    return A{};
}

A* operator|(A const&, A const&) noexcept // Non-compliant
{
    return new A{};
}

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
#include <cstdint>

class A {
};

bool operator==(A const&, A const&) noexcept // Compliant
{
    return true;
}
bool operator<(A const& lhs, A const& rhs) noexcept // Compliant
{
  return false;
}

bool operator!=(A const& lhs, A const& rhs) noexcept // Compliant
{
  return !(operator==(lhs, rhs));
}

std::int32_t operator>(A const&, A const&) noexcept // Non-compliant
{
  return -1;
}

A operator>=(A const& lhs, A const&) noexcept // Non-compliant
{
  return A{};
}

const A& operator<=(A const& lhs, A const& rhs) noexcept // Non-compliant
{
  return lhs;
}

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

```cpp
#include <cstdint>

template <typename T>
void f1(T&& t) noexcept(false)
```
Guidelines for the use of the C++14 language in critical and safety-related systems
AUTOSAR AP Release 17-03

```cpp
void f1(
    std::int32_t&& t) noexcept // Non-compliant - overloading a function with
    // forwarding reference
{
}

template <typename T>
void f2(T&& t) noexcept(false)
{
}

void f2(std::int32_t&) = delete; // Compliant by exception

int main(int, char**)
{
    std::int32_t x = 0;
    f1(x); // Calls f1(T&&) with T = int&
    f1(+x); // Calls f1(std::int32_t&&)
    f1(0); // Calls f1(std::int32_t&&)
    f1(0U); // Calls f1(T&&) with T = unsigned int
    f2(0); // Calls f2(T&&) with T = int
    // f2(x); // Compilation error, the overloading function is deleted
}
```

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**
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

```cpp
void fn() noexcept {
  std::uint32_t decimal1 = 3’000’000; // Compliant
  std::uint32_t decimal2 = 4’500; // Compliant
  std::uint32_t decimal3 = 54’00’30; // Non-compliant
  float decimal4 = 3.141’592’653; // Compliant
  float decimal5 = 3.1’4159’265’3; // Non-compliant
  std::uint32_t hex1 = 0xFF'FF'FF'FF; // Compliant
  std::uint32_t hex2 = 0xFAB'1'FFFFF; // Non-compliant
  std::uint8_t binary1 = 0b1001'0011; // Compliant
  std::uint8_t binary2 = 0b10'00'10'01; // Non-compliant
}
```

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

```cpp
// $Id: A14-1-1.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
#include <utility>
class A {
    public:
        A() = default;
        ~A() = default;
        A(A const&) = delete;
        A& operator=(A const&) = delete;
        A(A&&) = delete;
        A& operator=(A&&) = delete;
};
class B {
    public:
        B() = default;
        B(B const&) = default;
        B& operator=(B const&) = default;
        B(B&&) = default;
        B& operator=(B&&) = default;
};
template <typename T>
void f1(T const& obj) noexcept(false) {
    static_assert(
        std::is_copy_constructible<T>(),
        "Given template type is not copy constructible."); // Compliant
}
template <typename T>
class C {
    // Compliant
    static_assert(std::is_trivially_copy_constructible<T>(),
                  "Given template type is not trivially copy constructible.");
    // Compliant
    static_assert(std::is_trivially_move_constructible<T>(),
                  "Given template type is not trivially move constructible.");
    // Compliant
```
```cpp
class C
{
public:
    C() = default;
    C(C const&) = default;
    C& operator=(C const&) = default;
    C(C&&) = default;
    C& operator=(C&&) = default;

private:
    T c;
};

template <typename T>
class D
{
public:
    D() = default;
    D(D const&) = default; // Non-compliant - T may not be copyable
    D& operator=(D const&) = default; // Non-compliant - T may not be copyable
    D(D&) = default; // Non-compliant - T may not be movable
    D& operator=(D&) = default; // Non-compliant - T may not be movable

private:
    T d;
};

void f2() noexcept
{
    A a;
    B b;
    // f1<A>(a); // Class A is not copy constructible, compile-time error
    // occurs
    f1<B>(b); // Class B is copy constructible
    // C<A> c1; // Class A can not be used for template class C, compile-time
    // error occurs
    C<B> c2; // Class B can be used for template class C
    D<A> d1;
    // D<A> d2 = d1; // Class D can not be copied, because class A is not
    // copyable, compile-time error occurs
    // D<A> d3 = std::move(d1); // Class D can not be moved, because class A is
    // not movable, compile-time error occurs
    D<B> d4;
    D<B> d5 = d4;
    D<B> d6 = std::move(d4);
}
```

See also

---

AUTOSAR CONFIDENTIAL —

Document ID 839: AUTOSAR_RS_CPP14Guidelines

178 of 371
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

```cpp
// $Id: A14-7-1.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
#include <cstdint>
```
```cpp
class A {
  public:
    void setProperty(std::int32_t x) noexcept { property = x; }
    void doSomething() noexcept {} 

  private:
    std::int32_t property;
};

struct B {
};

class C {
  public:
    void doSomething() noexcept {
    }
};

template <typename T>

class D {
  public:
    void f1() {}
    void f2() {
      T t;
      t.setProperty(0);
    }
    void f3() {
      T t;
      t.doSomething();
    }
};

void fn() noexcept {
  D<A> d1;  // Compliant - struct A provides all needed members
d1.f1();
d1.f2();
d1.f3();

  D<B> d2;  // Non-compliant - struct B does not provide needed members
d2.f1();
  // d2.f2();  // Compilation error - no ’property’ in struct B
  // d2.f3();  // Compilation error - no member named ’doSomething’ in struct
    // B

  D<C> d3;  // Non-compliant - struct C does not provide property
d3.f1();
  // d3.f2();  // Compilation error - no property in struct C
  d3.f3();
```
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 <cstdint>
3 template <typename T>
4 void f1(T t)
5 {
6     // Implementation
7 }
8 void f1(std::int16_t n)
void fn1() noexcept
{
  std::int16_t x = 0;
  f1(x); // f1(std::int16_t) is called
  f1(x + 10); // f1(T) is called with T = int
  f1<>(x); // explicit call to f1(T) with T = short int
  f1<>(x + 10); // explicit call to f1(T) with T = int
}

template <typename T>
void f2(T t)
{
  // Implementation
}

template <>
void f2<std::int16_t>(std::int16_t n)
{
  // Implementation
}

void fn2() noexcept
{
  std::int16_t x = 0;
  f2(x); // f2(std::int16_t) is called
  f2(x + 10); // f2(T) is called with T = int
  f2<>(x); // explicit call to f2<T>(std::int16_t>
  f2<>(x + 10); // explicit call to f2<T> with T = int
  f2<std::int16_t>(x +
    10); // explicit call to f2<T>(std::int16_t>
}

void f3(std::int16_t n)
{
  // Implementation
}

void fn3() noexcept
{
  std::int16_t x = 0;
  f3(x); // f3(std::int16_t) is called
  f3(x + 10); // f3(std::int32_t) is called
}

template <typename T>
void f4(T t)

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
<table>
<thead>
<tr>
<th>Correctness of the exception handling</th>
<th>Exception handling mechanism is implemented by the compiler (by its library functions and machine code generator) and defined by the C++ Language Standard. Rule A1-2-1 requires that the compiler (including its exception handling routines), when used for safety-related software, meets appropriate safety requirements.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hidden control flow</td>
<td>ISO 26262-6 (Table *) recommends &quot;no hidden data flow or control flow&quot; for ASIL A software and highly recommends it for ASIL B/C/D. Therefore, the Rule A15-0-1 prohibits the usage of exceptions for normal control flow of software - they are allowed only for errors where a function failed to perform its assigned task.</td>
</tr>
<tr>
<td>Additional exit point from functions</td>
<td>ISO 26262-6 (Table *) highly recommends &quot;one entry and one exit point in subprograms and functions&quot; for ASIL A software. Therefore, the Rule A15-0-1 prohibits the usage of exceptions for normal control flow of software - they are allowed only for errors where a function failed to perform its assigned task. 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.</td>
</tr>
<tr>
<td>Code readability</td>
<td>If exceptions are used correctly, in particularly by using checked and unchecked exception types, see Rules: A15-0-4 and A15-0-5, the code is easier to read and maintain than if using error codes. It avoids nesting if/else error-checking statements.</td>
</tr>
<tr>
<td>Exception safety and program state consistency after exception is thrown</td>
<td>The Rule A15-0-2 requires that functions provide at least “basic exception safety” (Note: this C++ term is not related to functional safety)</td>
</tr>
<tr>
<td>Impact on runtime performance</td>
<td>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.</td>
</tr>
<tr>
<td>Impact on worst-case execution time</td>
<td>The A15-0-7 rule requires that the exception handling mechanism provides real-time implementation. Note that this is not the case for e.g. GCC compiler that allocates dynamic memory on throwing an exception. However, it is possible to fix it simply by avoiding memory allocation.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Tool support</th>
<th>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).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appropriate usage of exceptions in implementation</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 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
“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

```cpp
//% $Id: A15-0-1.cpp 271752 2017-03-23 12:07:07Z piotr.tanski $
#include <iostream>
#include <stdexcept>
#include <string>
#include <vector>
std::uint8_t computeCrc(std::string& msg);
bool isMessageCrcCorrect1(std::string& message)
{
    std::uint8_t computedCRC = computeCrc(message);
    std::uint8_t receivedCRC = message.at(0);
    if (computedCRC != receivedCRC)
    {
        throw std::logic_error("Computed CRC is invalid."); // Non-compliant - CheckMessageCRC()
        // was able to perform
        // its task, nothing exceptional about its invalid result
    }
}
```
bool isMessageCrcCorrect2(std::string& message)
{
    bool isCorrect = true;
    std::uint8_t computedCRC = computeCrc(message);
    std::uint8_t receivedCRC = message.at(0);
    if (computedCRC != receivedCRC)
    {
        isCorrect = false; // Compliant - if CRC is not correct, then return "false"
    }
    return isCorrect;
}

void sendData(std::string message)
{
    if (message.empty())
    {
        throw std::logic_error("Preconditions are not met."); // Compliant -
        // SendData() was
        // not able to
        // perform its
        // task
    }
}

bool sendTimeoutReached = false;

// Implementation
if (sendTimeoutReached)
{
    throw std::runtime_error("Timeout on sending a message has been reached."); // Compliant -
    // SendData() did not
    // perform its
    // task
}

std::int32_t findIndex(std::vector<std::int32_t>& v, std::int32_t x) noexcept
{
    try
    {
        std::size_t size = v.size();
        for (std::size_t i = 0U; i < size; ++i)
        {
            if (v.at(i) == x) // v.at() throws an std::out_of_range exception
            {
            }
throw i; // Non-compliant - nothing exceptional about finding a
// value in vector
}

} // Non-compliant - nothing exceptional about finding a
// value in vector
}

} // Non-compliant - nothing exceptional about finding a
// value in vector
}

} // Compliant - std::out_of_range error shall be handled
{
    return -1;
}

return -1;

bool readFile(std::string& filename) noexcept
{
    try
    {
        std::ifstream file(filename, std::ios_base::in);
        if (!file.is_open())
        {
            throw std::runtime_error(
                "File cannot be opened"); // Compliant - error on opening a
    // file is an exceptional case
        }

char c = file.get();

if (!file.good())
{
    throw std::runtime_error(
        "Cannot read from file"); // Compliant - error on reading from
    // file is an exceptional case
}

return true;
}
void fn1(
    std::uint32_t x) // Non-compliant - inefficient and less readable version
    // than its obvious alternative, e.g. fn2()

// function
{
    try
    {
        if (x < 10)
        {
            throw 10;
        }
        // Action "A"
    }
    catch (std::int32_t y)
    {
        // Action "B"
    }
}

void fn2(
    std::uint32_t x) // Compliant - the same functionality as fn1() function
{
    if (x < 10)
    {
        // Action "B"
    }
    else
    {
        // Action "A"
    }
}

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

```cpp
//% $Id: A15-0-2.cpp 271927 2017-03-24 12:01:35Z piotr.tanski $
#include <cstdint>
#include <cstring>
class C1
{
    public:
        C1(const C1& rhs)
        {
            copyBad(rhs); // Non-compliant - if an exception is thrown, an object
            // will be left in an indeterminate state
            copyGood(rhs); // Compliant - full object will be properly copied or
            // none of its properties will be changed
        }
        ~C1() { delete[] e; }
    void copyBad(const C1& rhs)
    {
        if (this != &rhs)
        {
```
```cpp
delete[] e;
e = nullptr; // e changed before the block where an exception can
// be thrown
s = rhs.s; // s changed before the block where an exception can be
// thrown

if (s > 0)
{
    e = new std::int32_t[s]; // If an exception will be thrown
    // here, the
    // object will be left in an indeterminate
    // state
    std::memcpy(e, rhs.e, s * sizeof(std::int32_t));
}

void copyGood(const C1& rhs)
{
    std::int32_t* eTmp = nullptr;
    if (rhs.s > 0)
    {
        eTmp = new std::int32_t[rhs.s]; // If an exception will be thrown
        // here, the
        // object will be left unchanged
        std::memcpy(eTmp, rhs.e, rhs.s * sizeof(std::int32_t));
    }

    delete[] e;
e = eTmp;
s = rhs.s;
}

private:
    std::int32_t* e;
    std::size_t s;
};

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

class C2
{
    public:
        C2() : a1(new A), a2(new A) // Non-compliant - if a2 memory allocation
        // fails, a1 will never be deallocated
        {
    }

    private:
```
A* a1;
A* a2;
};
class C3
{
   public:
      C3() : a1(nullptr), a2(nullptr) // Compliant
      {
         try
         {
            a1 = new A;
            a2 = new A; // If memory allocation for a2 fails, catch-block will
                         // deallocate a1
         }
         catch (...)
         {
            delete a1;
            a1 = nullptr;
            delete a2;
            a2 = nullptr;
            throw;
         }
      }
   private:
      A* a1;
      A* a2;
};

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
Suppling 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
//% $Id: A15-0-3.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
#include <cstdint>
#include <stdexcept>
#include <vector>

class A {
    public:
        explicit A(std::int32_t value) noexcept(false) : x(value) {
            if (x == 0) {
                throw std::invalid_argument("Constructor: Invalid Argument");
            }
        }

    private:
        std::int32_t x;
};

int main(int, char**)
{
    constexpr std::int32_t limit = 10;
    std::vector<A> vec1; // Constructor and assignment operator of A class
        // throw exceptions

    try {
        for (std::int32_t i = 1; i < limit; ++i) {
            vec1.push_back(A(i)); // Constructor of A class will not throw for
        // value from 1 to 10
        }

        vec1.emplace(vec1.begin(),
            0); // Non-compliant - constructor A(0) throws in an
        // emplace() method of std::vector. This leads to
        // unexpected result of emplace() method. Throwing an
        // exception inside an object constructor in emplace()
        // leads to duplication of one of vector’s elements.
        // Vector invariants are valid and the object is destructible.
        }
    catch (std::invalid_argument& e) {
        // Handle an exception
        }

    std::vector<A> vec2;
    vec2.reserve(limit);
    try {
        for (std::int32_t i = limit - 1; i >= 0; --i) {
            vec2.push_back(A(i)); // Compliant - constructor of A(0) throws for
        // i = 0, but in this case strong exception
        // safety is guaranteed. While push_back()
// offers strong exception safety guarantee,
// push_back can only succeed to add a new
// element or fails and does not change the
// container

}  
}

catch (std::invalid_argument& e)
{
  // Handle an exception
}

return 0;
}

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

```cpp
#include <cstdint>
#include <stdexcept>
class InvalidArguments : public std::logic_error // Compliant - invalid
  // arguments error is
  // "unchecked" exception
{
  public:
    using std::logic_error::logic_error;
};
class OutOfMemory : public std::bad_alloc // Compliant - insufficient memory
  // error is "unchecked" exception
```
public:
    using std::bad_alloc::bad_alloc;
};
class DivisionByZero : public std::logic_error // Compliant - division by zero
    // error is "unchecked"
    // exception
{
    public:
        using std::logic_error::logic_error;
    };
class CommunicationError : public std::logic_error // Non-compliant -
    // communication error
    // should be "checked"
    // exception but defined to be "unchecked"
{
    public:
        using std::logic_error::logic_error;
    };

double division(std::int32_t a, std::int32_t b) noexcept(false)
{
    // ...
    if (b == 0)
    {
        throw DivisionByZero(
            "Division by zero error"); // Unchecked exception thrown correctly
    }
    // ...
}
void allocate(std::uint32_t bytes) noexcept(false)
{
    // ...
    throw OutOfMemory(); // Unchecked exception thrown correctly
}
void initializeSocket() noexcept(false)
{
    bool validParameters = true;
    // ...
    if (!validParameters)
    {
        throw InvalidArguments("Invalid parameters passed"); // Unchecked
        // exception
        // thrown
        // correctly
    }
}
void sendData(std::int32_t socket) noexcept(false)
{
    bool isSentSuccessfully = true;
    // ...
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**

```cpp
#include <cstdint>
#include <stdexcept>

class CommunicationError : public std::exception // Compliant - communication error is "checked"
{
public:
    explicit CommunicationError(const char* message) : msg(message) {}
    CommunicationError(CommunicationError const&) noexcept = default;
    // ...}
    if (!isSentSuccessfully)
    {
        throw CommunicationError("Could not send data"); // Unchecked exception
        // thrown when checked
        // exception should
        // be.
    }
}
CommunicationError& operator=(CommunicationError const&) noexcept = default;
~CommunicationError() override = default;

const char* what() const noexcept override { return msg; }

private:
    const char* msg;
};

class BusError
    : public CommunicationError // Compliant - bus error is "checked"
{
public:
    using CommunicationError::CommunicationError;
};

class Timeout : public std::runtime_error // Compliant - communication timeout
    // is "checked"
{
public:
    using std::runtime_error::runtime_error;
};

class PreconditionsError : public std::exception // Non-compliant - error on
    // preconditions check should
    // be "unchecked" but is
    // defined to be "checked"
{
    // Implementation
};

void fn1(std::uint8_t* buffer, std::uint8_t bufferLength) noexcept(false)
{
    bool sentSuccessfully = true;

    // ...
    if (!sentSuccessfully)
    {
        throw CommunicationError(
            "Could not send data"); // Checked exception thrown correctly
    }
}

void fn2(std::uint8_t* buffer, std::uint8_t bufferLength) noexcept(false)
{
    bool initSuccessfully = true;

    if (!initSuccessfully)
    {
        throw PreconditionsError(); // An exception thrown on preconditions
        // check failure should be "Unchecked", but
        // PreconditionsError is "Checked"
    }

    // ...
    bool sentSuccessfully = true;
bool isTimeout = false;

// ...
if (!sentSuccessfully)
{
    throw BusError("Could not send data"); // Checked exception thrown correctly
}

// ...
if (isTimeout)
{
    throw Timeout("Timeout reached"); // Checked exception thrown correctly
}

void fn3(std::uint8_t* buffer) noexcept(false)
{
    bool isResourceBusy = false;

    // ...
    if (isResourceBusy)
    {
        throw std::runtime_error("Resource is busy now"); // Checked exception thrown correctly
    }
}

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
Guidelines for the use of the C++14 language in critical and safety-related systems

AUTOSAR AP Release 17-03

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

```cpp
#include <cstdlib>
#include <cstring>
struct __cxa_exception {
    // Exception’s structure implementation
};
extern "C" void fatalError(const char* msg) {
    // Reports an error and terminates the program
}

extern "C" void* __cxa_allocate_exception_dynamically(size_t thrownSize) {
    size_t size = thrownSize + sizeof(__cxa_exception);
    __cxa_exception* buffer = static_cast<__cxa_exception*>(malloc(size)); // Non-compliant - dynamic memory allocation used
    if (!buffer)
```
fatalError("Not enough memory to allocate exception!");
}

memset(buffer, 0, sizeof(__cxa_exception));
return buffer + 1;
}
}

extern "C" void* static_malloc(size_t size)
{
void* mem = NULL;
// Allocates memory using static memory pool
return mem;
}

extern "C" void* __cxa_allocate_exception_statically(size_t thrownSize)
{
size_t size = thrownSize + sizeof(__cxa_exception);
__cxa_exception* buffer = static_cast<__cxa_exception*>(static_malloc(size)); // Compliant - memory allocation on static memory pool used
if (!buffer)
{
fatalError("Not enough memory to allocate exception!");
}
memset(buffer, 0, sizeof(__cxa_exception));
return buffer + 1;
}

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.
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 an unified way. Also, "it makes your code easier to learn and re-use, because it matches established conventions with which programmers are already familiar.". [Effective Java 2nd Edition [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**

```cpp
#include <memory>
#include <stdexcept>
class A {
    // Implementation
};
class MyException : public std::logic_error {
public:
    using std::logic_error::logic_error;
    // Implementation
};
void f1() {
    throw -1; // Non-compliant - integer literal thrown
}
void f2() {
    throw nullptr; // Non-compliant - null-pointer-constant thrown
}
void f3() {
    throw A(); // Non-compliant - user-defined type that does not inherit from
              // std::exception thrown
}
void f4()
```
28 {  
29     throw std::logic_error{  
30         "Logic Error"};  // Compliant – std library exception thrown
31 }
32 
33 void f5()
34 {
35     throw MyException("Logic Error");  // Compliant – user-defined type that
36         // inherits from std::exception thrown
37 }
38 
39 void f6()
40 {
41     throw std::make_shared<std::exception>((
42         std::logic_error("Logic Error"));  // Non-compliant – shared_ptr does
43             // not inherit from std::exception
44 }
45 
46 void f7()
47 {
48     try
49     {
50         f6();
51     }
52     catch (std::exception& e)  // An exception of
53             // std::shared_ptr<std::exception> type will not
54             // be caught here
55     {
56         // Handle an exception
57     }
58     catch (std::shared_ptr<std::exception>& e)  // An exception of
59             // std::shared_ptr<std::exception> type will be caught here, but
60             // unable to access
61             // std::logic_error information
62     {
63         // Handle an exception
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

```cpp
#include <cstdint>

class A {
  // Implementation
};

void fn(std::int16_t i) {
  A a1;
  A& a2 = a1;
  A* a3 = new A;
  if (i < 10) {
    throw a1; // Compliant - copyable object thrown
  }
  else if (i < 20) {
    throw A(); // Compliant - copyable object thrown
  }
  else if (i < 30) {
    throw a2; // Compliant - copyable object thrown
  }
  else if (i < 40) {
    throw & a1; // Non-compliant - pointer type thrown
  }
  else if (i < 50) {
    throw a3; // Non-compliant - pointer type thrown
  }
  else if (i < 60) {
    throw(*a3); // Compliant - memory leak occurs, violates other rules
  }
```
\begin{verbatim}
}  
else
{
    throw new A; // Non-compliant - pointer type thrown
}
\end{verbatim}

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

```cpp
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("Error"); // Compliant - exception type is unique
30 }
31 void f4() noexcept(false)
32 {
33     // ...
34     throw std::logic_error("f3(): preconditions were not met"); // Compliant -
35       // error
36       // message is
37       // unique
38 }
39 void f5() noexcept(false)
40 {
41     // ...
42     throw std::logic_error(composeMessage(
43         __FILE__,
44         __func__,
45         __LINE__,
46         "Error");
47 }
```

Example 2

```cpp
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("Error"); // Compliant - exception type is unique
30 }
31 void f4() noexcept(false)
32 {
33     // ...
34     throw std::logic_error("f3(): preconditions were not met"); // Compliant -
35       // error
36       // message is
37       // unique
38 }
39 void f5() noexcept(false)
40 {
41     // ...
42     throw std::logic_error(composeMessage(
43         __FILE__,
44         __func__,
45         __LINE__,
46         "Error");
47 }
```
"postconditions were not met"); // Compliant - error message is unique
}

void f6() noexcept
{
    try
    {
        f1();
        f2();
        f3();
    }
    catch (std::invalid_argument& e)
    {
        std::cout << e.what() << '\n'; // Only f3() throws this type of
        // exception, it is easy to deduce which
        // function threw
    }
    catch (std::logic_error& e)
    {
        std::cout << e.what() << '\n'; // f1() and f2() throw exactly the same
        // exceptions, unable to deduce which
        // function threw
    }
    try
    {
        f4();
        f5();
    }
    catch (std::logic_error& e)
    {
        std::cout << e.what() << '\n'; // Debugging process simplified, because
        // of unique error message it is known
        // which function threw
    }
}

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

```cpp
#include <cstdint>
#include <memory>
#include <stdexcept>
extern std::uint32_t f1();
void fVeryBad() noexcept(false)
{
    std::logic_error* e = new std::logic_error("Logic Error 1");
    // ...
    std::uint32_t i = f1();
    if (i < 10)
    {
        throw(*e); // Non-compliant - fVeryBad() is not able to clean-up allocated memory
        // ...
    }
    delete e;
}
void fBad() noexcept(false)
{
    std::int32_t* x = new std::int32_t(0);
    // ...
    std::uint32_t i = f1();
    if (i < 10)
    \
```
{  
    throw std::logic_error("Logic Error 2"); // Non-compliant - exits from  
    // fBad() without cleaning-up  
    // allocated resources and  
    // causes a memory leak  
}

else if (i < 20)  
{  
    throw std::runtime_error("Runtime Error 3"); // Non-compliant - exits  
    // from fBad() without  
    // cleaning-up allocated  
    // resources and causes a  
    // memory leak  
}

// ...
// Deallocates claimed resource only in the end of fBad() scope  
void fGood() noexcept(false)  
{  
    std::int32_t* y = new std::int32_t(0);  
    // ...
    std::uint32_t i = f1();  
    if (i < 10)  
    {  
        delete y; // Deletes allocated resource before throwing an exception  
        throw std::logic_error("Logic Error 4"); // Compliant - deleting y  
        // variable before exception  
        // leaves the fGood() scope  
    }  

    else if (i < 20)  
    {  
        delete y; // Deletes allocated resource before throwing an exception  
        throw std::runtime_error("Runtime Error 5"); // Compliant - deleting y  
        // variable before  
        // exception leaves the  
        // fGood() scope  
    }  

    else if (i < 30)  
    {  
        delete y; // Deletes allocated resource before throwing an exception  
        // again, difficult to maintain  
        throw std::invalid_argument(  
            "Invalid Argument 1"); // Compliant - deleting  
        // y variable before  
        // exception leaves the  
        // fGood() scope  
    }  
}
```cpp
79 }  
80 // ...
81 delete y; // Deallocates claimed resource also in the end of fGood() scope
82 }
83 void fBest() noexcept(false)
84 {
85 std::unique_ptr<std::int32_t> z = std::make_unique<std::int32_t>(0);
86 // ...
87 std::uint32_t i = f1();
88 if (i < 10)
89 {
90     throw std::logic_error("Logic Error 6"); // Compliant - leaving the
91         // fBest() scope causes
92         // deallocation of all
93         // automatic variables, unique_ptrs, too
94 }  
95 else if (i < 20)
96 {
97     throw std::runtime_error("Runtime Error 3"); // Compliant - leaving the
98         // fBest() scope causes
99         // deallocation of all
100        // automatic variables, unique_ptrs, too
101 }  
102 else if (i < 30)
103 {
104     throw std::invalid_argument(
105         "Invalid Argument 2"); // Compliant - leaving the fBest() scope
106         // causes deallocation of all automatic
107         // variables, unique_ptrs,
108         // too
109     }
110     }  
111     // ...
112     // z is deallocated automatically here, too
113 }
114 class CRaii // Simple class that follows RAII pattern
115 {
116     public:
117         CRaii(std::int32_t* pointer) noexcept : x(pointer) {}
118         ~CRaii()  
119         {  
120             delete x;
121             x = nullptr;
122         }
123     private:
124```
Guidelines for the use of the C++14 language in critical and safety-related systems

AUTOSAR AP Release 17-03

std::int32_t* x;

void fBest2() noexcept(false)
{
    CRaii a1(new std::int32_t(10));
    // ...
    std::uint32_t i = f1();
    if (i < 10)
    {
        throw std::logic_error("Logic Error 7"); // Compliant - leaving the
        // fBest2() scope causes a1
        // variable deallocation
        // automatically
    }
    else if (i < 20)
    {
        throw std::runtime_error("Runtime Error 4"); // Compliant - leaving the
        // fBest2() scope causes
        // a1 variable
        // deallocation
        // automatically
    }
    else if (i < 30)
    {
        throw std::invalid_argument("Invalid Argument 3"); // Compliant - leaving the fBest2() scope
        // causes a1 variable deallocation
        // automatically
    }
}

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**

```cpp
// A15-2-1.cpp 271927 2017-03-24 12:01:35Z piotr.tanski
#include <cstdint>
#include <stdexcept>

class A
{
  public:
    A() noexcept : x(0)
    {
    // ...
    }

  explicit A(std::int32_t n) : x(n)
  {
    // ...
    throw std::runtime_error("Unexpected error");
  }
};
```
A(std::int32_t i, std::int32_t j) noexcept : x(i + j) {
  try {
    // ...
    throw std::runtime_error("Error");
    // ...
  }
  catch (std::exception& e) {
  }
}

private:
std::int32_t x;

static A a1; // Compliant - default constructor of type A is noexcept
static A a2(5); // Non-compliant - constructor of type A throws, and the
  // exception will not be caught by the handler in main function
static A a3(5, 10); // Compliant - constructor of type A is noexcept, it
  // handles exceptions internally

int main(int, char**) {
  try {
    // program code
  } catch (...) {
    // Handle exceptions
  }
  return 0;
}

See also

- SEI CERT C++ [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

Guidelines for the use of the C++14 language in critical and safety-related systems
AUTOSAR AP Release 17-03

15
16 A(std::int32_t i, std::int32_t j) noexcept : x(i + j) {
17
  try {
18    // ...
19    throw std::runtime_error("Error");
20    // ...
21  }
22
23  catch (std::exception& e) {
24  }
25  // ...
26  }
27
28 private:
29  std::int32_t x;
30
31 static A a1; // Compliant - default constructor of type A is noexcept
32 static A a2(5); // Non-compliant - constructor of type A throws, and the
33  // exception will not be caught by the handler in main function
34 static A a3(5, 10); // Compliant - constructor of type A is noexcept, it
35  // handles exceptions internally
36
37 int main(int, char**) {
38  try {
39    // program code
40  }
41  catch (...) {
42    // Handle exceptions
43  }
44  return 0;
45  }
```cpp
#include <fstream>
#include <stdexcept>

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

class C1
{
    public:
        C1() noexcept(false)
            : a1(new A), a2(new A) // Non-compliant - if a2 memory allocation
            // fails, a1 will never be deallocated
        {
        }
        C1(A* pA1, A* pA2) noexcept : a1(pA1), a2(pA2) // Compliant - memory allocated outside of C1
            // constructor, and no exceptions can be thrown
        {
        }
    private:
        A* a1;
        A* a2;
};

class C2
{
    public:
        C2() noexcept(false) : a1(nullptr), a2(nullptr)
    {
        try
            {
                a1 = new A;
                a2 = new A; // If memory allocation for a2 fails, catch-block will
                // deallocate a1
            }
        catch (std::exception& e)
            {
                throw; // Non-compliant - whenever a2 allocation throws an
                // exception, a1 will never be deallocated
            }
    }
    private:
        A* a1;
        A* a2;
    };
```cpp
public:
    C3() noexcept(false) : a1(nullptr), a2(nullptr), file("./filename.txt")
    {
        try
        {
            a1 = new A;
            a2 = new A;

            if (!file.good())
            {
                throw std::runtime_error("Could not open file.");
            }
        }

        catch (std::exception& e)
        {
            delete a1;
            a1 = nullptr;
            delete a2;
            a2 = nullptr;
            file.close();
            throw; // Compliant - all resources are deallocated before the
            // constructor exits with an exception
        }
    }

private:
    A* a1;
    A* a2;
    std::ofstream file;
};

class C4
{
    public:
        C4() : x(0), y(0)
        {
            // Does not need to check preconditions here - x and y initialized with
            // correct values
        }
        C4(std::int32_t first, std::int32_t second) noexcept(false) : x(first), y(second)
        {
            checkPreconditions(x,
                                y); // Compliant - if constructor failed to create a
                                // valid object, then throw an exception
        }

        static void checkPreconditions(std::int32_t x,
                                        std::int32_t y) noexcept(false)
        {
            if (x < 0 || x > 1000)
```


```cpp
    {  
        throw std::invalid_argument(
            "Preconditions of class C4 were not met");  
    }

else if (y < 0 || y > 1000)
{
    throw std::invalid_argument(
        "Preconditions of class C4 were not met");
}

private:
   std::int32_t x; // Acceptable range: <0; 1000>
   std::int32_t y; // Acceptable range: <0; 1000>
```

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**

```cpp
// $Id: A15-3-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
```
```cpp
#include <cstdint>
#include <stdexcept>

class OutOfMemory : public std::logic_error // Unchecked exception
{
public:
    using std::logic_error::logic_error;
};

void f1() noexcept(false)
{
    // ...
    throw OutOfMemory("Not enough memory");
}

void f2() noexcept
{
    try
    {
        // ...
        f1();
    }
    catch (OutOfMemory& e) // Non-compliant - program is not able to handle an
                              // OutOfMemory error in a meaningful way, the error
                              // will still exist
    {
        // Handle an exception
    }
}

void f3() noexcept(false)
{
    // ...
    try
    {
        // ...
        f1();
    }
    catch (OutOfMemory& e)
    {
        // Nothing to do, just re-throw
        throw; // Non-compliant - it is inefficient to catch and re-throw an
                  // error that can not be handled in f3()
    }
}

void f4() noexcept(false)
{
    // ...
    f1(); // Compliant - OutOfMemory error can not be handled in f4()
    // ...
}

int main(int, char**)
{
    try
    {
        f4();
    }
    // ...
}
```
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
```cpp
//% $Id: A15-3-2.cpp 271752 2017-03-23 12:07:07Z piotr.tanski $
#include <cstdint>
#include <iostream>
#include <stdexcept>
class CommunicationError : public std::exception
{
    // Implementation
};
/// @throw CommunicationError Exceptional communication errors
extern void send(std::uint8_t* buffer) noexcept(false);
void sendData1(std::uint8_t* data) noexcept(false)
{
    try
    {
        send(data);
    }
    catch (CommunicationError& e)
    {
        std::cerr << "Communication error occurred" << std::endl;
        throw; // Non-compliant - exception is not handled, just re-thrown
    }
}
extern void busRestart() noexcept;
extern void bufferClean() noexcept;
void sendData2(std::uint8_t* data) noexcept(false)
```
{  
  try  
  {  
    send(data);  
  }  
  
  catch (CommunicationError& e)  
  {  
    std::cerr << "Communication error occurred" << std::endl;  
    bufferClean();  
    throw; // Compliant - exception is partially handled and re-thrown  
  }  
}  

void f1() noexcept  
{  
  std::uint8_t* buffer = nullptr;  
  
  // ...  
  try  
  {  
    sendData2(buffer);  
  }  
  
  catch (CommunicationError& e)  
  {  
    std::cerr << "Communication error occurred" << std::endl;  
    busRestart();  
    // Compliant - including sendData2() exception handler, exception is now  
    // fully handled  
  }  
}  

void sendData3(std::uint8_t* data) noexcept  
{  
  try  
  {  
    send(data);  
  }  
  
  catch (CommunicationError& e)  
  {  
    std::cerr << "Communication error occurred" << std::endl;  
    bufferClean();  
    busRestart();  
    // Compliant - exception is fully handled  
  }  
}  

See also

• none
Guidelines for the use of the C++14 language in critical and safety-related systems
AUTOSAR AP Release 17-03

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

```cpp
#include <stdexcept>

int mainGood(int, char**) // Compliant
{
    try
        // program code
    }
    catch (std::runtime_error& e)
    {
        // Handle runtime errors
    }
    catch (std::logic_error& e)
    {
        // Handle logic errors
    }
    catch (std::exception& e)
    {
        // Handle all expected exceptions
    }
    catch (...)
    {
        // Handle all unexpected exceptions
    }
    return 0;
}

int mainBad(int, char**) // Non-compliant - unexpected exceptions are not caught
{
    try
    {
```
// program code
}
catch (std::runtime_error& e)
{
    // Handle runtime errors
}
catch (std::logic_error& e)
{
    // Handle logic errors
}
catch (std::exception& e)
{
    // Handle all expected exceptions
}
return 0;
}

void threadMainGood() // Compliant
{
    try
    {
        // thread code
    }
catch (std::exception& e)
    {
        // Handle all expected exceptions
    }
catch (...)
    {
        // Handle all unexpected exception
    }

void threadMainBad() // Non-compliant - unexpected exceptions are not caught
{
    try
    {
        // thread code
    }
catch (std::exception& e)
    {
        // Handle all expected exceptions
    }

    // Uncaught unexpected exception will cause an immediate program termination

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
```cpp
//% $Id: A15-3-4.cpp 271752 2017-03-23 12:07:07Z piotr.tanski $
#include <stdexcept>
#include <thread>
extern std::int32_t fn(); // Prototype of external third-party library function
void f1() noexcept(false) {
    try {
        std::int32_t ret = fn();
        // ...
    }
    catch (...) { // Compliant
        // Handle all unexpected exceptions from fn() function
    }
}
void f2() noexcept(false) {
    std::int32_t ret = fn(); // Non-compliant - can not be sure whether fn() throws or not
    if (ret < 10) {
        throw std::underflow_error("Error");
    }
    else if (ret < 20) {
        // ...
    }
    else if (ret < 30) {
```
throw std::overflow_error("Error");
}

else
{
    throw std::range_error("Error");
}
void f3() noexcept(false)
{
    try
    {
        f2();
    }
    catch (std::exception& e) // Non-compliant - caught exception is too
    // general, no information which error occurred
    {
        // Nothing to do
        throw;
    }
}
void f4() noexcept(false)
{
    try
    {
        f3();
    }
    catch (...) // Non-compliant - no information about the exception
    {
        // Nothing to do
        throw;
    }
}
class ExecutionManager
{
public:
    ExecutionManager() = default;
    void execute() noexcept(false)
    {
        try
        {
            f3();
        }
        // ...
        catch (std::exception& e) // Compliant
        {
            // Handle all expected exceptions
        }
catch (...) // Compliant
{
    // Handle all unexpected exceptions
}
}

void threadMain() noexcept
{
    try
    {
        f3();
    }
    // ...
    catch (std::exception& e) // Compliant
    {
        // Handle all expected exceptions
    }
    catch (...) // Compliant
    {
        // Handle all unexpected exceptions
    }
}

int main(int, char**)  
{
    try
    {
        ExecutionManager execManager;
        execManager.execute();
        // ...
        std::thread t(&threadMain);
        // ...
        f4();
    }
    // ...
    catch (std::exception& e) // Compliant
    {
        // Handle all expected exceptions
    }
    catch (...) // Compliant
    {
        // Handle all unexpected exceptions
    }
    return 0;
}

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
```cpp
//% $Id: A15-3-5.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
#include <iostream>
#include <stdexcept>
class Exception : public std::runtime_error
{
public:
    using std::runtime_error::runtime_error;
    const char* what() const noexcept(true) override
    {
        return "Exception error message";
    }
};
void fn()
{
    try
    {
        // ...
        throw std::runtime_error("Error");
        // ...
        throw Exception("Error");
    }
    catch (const std::logic_error& e) // Compliant - caught by const reference
```
\{  
  // Handle exception
\}

\catch (std::runtime_error& e) // Compliant - caught by reference
\{
  std::cout << e.what() << "\n"; // "Error" or "Exception error message"
  // will be printed, depending upon the
  // actual type of thrown object
  throw e; // The exception re-thrown is of its original type
\}

\catch (  
  std::runtime_error
  e) // Non-compliant - derived types will be caught as the base type
\{
  std::cout
  << e.what()
  << "\n"; // Will always call what() method from std::runtime_error
  throw e; // The exception re-thrown is of the std::runtime_error type,
  // not the original exception type
\}

See also

- MISRA C++ 2008 [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

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

See also
- C++ Core Guidelines [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
```cpp
#include <stdexcept>

void libraryFunc();

void f1() noexcept
{
    // ...
    throw std::runtime_error("Error"); // Non-compliant - f1 declared to be noexcept, but exits with exception.
    // This leads to std::terminate() call
}

void f2() noexcept(true)
{
    try
    {
        // ...
        throw std::runtime_error("Error"); // Compliant - exception will not leave f2
    }
    catch (std::runtime_error& e)
    {
        // Handle runtime error
    }
}

void f3() noexcept(false)
{
    // ...
    throw std::runtime_error("Error"); // Compliant
}

void f4() noexcept(false) // Compliant - no information whether library_func() throws or not
{
    // ...
    libraryFunc();
}
```

See also 227 of 371
Guidelines for the use of the C++14 language in critical and safety-related systems

AUTOSAR AP Release 17-03

- 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 overrides.

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 overrides of a member function.

Example

```cpp
// % $Id: A15-4-3.cpp 270728 2017-03-16 10:38:20Z piotr.tanski $
// f1.hpp
void fn() noexcept;

// f1.cpp
#include <f1.hpp>
void fn() noexcept // Compliant
{
// Implementation
}

// f2.cpp
#include <f1.hpp>
void fn() noexcept(false) // Non-compliant - different exception specifier
{
// Implementation
}

class A
{
public:
    void f() noexcept;  
    void g() noexcept(false);
    virtual void v1() noexcept = 0;
    virtual void v2() noexcept(false) = 0;
};
void A::f() noexcept // Compliant
    // void A::f() noexcept(false) // Non-compliant - different exception specifier than in declaration
{
// Implementation
}

void A::g() noexcept(false) // Compliant
    // void A::g() noexcept // Non-compliant - different exception specifier than
    // in declaration
{  
```
class B : public A
{
    public:
        void v1() noexcept override // Compliant
        // void v1() noexcept(false) override // Non-compliant - looser exception
        // specifier in derived method,
        // non-compilable
        {
            // Implementation
        }
        void v2() noexcept override // Compliant
        {
            // Implementation
        }
};

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
void f2() noexcept; // Compliant - f2 does not throw exceptions
void f3() noexcept(true); // Compliant - f3 does not throw exceptions
void f4() noexcept(false); // Compliant - f4 declares to throw exceptions
void f5() noexcept // Compliant - f5 does not throw exceptions
{
    try {
        f1(); // Exception handling needed, f1 has no noexcept specification
    }
    catch (std::exception& e) {
        // Handle exceptions
    }
    f2(); // Exception handling not needed, f2 is noexcept
    f3(); // Exception handling not needed, f3 is noexcept(true)
    try {
        f4(); // Exception handling needed, f4 is noexcept(false)
    }
    catch (std::exception& e) {
        // Handle exceptions
    }
} //...
class C2
{
  public:
  C2() // Compliant - constructor of class C2 throws exceptions
  {
  }
  // ...
};

void f7() noexcept // Compliant - f7 does not throw exceptions
{
  std::cout << noexcept(A<C1>()) << 'n'; // prints '1' - constructor of
  // A<C1> class is noexcept(true)
  // because constructor of C1 class
  // is declared to be noexcept(true)
  std::cout << noexcept(A<C2>()) << 'n'; // prints '0' - constructor of
  // A<C2> class is noexcept(false)
  // because constructor of C2 class
  // has no noexcept specifier
}

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
  //% $Id: A15-4-5.cpp 271752 2017-03-23 12:07:07Z piotr.tanski $
```cpp
#include <cstdint>
#include <stdexcept>

class CommunicationError : public std::exception
{
    // Implementation
};

class BusError : public CommunicationError
{
    // Implementation
};
class Timeout : public std::runtime_error
{
    public:
        using std::runtime_error::runtime_error;
        // Implementation
    }

    /// @throw CommunicationError Communication error
    /// @throw BusError Bus error
    /// @throw Timeout On send timeout exception
    void send1(
        std::uint8_t* buffer,
        std::uint8_t bufferLength) noexcept(false) // Compliant - All and only
        // those checked exceptions
        // that can be thrown are
        // specified
    {
        // ...
        throw CommunicationError();
        // ...
        throw BusError();
        // ...
        throw Timeout("Timeout reached");
        // ...
    }

    /// @throw CommunicationError Communication error
    void send2(
        std::uint8_t* buffer,
        std::uint8_t bufferLength) noexcept(false) // Non-compliant - checked
        // exceptions that can be
        // thrown are missing from
        // specification
    {
        // ...
        throw CommunicationError();
        // ...
        throw Timeout("Timeout reached");
        // ...
    }

    class MemoryPartitioningError : std::exception
    {
        // Implementation
    }
```
void send3(
  std::uint8_t* buffer,
  std::uint8_t bufferLength) noexcept(false) // Non-compliant - additional
  // checked exceptions are
  // specified
{
  // ...
  throw CommunicationError();
  // ...
  throw Timeout("Timeout reached");
  // ...
}

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 $
void send1(std::uint8_t* buffer, std::uint8_t bufferLength) noexcept(false) // Non-compliant - unchecked exception documented
{
    // ...
    throw InvalidInitParameters("Invalid parameters");
}

void send2(std::uint8_t* buffer, std::uint8_t bufferLength) noexcept(false) // Compliant - unchecked exception not documented
{
    // ...
    throw InvalidInitParameters("Invalid parameters");
}

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

```cpp
//% $Id: A15-5-1.cpp 270728 2017-03-16 10:38:20Z piotr.tanski $
#include <stdexcept>
class C1
{
  public:
    C1() = default;
    C1(C1&& rhs)
      noexcept // Compliant - move constructor declared to be noexcept
      {
        // Implementation
      }
    C1& operator=(C1&& rhs) noexcept // Compliant - move assignment operator
      // declared to be noexcept
    {
      // Implementation
    }
    ~C1() noexcept // Compliant - no exceptions thrown from destructor
    {
      // Implementation
    }
};
void swap(C1& lhs, C1& rhs) noexcept // Compliant - swap function does not exit
  // with an exception
{
  // Implementation
}
class C2
{
  public:
    C2() = default;
    C2(C2&& rhs)
      noexcept // Compliant - move constructor declared to be noexcept
      {
        try
        {
          // ...
          throw std::runtime_error("Error"); // Exception will not leave move constructor
        }
        catch (std::exception& e)
        {
          // Handle runtime error
        }
      }
    C2& operator=(C2&& rhs) noexcept // Compliant - move assignment operator
      // declared to be noexcept
    {
      try
      {
        // ...
        throw std::runtime_error("Error"); // Exception will not leave move assignment operator
      }
    }
};
```
"Error"); // Exception will not leave assignment operator
}
catch (std::exception& e)
{
    // Handle runtime error
}
return *this;
~C2() // Non-compliant - the destructor does not contain the noexcept
    // specification
{
    try
    {
        // ...
        throw std::runtime_error("Error"); // Exception will not leave the destructor
    }
    catch (std::exception& e)
    {
        // Handle runtime error
    }
};
void swap(C2& lhs, C2& rhs) noexcept(false) // Non-compliant - swap function declared to be noexcept(false)
{
    // Implementation
    throw std::runtime_error("Swap function failed"); // Non-compliant - swap
         // function exits with an
    // exception
}
class C3
{
public:
    C3() = default;
    C3(C3&& rhs) // Non-compliant - move constructor throws
    {
        // ...
        throw std::runtime_error("Error");
    }
C3& operator=(C3&& rhs) // Non-compliant - move assignment operator throws
    {
        // ...
        throw std::runtime_error("Error");
        return *this;
    }
~C3() // Non-compliant - destructor exits with an exception
{
throw std::runtime_error("Error");
}
static void operator delete(void* ptr, std::size_t sz)
{
    // ...
    throw std::runtime_error("Error"); // Non-compliant - operator delete
    // exits with an exception
}
void fn()
{
    C3 c1; // program terminates when c1 is destroyed
    C3* c2 = new C3;
    // ...
    delete c2; // program terminates when c2 is deleted
}

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);
```cpp
void f2() // Non-compliant
{
    f1(); // A call to throwing f1() may result in an implicit call to
    // std::terminate()
}

void f3() // Compliant
{
    try
    {
        f1(); // Handles all exceptions from f1() and does not re-throw
    }
    catch (...)
    {
        // Handle an exception
    }
}

void f4(const char* log)
{
// Report a log error
// ...
    std::exit(0); // Call std::exit() function which safely cleans up resources
}

void f5() // Compliant by exception
{
    try
    {
        f1();
    }
    catch (...)
    {
        f4("f1() function failed");
    }
}

int main(int, char**)
{
    if (std::atexit(&f2) != 0)
    {
        // Handle an error
    }

    if (std::atexit(&f3) != 0)
    {
        // Handle an error
    }

    // ...
    return 0;
}
```

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 \texttt{std::terminate}

• SEI CERT C++ [9]: ERR50-CPP. Do not abruptly terminate the program

**Rule A15-5-3 (required, implementation, automated)**
The \texttt{std::terminate()} function shall not be called implicitly.

**Rationale**
It is implementation-defined whether the call stack is unwound before \texttt{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 \texttt{std::terminate()} function implicitly, according to (\texttt{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 \texttt{std::atexit} or \texttt{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 \texttt{std::unexpected} is executed
7. a non-default handler for \texttt{std::unexpected} throws an exception that violates the previously violated dynamic exception specification, if the specification does not include \texttt{std::bad_exception}
8. \texttt{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 \texttt{std::thread}
10. a joinable \texttt{std::thread} is destroyed or assigned to

**Example**
```cpp
//% $Id: A15-5-3.cpp 271752 2017-03-23 12:07:07Z piotr.tanski $
#include <stdexcept>
#include <thread>
extern bool f1();
```
```cpp
class A
{
    public:
    A() noexcept(false)
    {
        // ...
        throw std::runtime_error("Error1");
    }
    ~A()
    {
        // ...
        throw std::runtime_error("Error2"); // Non-compliant - std::terminate()
        // called on throwing an exception
        // from noexcept(true) destructor
    }
};

class B
{
    public:
    ~B() noexcept(false)
    {
        // ...
        throw std::runtime_error("Error3");
    }
};

void f2()
{
    throw;
}

void threadFunc()
{
    A a; // Throws an exception from a’s constructor and does not handle it in
    // thread_func()
}

void f3()
{
    try
    {
        std::thread t(threadFunc); // Non-compliant - std::terminate() called
        // on throwing an exception from
        // thread_func()
        if (f1())
        {
            throw std::logic_error("Error4");
        }
        else
        {
            f2(); // Non-compliant - std::terminate() called if there is no
            // active exception to be re-thrown by f2
        }
    }
    catch (std::exception &exc)
    {
        std::cerr << "Exception caught: " << exc.what() << std::endl;
    }
}
```


\begin{verbatim}
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   f3(); // Non-compliant - std::terminate() called if std::logic_error is
71   // thrown
72   return 0;
73 }
\end{verbatim}

See also

- MISRA C++ 2008 [6]: 15-5-3 (Required) The terminate() function shall not be
called implicitly.

\section*{6.16 Preprocessing directives}

\subsection*{6.16.0 General}

\begin{shaded}
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.
\end{shaded}

\textbf{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.

\textbf{Example}

\begin{verbatim}
1 // $Id: A16-0-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #pragma once // Non-compliant - implementation-defined feature
3 #ifndef HEADER_FILE_NAME // Compliant - include guard
4 #define HEADER_FILE_NAME // Compliant - include guard
\end{verbatim}
#include <cstdint> // Compliant - unconditional file inclusion

#ifdef WIN32
#include <windows.h> // Compliant - conditional file inclusion
#endif

#ifdef WIN32
std::int32_t fn1(
    std::int16_t x,
    std::int16_t y) noexcept; // Non-compliant - not a file inclusion
#endif

#if defined VERSION && VERSION > 2011L // Compliant
#include <array> // Compliant - conditional file inclusion
#elif VERSION > 1998L // Compliant
#include <vector> // Compliant - conditional file inclusion
#endif // Compliant

#define MAX_ARRAY_SIZE 1024U // Non-compliant
#ifndef MAX_ARRAY_SIZE // Non-compliant
#error "MAX_ARRAY_SIZE has not been defined" // Non-compliant
#endif // Non-compliant
#undef MAX_ARRAY_SIZE // Non-compliant

#define MIN(a, b) (((a) < (b)) ? (a) : (b)) // Non-compliant
#define PLUS2(X) ((X) + 2) // Non-compliant - function should be used instead
#define PI 3.14159F // Non-compliant - constexpr should be used instead
#define std ::int32_t long // Non-compliant - 'using' should be used instead
#define STARTIF if( // Non-compliant - language redefinition
#define HEADER "filename.h" // Non-compliant - string literal

void fn2() noexcept
{
    #ifdef __linux__ // Non-compliant - ifdef not used for file inclusion
    // ...
#endif

    #elif WIN32 // Non-compliant - elif not used for file inclusion
    // ...
#endif

    #elif __APPLE__ // Non-compliant - elif not used for file inclusion
    // ...
#endif

    #else // Non-compliant - else not used for file inclusion
    // ...
#endif // Non-compliant - endif not used for file inclusion or include guards
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 <headerfile.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

```cpp
#include <algorithm> // Non-compliant - nothing from algorithm header file is used
#include <array> // Non-compliant - nothing from array header file is used
#include <cstdint> // Compliant - std::int32_t, std::uint8_t are used
#include <iostream> // Compliant - cout is used
#include <stdexcept> // Compliant - out_of_range is used
#include <vector> // Compliant - vector is used

void fn1() noexcept
{
    std::int32_t x = 0;
    // ...
    std::uint8_t y = 0;
    // ...
}

void fn2() noexcept(false)
{
    try
    {
        std::vector<std::int32_t> v;
        // ...
        std::uint8_t idx = 3;
        std::int32_t value = v.at(idx);
    }
    catch (std::out_of_range& e)
    {
        std::cout << e.what() << '
';
    }
}
```

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**

```cpp
#ifndef HEADER_HPP
#define HEADER_HPP

#include <array>
#include <cstdint>

class B; // Compliant - type B can be included using forward declaration
    // std::into
    // the header file

class OutOfRangeException
    : public std::out_of_range // Non-compliant - <stdexcept> which defines
    // out_of_range included
    // implicitly through <array>
{
public:
    using std::out_of_range::out_of_range;
};

class A
{
public:
    // Interface of class A

private:
    std::array<std::uint32_t, 10> m_array; // Compliant - <array> included explicitly
    B* m_b;
    std::int32_t m_x; // Compliant - <cstdint> included explicitly
};
#endif
```

**See also**

- none

### 6.16.3 Macro replacement

**Rule M16-3-1** (required, implementation, automated)
There shall be at most one occurrence of the # or ## operators in a single macro definition.

See MISRA C++ 2008 [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**

```cpp
// $Id: A16-6-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
#include <cstdint>
#include <type_traits>
constexpr std::int32_t value = 0;
#if value > 10
#error "Incorrect value" // Non-compliant
#endif
void f1() noexcept
{
    static_assert(value <= 10, "Incorrect value"); // Compliant
    // ...
}
template <typename T>
void f2(T& a)
{
    static_assert(std::is_copy_constructible<T>::value,
                  "f2() function requires copying"); // Compliant
    // ...
}
```

**See also**

- none
6.16.7  Pragma directive

**Rule A16-7-1 (required, implementation, automated)**
The `#pragma` directive shall not be used.

**Rationale**

The `#pragma` directive is implementation-defined and causes the implementation to behave in implementation-defined manner.

**Example**

```cpp
// $Id: A16-7-1.hpp 270497 2017-03-14 14:58:50Z piotr.tanski $
// #pragma once // Non-compliant - implementation-defined manner
#ifdef A16_7_1_HPP // Compliant - equivalent to #pragma once directive
#define A16_7_1_HPP

// ...
#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**
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

```cpp
// $Id: A17-1-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
#include <cerrno>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <stdexcept>

void fn1(const char* filename) // Compliant - C code is isolated; fn1()
  // function is a wrapper.
{
  FILE* handle = fopen(filename, "rb");
  if (handle == NULL)
    {
      throw std::system_error(errno, std::system_category());
    }
  // ...
  fclose(handle);
}

void fn2() noexcept
{
  try
  {
    fn1("filename.txt"); // Compliant - fn1() allows you to use C code like
    // C++ code
    // ...
  }
```
catch (std::system_error& e)
{
    std::cerr << "Error: " << e.code() << " - " << e.what() << '
';
}

std::int32_t fn3(const char* filename) noexcept // Non-compliant - placing C
// functions calls along with C++
// code forces a developer to be
// responsible for C-specific error
// handling, explicit resource
// cleanup, etc.
{
    FILE* handle = fopen(filename, "rb");
    if (handle == NULL)
    {
        std::cerr << "An error occured: " << errno << " - " << strerror(errno)
               << '
';
        return errno;
    }

    try
    {
        // ...
        fclose(handle);
    }

    catch (std::system_error& e)
    {
        fclose(handle);
    }

    catch (std::exception& e)
    {
        fclose(handle);
    }

    return errno;

See also

- MISRA C++ 2008 [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.

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

```cpp
#include <cstdint>
#include <cstdlib>
#include <string>

std::int32_t f1(const char* str) noexcept
{
    return atoi(str); // Non-compliant - undefined behavior if str can not
                     // be converted
}

std::int32_t f2(std::string const& str) noexcept(false)
{
    return std::stoi(str); // Compliant - throws a std::invalid_argument
                           // exception if str can not be converted
}
```

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 <clocale> (locale.h) and the setlocale function shall not be used.

**Rationale**

A call to the setlocale function introduces a data race with other calls to setlocale function.

It may also introduce a data race with calls to functions that are affected by the current locale settings: fprintf, isprint, iswdigit, localeconv, tolower, fscanf, ispunct, iswgraph, mblen, toupper, isalnum, isspace, iswlower, mbstowcs, towlower, isalpha, isupper, iswprint, mbtowc, towupper, isblank, iswalnum, iswuprincip, setlocale, wcscoll, iscntrl, iswalpha, iswspace, strcoll, wcstod, isdigit, iswblank, iswupper, strerror, wcstombs, isgraph, iswcntrl, iswxdigit, strtol, 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**

```cpp
// $Id: A18-1-1.cpp 271752 2017-03-23 12:07:07Z piotr.tanski $
#include <algorithm>
#include <array>
#include <cstdint>

void fn() noexcept
{
  const std::uint8_t size = 10;
  std::int32_t a1[size]; // Non-compliant
  std::array<std::int32_t, size> a2; // Compliant
  // ...
  std::sort(a1, a1 + size);
  std::sort(a2.begin(), a2.end()); // More readable and maintainable way of
  // working with STL algorithms
}

class A
{
public:
  static constexpr std::uint8_t array[] = {0, 1, 2}; // Compliant by exception
};
```

**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**
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**

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

void fn() noexcept
{
    std::auto_ptr<std::int32_t> ptr1(new std::int32_t(10)); // Non-compliant
    std::unique_ptr<std::int32_t> ptr2 =
        std::make_unique<std::int32_t>(10); // Compliant
    std::vector<std::auto_ptr<std::int32_t>> v; // Non-compliant
}
```

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

```cpp
// $Id: A18-1-4.cpp 271752 2017-03-23 12:07:07Z piotr.tanski $
#include <array>
#include <memory>
#include <vector>
class A {
};

void f1() noexcept(false) {
    A cArray[10]; // Compliant
    std::shared_ptr<A> ptr1(cArray); // Non-compliant - ptr1 will not delete all of its elements
    std::shared_ptr<A> ptr2(new A[10]); // Non-compliant - ptr2 will not delete all of its elements
    // std::shared_ptr<A[]> ptr3(new A[10]); // Non-compliant - compilation error
    std::shared_ptr<std::array<A, 10>> ptr4 = std::make_shared<std::array<A, 10>>(); // Compliant
    std::shared_ptr<std::vector<A>> ptr5 = std::make_shared<std::vector<A>>(10, A()); // Compliant
}

void f2() noexcept(false) {
    // std::unique_ptr<A> ptr1 = std::make_unique<A>(10); // Non-compliant - no such constructor in class A
    std::unique_ptr<A> ptr2{
        new A[10]}; // Non-compliant - ptr2 will not delete all of its elements
    std::unique_ptr<A[]> ptr3 = std::make_unique<A[]>(10); // Compliant - std::unique_ptr provides a specialization for T[] types
}
```

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

```cpp
// $Id: A18-1-5.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
#include <cstdint>
#include <memory>

void fn1(std::unique_ptr<std::int32_t> ptr) // Compliant
{
    // fn1() now owns ptr
}

void fn2(std::unique_ptr<std::int32_t>& ptr) // Compliant
{
    // fn2() is explicitly allowed to make a ptr to refer to a different object
}

void fn3(const std::unique_ptr<std::int32_t>& ptr) // Non-compliant
{
    // fn3 takes ptr by const reference but still it is able to make a ptr to
    // refer to a different object
    *ptr = 10; // No compilation error
}

void fn4(const std::int32_t* ptr) // Compliant
{
    // fn4 takes a const raw pointer
    //*ptr = 10; // Compilation error
}

void fn5()
{
    fn1(std::make_unique<std::int32_t>(0));
    std::unique_ptr<std::int32_t> ptr = std::make_unique<std::int32_t>(0);
    fn2(ptr);
    fn3(ptr);
    fn4(ptr.get());
}
```

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

<table>
<thead>
<tr>
<th>Issue</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory leaks</td>
<td>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).</td>
</tr>
<tr>
<td>Memory fragmentation</td>
<td>Memory allocator used in the project needs to guarantee that no memory fragmentation occurs (A18-5-5).</td>
</tr>
<tr>
<td>Invalid memory access</td>
<td>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).</td>
</tr>
</tbody>
</table>
Guidelines for the use of the C++14 language in critical and safety-related systems

AUTOSAR AP Release 17-03

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

```cpp
// $Id: A18-5-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
#include <cstdint>
#include <cstdlib>

void f1() noexcept(false)
{
    // Non-compliant
    std::int32_t* p1 = static_cast<std::int32_t*>(malloc(sizeof(std::int32_t)));
    *p1 = 0;

    // Compliant
    std::int32_t* p2 = new std::int32_t(0);
}
```
// Compliant
delete p2;

// Non-compliant
free(p1);

// Non-compliant
std::int32_t* array1 =
    static_cast<std::int32_t*>(calloc(10, sizeof(std::int32_t)));

// Non-compliant
std::int32_t* array2 =
    static_cast<std::int32_t*>(realloc(array1, 10 * sizeof(std::int32_t)));

// Compliant
std::int32_t* array3 = new std::int32_t[10];

// Compliant
delete[] array3;

// Non-compliant
free(array2);

// Non-compliant
free(array1);

} 

void f2() noexcept(false) 
{

    // Non-compliant
    std::int32_t* p1 = static_cast<std::int32_t*>(malloc(sizeof(std::int32_t)));
    // Non-compliant - undefined behavior
delete p1;

    std::int32_t* p2 = new std::int32_t(0); // Compliant
    free(p2); // Non-compliant - undefined behavior

} 

void operator delete(void* ptr) noexcept
{
    std::free(ptr); // Compliant by exception
}

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

```cpp
// $Id: A1-7-2.cpp 270728 2017-03-16 10:38:20Z piotr.tanski $
#include <cstdint>
#include <memory>
#include <vector>
std::int32_t fn1()
{
    std::int32_t errorCode{0};
    std::int32_t* ptr =
        new std::int32_t{0}; // Non-compliant - new called explicitly
    // ...
    if (errorCode != 0)
    {
        throw std::runtime_error("Error"); // Memory leak could occur here
    }
    // ...
    if (errorCode != 0)
    {
        return 1; // Memory leak could occur here
    }
    // ...
    return errorCode; // Memory leak could occur here
}
std::int32_t fn2()
{
    std::int32_t errorCode{0};
```
...
return 1; // No memory leak
}

// ...
return errorCode; // No memory leak

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

// $Id: A18-5-3.cpp 271687 2017-03-23 08:57:35Z piotr.tanski$
#include <cstdint>
void fn1()
{
    std::int32_t* array =
        new std::int32_t[10]; // new[] operator used to allocate the
        // memory for an array
    // ...
    delete array; // Non-compliant - delete[] operator supposed to be used
}

void fn2()
{
    std::int32_t* object = new std::int32_t{0}; // new operator used to allocate the memory for an
        // integer type
    // ...
    delete[] object; // Non-compliant - delete operator supposed to be used
}

void fn3()
{
    std::int32_t* object = new std::int32_t{0};
    std::int32_t* array = new std::int32_t[10];
    // ...
    delete[] array; // Compliant
delete object; // Compliant
}

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**

```cpp
#include <cstdlib>

void operator delete(
    void* ptr) noexcept // Compliant - sized version is defined
{
    std::free(ptr);
}

void operator delete(
    void* ptr,
    std::size_t size) noexcept // Compliant - unsized version is defined
{
    std::free(ptr);
}
```

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

```c
#include <dlfcn.h>
#include <cstddef>

void* mallocBad(size_t size) // Non-compliant, malloc from libc does not guarantee deterministic execution time
{
    void* (*libc_malloc)(size_t) = dlsym(RTLD_NEXT, "malloc");
    return libc_malloc(size);
}

void freeBad(void* ptr) // Non-compliant, malloc from libc does not guarantee deterministic execution time
{
    void (*libc_free)(void*) = dlsym(RTLD_NEXT, "free");
    libc_free(ptr);
}

void* mallocGood(size_t size) // Compliant - custom malloc implementation that will guarantee deterministic execution time
{
```
// Custom implementation that provides deterministic worst-case execution
// time

void freeGood(void* ptr) // Compliant - custom malloc implementation that will
// guarantee deterministic execution time
{
    // Custom implementation that provides deterministic worst-case execution
    // time
}
Guidelines for the use of the C++14 language in critical and safety-related systems
AUTOSAR AP Release 17-03

1 // Id: A18-5-3.cpp 270728 2017-03-16 10:38:20Z piotr.tanski $
2 #include <cstdint>
3 #include <memory>
4 #include <vector>
5 std::int8_t appMainLoop() noexcept
6 {
7     std::int8_t retCode = 0;
8     std::int32_t* arr[10];
9     while (true)
10     {
11         for (std::int8_t i = 0; i < 10; ++i)
12         {
13             arr[i] = new std::int32_t{i}; // Non-compliant - allocation in a phase that
14                 // requires real-time
15         }
16         // Implementation
17         for (auto& i : arr)
18         {
19             delete i; // Non-compliant - deallocation in a phase that requires
20                 // real-time
21         }
22     }
23     return retCode;
24 }
25 static std::int32_t* object =
26     new std::int32_t{0}; // Compliant - allocating in start-up phase
27 int main(int, char**)
28 {
29     std::unique_ptr<std::int32_t> ptr =
30         std::make_unique<std::int32_t>({0}); // Compliant
31     std::vector<std::int32_t> vec; // Compliant
32     vec.reserve(10); // Compliant
33     std::int8_t code = appMainLoop();
34     return code;
35 }

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

```cpp
// $Id: A18-9-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
#include <cstdint>
#include <functional>
class A {
    // Implementation
};
void fn(A const& a, double y) noexcept {
    // Implementation
}
void f1() noexcept {
    double y = 0.0;
    auto function = std::bind(&fn, std::placeholders::_1, y); // Non-compliant
    // ...
    A const a{};
    function(a);
}
void f2() noexcept {
    auto lambda = [](A const& a) -> void {
        double y = 0.0;
        fn(a, y);
    }; // Compliant
    // ...
    A const a{};
    lambda(a);
}
```

See also
- Effective Modern C++ [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

```cpp
1 // $Id: A18-9-2.cpp 271927 2017-03-24 12:01:35Z piotr.tanski $
2 #include <cstdint>
3 #include <string>
4 #include <utility>
5 class A
6 {
7    public:
8        explicit A(std::string&& s)
9            : str(std::move(s)) // Compliant - forwarding rvalue reference
10        {
11        }
12    }
13
14    private:
15        std::string str;
16    };
17 class B
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
```
void fn4() noexcept
{
    B b1;
    B& b2 = b1;
    fn2(b2); // fn1(const B&) is called
    fn2(std::move(b1)); // fn1(B&&) is called
    fn3(b2); // fn1(B&&) is called
    fn3(std::move(b1)); // fn1(B&&) is called
}

See also

- HIC++ v4.0 [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

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

See also

- HIC++ v4.0 [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

```cpp
#include <cstdint>
#include <iostream>
#include <utility>

template <typename T1, typename T2>
void f1(T1 const& t1, T2& t2){
    // ...
}

template <typename T1, typename T2>
void f2(T1&& t1, T2&& t2)
{
    f1(std::forward<T1>(t1), std::forward<T2>(t2));
    ++t2;  // Non-compliant
}
```

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

```cpp
//% $Id: A23-0-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
#include <cstdint>
#include <vector>

void fn1(std::vector<std::int32_t>& v) noexcept
{
    for (std::vector<std::int32_t>::const_iterator iter{v.cbegin()}, end{v.cend()};
         iter != end;
         ++iter) // Compliant
    {
        // ...
    }
}

void fn2(std::vector<std::int32_t>& v) noexcept
{
    for (auto iter{v.cbegin()}, end{v.cend()}; iter != end;
         ++iter) // Compliant
    {
        // ...
    }
}

void fn3(std::vector<std::int32_t>& v) noexcept
{
    for (std::vector<std::int32_t>::const_iterator iter{v.begin()},
         end{v.end()};
         iter != end;
         ++iter) // Non-compliant
    {
        // ...
    }
}
```

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

```cpp
1 // $Id: A27-0-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <string.h>
3 #include <cstdint>
4 #include <cstdio>
5 void f1(const char* name) // name restricted to 256 or fewer characters
    { 
        static const char format[] = "Name: %s .");
        size_t len = strlen(name) + sizeof(format);
        char* msg = new char[len];

        if (msg == nullptr)
        {
            // Handle an error
        }

        std::int32_t ret =
            snprintf(msg,
                len,
                format,
                name); // Non-compliant - no additional check for overflows

        if (ret < 0)
        {
            // Handle an error
        }
        else if (ret >= len)
        {
            // Handle truncated output
        }
    }
```
void f2(const char* name)
{
    static const char format[] = "Name: %s .";
    fprintf(stderr, format, name); // Compliant - untrusted input passed as one
    // of the variadic arguments, not as part of
    // vulnerable format string
}

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**

```cpp
#include <iostream>
#include <string>

void f1() noexcept
{
    char buffer[10];
    std::cin >> buffer; // Non-compliant - this could lead to a buffer overflow
}

void f2() noexcept
{
    std::string string1;
    std::string string2;
    std::cin >> string1 >> string2; // Compliant - no buffer overflows
}

void f3(std::istream& in) noexcept
{
    char buffer[32];
    try
    {
        in.read(buffer, sizeof(buffer));
    }
```
catch (std::ios_base::failure&)
{
    // Handle an error
}

std::string str(buffer); // Non-compliant - if 'buffer' is not null
// terminated, then constructing std::string leads
// to undefined behavior.
}

void f4(std::istream& in) noexcept
{
    char buffer[32];

    try
    {
        in.read(buffer, sizeof(buffer));
    }
    catch (std::ios_base::failure&)
    {
        // Handle an error
    }

    std::string str(buffer, in.gcount()); // Compliant
}

See also

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

Bibliography


[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.

<table>
<thead>
<tr>
<th>MISRA Rule:</th>
<th>Relation type:</th>
<th>Related rule:</th>
<th>Comment:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1-1 (Required) A project shall not contain unreachable code.</td>
<td>1 - Identical</td>
<td>M0-1-1</td>
<td>-</td>
</tr>
<tr>
<td>0-1-2 (Required) A project shall not contain infeasible paths.</td>
<td>2 - Small differences</td>
<td>M0-1-2</td>
<td>Note about constexpr functions added.</td>
</tr>
<tr>
<td>0-1-3 (Required) A project shall not contain unused variables.</td>
<td>1 - Identical</td>
<td>M0-1-3</td>
<td>-</td>
</tr>
<tr>
<td>0-1-4 (Required) A project shall not contain non-volatile POD variables having only one use.</td>
<td>1 - Identical</td>
<td>M0-1-4</td>
<td>-</td>
</tr>
<tr>
<td>0-1-5 (Required) A project shall not contain unused type declarations.</td>
<td>1 - Identical</td>
<td>M0-1-5</td>
<td>-</td>
</tr>
<tr>
<td>Requirement</td>
<td>Level</td>
<td>Rule</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>0-1-6</td>
<td>2</td>
<td>A0-1-1</td>
<td>A project shall not contain instances of non-volatile variables being given values that are never subsequently used.</td>
</tr>
<tr>
<td>0-1-7</td>
<td>2</td>
<td>A0-1-2</td>
<td>The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.</td>
</tr>
<tr>
<td>0-1-8</td>
<td>1</td>
<td>M0-1-8</td>
<td>All functions with void return type shall have external side effect(s).</td>
</tr>
<tr>
<td>0-1-9</td>
<td>1</td>
<td>M0-1-9</td>
<td>There shall be no dead code.</td>
</tr>
<tr>
<td>0-1-10</td>
<td>3</td>
<td>M0-1-10, A0-1-2</td>
<td>Every defined function shall be called at least once.</td>
</tr>
<tr>
<td>0-1-11</td>
<td>1</td>
<td>M0-1-11</td>
<td>There shall be no unused parameters (named or unnamed) in non-virtual functions.</td>
</tr>
<tr>
<td>0-1-12</td>
<td>1</td>
<td>M0-1-12</td>
<td>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.</td>
</tr>
<tr>
<td>0-2-1</td>
<td>1</td>
<td>M0-2-1</td>
<td>An object shall not be assigned to an overlapping object.</td>
</tr>
<tr>
<td>0-3-1</td>
<td>1</td>
<td>M0-3-1</td>
<td>Minimization of runtime 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.</td>
</tr>
<tr>
<td>0-3-2</td>
<td>1</td>
<td>M0-3-2</td>
<td>If a function generates error information, then that error information shall be tested.</td>
</tr>
<tr>
<td>0-4-1</td>
<td>1</td>
<td>M0-4-1</td>
<td>Use of scaled-integer or fixed-point arithmetic shall be documented.</td>
</tr>
<tr>
<td>0-4-2</td>
<td>1</td>
<td>M0-4-2</td>
<td>Use of floating-point arithmetic shall be documented.</td>
</tr>
<tr>
<td>Rule Code</td>
<td>Overview</td>
<td>Level</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>0-4-3</td>
<td>Floating-point implementations shall comply with a defined floating-point standard.</td>
<td>3 - Significant differences</td>
<td>Specified that floating-point implementations need to comply with IEEE 754 standard.</td>
</tr>
<tr>
<td>1-0-2</td>
<td>Multiple compilers shall only be used if they have a common, defined interface.</td>
<td>1 - Identical</td>
<td>-</td>
</tr>
<tr>
<td>1-0-3</td>
<td>The implementation of integer division in the chosen compiler shall be determined and documented.</td>
<td>3 - Significant differences</td>
<td>Specified that the implementation of integer division shall comply with the C++ Language Standard.</td>
</tr>
<tr>
<td>2-2-1</td>
<td>The character set and the corresponding encoding shall be documented.</td>
<td>4 - Rejected</td>
<td>Rule rejected. The character set explicitly specified in A2-2-1.</td>
</tr>
<tr>
<td>2-3-1</td>
<td>Trigraphs shall not be used.</td>
<td>2 - Small differences</td>
<td>All trigraphs listed in rationale. Example extended.</td>
</tr>
<tr>
<td>2-5-1</td>
<td>Digraphs should not be used.</td>
<td>3 - Significant differences</td>
<td>Obligation level changed to “Required”.</td>
</tr>
<tr>
<td>2-7-1</td>
<td>The character sequence /* shall not be used within a C-style comment.</td>
<td>3 - Significant differences</td>
<td>Using the C-style comments is not allowed.</td>
</tr>
<tr>
<td>2-7-2</td>
<td>Sections of code shall not be commented out using C-style comments.</td>
<td>2 - Small differences</td>
<td>Commenting-out code sections is not allowed.</td>
</tr>
<tr>
<td>2-7-3</td>
<td>Sections of code should not be “commented out” using C++ comments.</td>
<td>2 - Small differences</td>
<td>Obligation level changed to “Required”. Commenting-out code sections is not allowed.</td>
</tr>
<tr>
<td>2-10-1</td>
<td>Different identifiers shall be typographically unambiguous.</td>
<td>1 - Identical</td>
<td>-</td>
</tr>
<tr>
<td>Rule</td>
<td>Description</td>
<td>Type</td>
<td>Change</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td>------</td>
<td>--------</td>
</tr>
<tr>
<td>2-10-2</td>
<td>Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.</td>
<td>2 - Small differences</td>
<td>A2-11-1</td>
</tr>
<tr>
<td>2-10-3</td>
<td>A typedef name (including qualification, if any) shall be a unique identifier.</td>
<td>1 - Identical</td>
<td>M2-10-3</td>
</tr>
<tr>
<td>2-10-4</td>
<td>A class, union or enum name (including qualification, if any) shall be a unique identifier.</td>
<td>2 - Small differences</td>
<td>A2-11-3</td>
</tr>
<tr>
<td>2-10-5</td>
<td>The identifier name of a non-member object or function with static storage duration should not be reused.</td>
<td>3 - Significant differences</td>
<td>A2-11-4</td>
</tr>
<tr>
<td>2-10-6</td>
<td>If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.</td>
<td>1 - Identical</td>
<td>M2-10-6</td>
</tr>
<tr>
<td>2-13-1</td>
<td>Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.</td>
<td>2 - Small differences</td>
<td>A2-14-1</td>
</tr>
<tr>
<td>2-13-2</td>
<td>Octal constants (other than zero) and octal escape sequences (other than “\0”) shall not be used.</td>
<td>1 - Identical</td>
<td>M2-13-2</td>
</tr>
<tr>
<td>2-13-3</td>
<td>A “U” suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.</td>
<td>1 - Identical</td>
<td>M2-13-3</td>
</tr>
<tr>
<td>2-13-4</td>
<td>Literal suffixes shall be upper case.</td>
<td>1 - Identical</td>
<td>M2-13-4</td>
</tr>
<tr>
<td>2-13-5</td>
<td>Narrow and wide string literals shall not be concatenated.</td>
<td>2 - Small differences</td>
<td>A2-14-2</td>
</tr>
<tr>
<td>3-1-1</td>
<td>It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.</td>
<td>3 - Significant differences</td>
<td>A3-1-1</td>
</tr>
<tr>
<td>3-1-2</td>
<td>Functions shall not be declared at block scope.</td>
<td>1 - Identical</td>
<td>M3-1-2</td>
</tr>
<tr>
<td>3-1-3</td>
<td>When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.</td>
<td>2 - Small differences</td>
<td>A3-1-4</td>
</tr>
<tr>
<td>3-2-1</td>
<td>All declarations of an object or function shall have compatible types.</td>
<td>1 - Identical</td>
<td>M3-2-1</td>
</tr>
<tr>
<td>Rule</td>
<td>(Required)</td>
<td>Description</td>
<td>Type of Difference</td>
</tr>
<tr>
<td>------</td>
<td>------------</td>
<td>-------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>3-2-2</td>
<td>The One Definition Rule shall not be violated.</td>
<td>1 - Identical</td>
<td>M3-2-2</td>
</tr>
<tr>
<td>3-2-3</td>
<td>A type, object or function that is used in multiple translation units shall be declared in one and only one file.</td>
<td>1 - Identical</td>
<td>M3-2-3</td>
</tr>
<tr>
<td>3-2-4</td>
<td>An identifier with external linkage shall have exactly one definition.</td>
<td>1 - Identical</td>
<td>M3-2-4</td>
</tr>
<tr>
<td>3-3-1</td>
<td>Objects or functions with external linkage shall be declared in a header file.</td>
<td>2 - Small differences</td>
<td>A3-3-1</td>
</tr>
<tr>
<td>3-3-2</td>
<td>If a function has internal linkage then all re-declarations shall include the static storage class specifier.</td>
<td>1 - Identical</td>
<td>M3-3-2</td>
</tr>
<tr>
<td>3-4-1</td>
<td>An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.</td>
<td>1 - Identical</td>
<td>M3-4-1</td>
</tr>
<tr>
<td>3-9-1</td>
<td>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.</td>
<td>1 - Identical</td>
<td>M3-9-1</td>
</tr>
<tr>
<td>3-9-2</td>
<td>(Advisory) typedefs that indicate size and signedness should be used in place of the basic numerical types.</td>
<td>3 - Significant differences</td>
<td>M3-9-1</td>
</tr>
<tr>
<td>3-9-3</td>
<td>The underlying bit representations of floating-point values shall not be used.</td>
<td>1 - Identical</td>
<td>M3-9-3</td>
</tr>
<tr>
<td>4-5-1</td>
<td>Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &amp;&amp;,</td>
<td></td>
<td>, !, the equality operators == and !==, the unary &amp; operator, and the conditional operator.</td>
</tr>
<tr>
<td>4-5-2</td>
<td>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 &amp; operator, and the relational operators &lt;, &lt;=, &gt;, &gt;=.</td>
<td>3 - Significant differences</td>
<td>A4-5-1</td>
</tr>
<tr>
<td>Requirement</td>
<td>Type</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>4-5-3</td>
<td>Required</td>
<td>Expressions with type (plain) <code>char</code> and <code>wchar_t</code> shall not be used as operands to built-in operators other than the assignment operator <code>=</code>, the equality operators <code>==</code> and <code>!=</code>, and the unary <code>&amp;</code> operator.</td>
<td></td>
</tr>
<tr>
<td>4-10-1</td>
<td>Required</td>
<td>NULL shall not be used as an integer value.</td>
<td></td>
</tr>
<tr>
<td>4-10-2</td>
<td>Required</td>
<td>Literal zero (0) shall not be used as the null-pointer-constant.</td>
<td></td>
</tr>
<tr>
<td>5-0-1</td>
<td>Required</td>
<td>The value of an expression shall be the same under any order of evaluation that the standard permits.</td>
<td></td>
</tr>
<tr>
<td>5-0-2</td>
<td>Advisory</td>
<td>Limited dependence should be placed on C++ operator precedence rules in expressions.</td>
<td></td>
</tr>
<tr>
<td>5-0-3</td>
<td>Required</td>
<td>A cvalue expression shall not be implicitly converted to a different underlying type.</td>
<td></td>
</tr>
<tr>
<td>5-0-4</td>
<td>Required</td>
<td>An implicit integral conversion shall not change the signedness of the underlying type.</td>
<td></td>
</tr>
<tr>
<td>5-0-5</td>
<td>Required</td>
<td>There shall be no implicit floating-integral conversions.</td>
<td></td>
</tr>
<tr>
<td>5-0-6</td>
<td>Required</td>
<td>An implicit integral or floating-point conversion shall not reduce the size of the underlying type.</td>
<td></td>
</tr>
<tr>
<td>5-0-7</td>
<td>Required</td>
<td>There shall be no explicit floating-integral conversions of a cvalue expression.</td>
<td></td>
</tr>
<tr>
<td>5-0-8</td>
<td>Required</td>
<td>An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.</td>
<td></td>
</tr>
<tr>
<td>5-0-9</td>
<td>Required</td>
<td>An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.</td>
<td></td>
</tr>
<tr>
<td>5-0-10</td>
<td>Required</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>5-0-11</td>
<td>Required</td>
<td>The plain char type shall only be used for the storage and use of character values.</td>
<td></td>
</tr>
<tr>
<td>5-0-12</td>
<td>Required</td>
<td>Signed char and unsigned char type shall only be used for the storage and use of numeric values.</td>
<td></td>
</tr>
<tr>
<td>Requirement</td>
<td>Type of Difference</td>
<td>Rule</td>
<td>Rationale/Comment</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------</td>
<td>------</td>
<td>------------------</td>
</tr>
<tr>
<td>5-0-13</td>
<td>2 - Small differences</td>
<td>A5-0-2</td>
<td>Example extended.</td>
</tr>
<tr>
<td>5-0-14</td>
<td>1 - Identical</td>
<td>M5-0-14</td>
<td></td>
</tr>
<tr>
<td>5-0-15</td>
<td>1 - Identical</td>
<td>M5-0-15</td>
<td></td>
</tr>
<tr>
<td>5-0-16</td>
<td>1 - Identical</td>
<td>M5-0-16</td>
<td></td>
</tr>
<tr>
<td>5-0-17</td>
<td>1 - Identical</td>
<td>M5-0-17</td>
<td></td>
</tr>
<tr>
<td>5-0-18</td>
<td>1 - Identical</td>
<td>M5-0-18</td>
<td></td>
</tr>
<tr>
<td>5-0-19</td>
<td>2 - Small differences</td>
<td>A5-0-3</td>
<td>Example changed - typedef replaced with using.</td>
</tr>
<tr>
<td>5-0-20</td>
<td>1 - Identical</td>
<td>M5-0-20</td>
<td></td>
</tr>
<tr>
<td>5-0-21</td>
<td>1 - Identical</td>
<td>M5-0-21</td>
<td></td>
</tr>
<tr>
<td>5-2-1</td>
<td>1 - Identical</td>
<td>M5-2-1</td>
<td></td>
</tr>
<tr>
<td>5-2-2</td>
<td>1 - Identical</td>
<td>M5-2-2</td>
<td></td>
</tr>
<tr>
<td>5-2-3</td>
<td>1 - Identical</td>
<td>M5-2-3</td>
<td></td>
</tr>
<tr>
<td>5-2-4</td>
<td>3 - Significant differences</td>
<td>A5-2-2</td>
<td>Rule title and rationale reformulated, detailed explanation and possible alternatives added. Example reworked.</td>
</tr>
</tbody>
</table>

Guidelines for the use of the C++14 language in critical and safety-related systems

AUTOSAR AP Release 17-03
<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
<th>Difference</th>
<th>Related Clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-2-5</td>
<td>(Required) A cast shall not remove any const or volatile qualification from the type of a pointer or reference.</td>
<td>2 - Small differences</td>
<td>A5-2-3 Added a note to rationale. Example reworked.</td>
</tr>
<tr>
<td>5-2-6</td>
<td>(Required) A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.</td>
<td>1 - Identical</td>
<td>M5-2-6</td>
</tr>
<tr>
<td>5-2-7</td>
<td>(Required) An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.</td>
<td>3 - Significant differences</td>
<td>A5-2-4 Rule title and rationale reformulated to prohibit reinterpret_cast usage. Example reworked.</td>
</tr>
<tr>
<td>5-2-8</td>
<td>(Required) An object with integer type or pointer to void type shall not be converted to an object with pointer type.</td>
<td>1 - Identical</td>
<td>M5-2-8</td>
</tr>
<tr>
<td>5-2-9</td>
<td>(Advisory) A cast shall not convert a pointer type to an integral type.</td>
<td>2 - Small differences</td>
<td>M5-2-9 Obligation level changed to “Required”.</td>
</tr>
<tr>
<td>5-2-10</td>
<td>(Advisory) The increment (++) and decrement (–) operators shall not be mixed with other operators in an expression.</td>
<td>2 - Small differences</td>
<td>M5-2-10 Obligation level changed to “Required”.</td>
</tr>
<tr>
<td>5-2-11</td>
<td>(Required) The comma operator, &amp;&amp; operator and the operator shall not be overloaded.</td>
<td>1 - Identical</td>
<td>M5-2-11</td>
</tr>
<tr>
<td>5-2-12</td>
<td>(Required) An identifier with array type passed as a function argument shall not decay to a pointer.</td>
<td>1 - Identical</td>
<td>M5-2-12</td>
</tr>
<tr>
<td>5-3-1</td>
<td>(Required) Each operand of the ! operator, the logical &amp;&amp; or the logical</td>
<td></td>
<td>operators shall have type bool.</td>
</tr>
<tr>
<td>5-3-2</td>
<td>(Required) The unary minus operator shall not be overloaded.</td>
<td>1 - Identical</td>
<td>M5-3-2</td>
</tr>
<tr>
<td>5-3-3</td>
<td>(Required) The unary &amp; operator shall not be overloaded.</td>
<td>1 - Identical</td>
<td>M5-3-3</td>
</tr>
<tr>
<td>5-3-4</td>
<td>(Required) Evaluation of the operand to the sizeof operator shall not contain side effects.</td>
<td>1 - Identical</td>
<td>M5-3-4</td>
</tr>
<tr>
<td>5-8-1</td>
<td>(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.</td>
<td>1 - Identical</td>
<td>M5-8-1</td>
</tr>
<tr>
<td>5-14-1</td>
<td>(Required) The right hand operand of a logical &amp;&amp; or</td>
<td></td>
<td>operator shall not contain side effects.</td>
</tr>
<tr>
<td>Section</td>
<td>Requirement</td>
<td>Detailed Description</td>
<td>Level</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>----------------------</td>
<td>-------</td>
</tr>
<tr>
<td>5-17-1</td>
<td>(Required)</td>
<td>The semantic equivalence between a binary operator and its assignment operator form shall be preserved.</td>
<td>1 - Identical</td>
</tr>
<tr>
<td>5-18-1</td>
<td>(Required)</td>
<td>The comma operator shall not be used.</td>
<td>1 - Identical</td>
</tr>
<tr>
<td>5-19-1</td>
<td>(Required)</td>
<td>Evaluation of constant unsigned integer expressions shall not lead to wrap-around.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>6-2-1</td>
<td>(Required)</td>
<td>Assignment operators shall not be used in subexpressions.</td>
<td>1 - Identical</td>
</tr>
<tr>
<td>6-2-2</td>
<td>(Required)</td>
<td>Floating-point expressions shall not be directly or indirectly tested for equality or inequality.</td>
<td>1 - Identical</td>
</tr>
<tr>
<td>6-2-3</td>
<td>(Required)</td>
<td>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.</td>
<td>1 - Identical</td>
</tr>
<tr>
<td>6-3-1</td>
<td>(Required)</td>
<td>The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.</td>
<td>1 - Identical</td>
</tr>
<tr>
<td>6-4-1</td>
<td>(Required)</td>
<td>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.</td>
<td>1 - Identical</td>
</tr>
<tr>
<td>6-4-2</td>
<td>(Required)</td>
<td>All if ... else if constructs shall be terminated with an else clause.</td>
<td>1 - Identical</td>
</tr>
<tr>
<td>6-4-3</td>
<td>(Required)</td>
<td>A switch statement shall be a well-formed switch statement.</td>
<td>1 - Identical</td>
</tr>
<tr>
<td>6-4-4</td>
<td>(Required)</td>
<td>A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.</td>
<td>1 - Identical</td>
</tr>
<tr>
<td>6-4-5</td>
<td>(Required)</td>
<td>An unconditional throw or break statement shall terminate every non-empty switch clause.</td>
<td>1 - Identical</td>
</tr>
<tr>
<td>6-4-6</td>
<td>(Required)</td>
<td>The final clause of a switch statement shall be the default-clause.</td>
<td>1 - Identical</td>
</tr>
<tr>
<td>Rule</td>
<td>Description</td>
<td>Level</td>
<td>Notes</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
<td>-------------</td>
<td>-------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>6-4-7</td>
<td>The condition of a switch statement shall not have bool type.</td>
<td>1 - Identical</td>
<td></td>
</tr>
<tr>
<td>6-4-8</td>
<td>Every switch statement shall have at least one case-clause.</td>
<td>3 - Significant differences</td>
<td>Rule reformulated. Example reworked.</td>
</tr>
<tr>
<td>6-5-1</td>
<td>A for loop shall contain a single loop-counter which shall not have floating type.</td>
<td>2 - Small differences</td>
<td>Additional note about floating types added. Rule extended.</td>
</tr>
<tr>
<td>6-5-2</td>
<td>If loop-counter is not modified by – or ++, then, within condition, the loop-counter shall only be used as an operand to &lt;=, &lt;, &gt; or &gt;=.</td>
<td>1 - Identical</td>
<td></td>
</tr>
<tr>
<td>6-5-3</td>
<td>The loop-counter shall not be modified within condition or statement.</td>
<td>1 - Identical</td>
<td></td>
</tr>
<tr>
<td>6-5-4</td>
<td>The loop-counter shall be modified by one of: –, ++, –=n, or +=n; where n remains constant for the duration of the loop.</td>
<td>1 - Identical</td>
<td></td>
</tr>
<tr>
<td>6-5-5</td>
<td>A loop-control-variable other than the loop-counter shall not be modified within condition or expression.</td>
<td>1 - Identical</td>
<td></td>
</tr>
<tr>
<td>6-5-6</td>
<td>A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.</td>
<td>1 - Identical</td>
<td></td>
</tr>
<tr>
<td>6-6-1</td>
<td>Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.</td>
<td>1 - Identical</td>
<td></td>
</tr>
<tr>
<td>6-6-2</td>
<td>The goto statement shall jump to a label declared later in the same function body.</td>
<td>1 - Identical</td>
<td></td>
</tr>
<tr>
<td>6-6-3</td>
<td>The continue statement shall only be used within a well-formed for loop.</td>
<td>1 - Identical</td>
<td></td>
</tr>
<tr>
<td>6-6-4</td>
<td>For any iteration statement there shall be no more than one break or goto statement used for loop termination.</td>
<td>4 - Rejected</td>
<td>The goto statement shall not be used, see: A6-6-1. There can be more than one break in an iteration statement.</td>
</tr>
<tr>
<td>Requirement</td>
<td>Description</td>
<td>Level</td>
<td>Notes</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>6-6-5</td>
<td>A function shall have a single point of exit at the end of the function.</td>
<td>4 - Rejected</td>
<td>Single point of exit approach does not necessarily improve readability, maintainability and testability. A function can have multiple points of exit.</td>
</tr>
<tr>
<td>7-1-1</td>
<td>A variable which is not modified shall be const qualified.</td>
<td>4 - Rejected</td>
<td>Rule replaced with A7-1-1, A7-1-2 that concern constexpr and const specifiers.</td>
</tr>
<tr>
<td>7-1-2</td>
<td>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.</td>
<td>1 - Identical</td>
<td>M7-1-2</td>
</tr>
<tr>
<td>7-2-1</td>
<td>An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.</td>
<td>2 - Small differences</td>
<td>A7-1-2 Example extended.</td>
</tr>
<tr>
<td>7-3-1</td>
<td>The global namespace shall only contain main, namespace declarations and extern &quot;C&quot; declarations.</td>
<td>1 - Identical</td>
<td>M7-3-1</td>
</tr>
<tr>
<td>7-3-2</td>
<td>The identifier main shall not be used for a function other than the global function main.</td>
<td>1 - Identical</td>
<td>M7-3-2</td>
</tr>
<tr>
<td>7-3-3</td>
<td>There shall be no unnamed namespaces in header files.</td>
<td>1 - Identical</td>
<td>M7-3-3</td>
</tr>
<tr>
<td>7-3-4</td>
<td>Using-directives shall not be used.</td>
<td>1 - Identical</td>
<td>M7-3-4</td>
</tr>
<tr>
<td>7-3-5</td>
<td>Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier.</td>
<td>1 - Identical</td>
<td>M7-3-5</td>
</tr>
<tr>
<td>7-3-6</td>
<td>Using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.</td>
<td>1 - Identical</td>
<td>M7-3-6</td>
</tr>
<tr>
<td>7-4-1</td>
<td>All usage of assembler shall be documented.</td>
<td>1 - Identical</td>
<td>M7-4-1</td>
</tr>
<tr>
<td>7-4-2</td>
<td>Assembler instructions shall only be introduced using the <code>asm</code> declaration.</td>
<td>1 - Identical</td>
<td>M7-4-2</td>
</tr>
<tr>
<td>7-4-3</td>
<td>Assembly language shall be encapsulated and isolated.</td>
<td>1 - Identical</td>
<td>M7-4-3</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Difference</td>
<td>Rationale</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>------------</td>
<td>------------</td>
</tr>
<tr>
<td>7-5-1</td>
<td>A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.</td>
<td>1 - Identical</td>
<td></td>
</tr>
<tr>
<td>7-5-2</td>
<td>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.</td>
<td>2 - Small differences</td>
<td>Added a note saying that the rule applies to std::unique_ptr, std::shared_ptr and std::weak_ptr too.</td>
</tr>
<tr>
<td>7-5-3</td>
<td>A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.</td>
<td>3 - Significant differences</td>
<td>Rule reformulated so it is allowed to return a reference or a pointer to non-const reference parameter. Rationale reformulated. Example reworked.</td>
</tr>
<tr>
<td>7-5-4</td>
<td>Functions should not call themselves, either directly or indirectly.</td>
<td>2 - Small differences</td>
<td>Obligation level changed to “Required”. Example reworked.</td>
</tr>
<tr>
<td>8-0-1</td>
<td>An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.</td>
<td>1 - Identical</td>
<td></td>
</tr>
<tr>
<td>8-3-1</td>
<td>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.</td>
<td>1 - Identical</td>
<td></td>
</tr>
<tr>
<td>8-4-1</td>
<td>Functions shall not be defined using the ellipsis notation.</td>
<td>3 - Significant differences</td>
<td>Rationale reformulated. Added a note that variadic templates should be used instead. Example extended.</td>
</tr>
<tr>
<td>8-4-2</td>
<td>The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.</td>
<td>1 - Identical</td>
<td></td>
</tr>
<tr>
<td>Rule</td>
<td>Description</td>
<td>Level</td>
<td>Code</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>8-4-3</td>
<td>All exit paths from a function with non-void return type shall have an explicit return statement with an expression.</td>
<td>2 - Small differences</td>
<td>A8-4-2</td>
</tr>
<tr>
<td>8-4-4</td>
<td>A function identifier shall either be used to call the function or it shall be preceded by &amp;.</td>
<td>1 - Identical</td>
<td>M8-4-4</td>
</tr>
<tr>
<td>8-5-1</td>
<td>All variables shall have a defined value before they are used.</td>
<td>1 - Identical</td>
<td>M8-5-1</td>
</tr>
<tr>
<td>8-5-2</td>
<td>Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.</td>
<td>1 - Identical</td>
<td>M8-5-2</td>
</tr>
<tr>
<td>8-5-3</td>
<td>In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.</td>
<td>3 - Significant differences</td>
<td>A7-2-4</td>
</tr>
<tr>
<td>9-3-1</td>
<td>Const member functions shall not return non-const pointers or references to class-data.</td>
<td>1 - Identical</td>
<td>M9-3-1</td>
</tr>
<tr>
<td>9-3-2</td>
<td>Member functions shall not return non-const handles to class-data.</td>
<td>2 - Small differences</td>
<td>A9-3-1</td>
</tr>
<tr>
<td>9-3-3</td>
<td>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.</td>
<td>1 - Identical</td>
<td>M9-3-3</td>
</tr>
<tr>
<td>9-5-1</td>
<td>Unions shall not be used.</td>
<td>1 - Identical</td>
<td>M9-5-1</td>
</tr>
<tr>
<td>9-6-1</td>
<td>When the absolute positioning of bits representing a bit-field is required, then the behavior and packing of bit-fields shall be documented.</td>
<td>1 - Identical</td>
<td>M9-6-1</td>
</tr>
<tr>
<td>9-6-2</td>
<td>Bit-fields shall be either bool type or an explicitly unsigned or signed integral type.</td>
<td>4 - Rejected</td>
<td>-</td>
</tr>
<tr>
<td>9-6-3</td>
<td>Bit-fields shall not have enum type.</td>
<td>4 - Rejected</td>
<td>-</td>
</tr>
<tr>
<td>Requirement</td>
<td>Recommendation</td>
<td>Status</td>
<td>Reason</td>
</tr>
<tr>
<td>-------------</td>
<td>----------------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>9-6-4 (Required) Named bit-fields with signed integer type shall have a length of more than one bit.</td>
<td>4 - Rejected</td>
<td>-</td>
<td>Permitted types changed. New rule introduced: A9-6-1.</td>
</tr>
<tr>
<td>10-1-1 (Advisory) Classes should not be derived from virtual bases</td>
<td>1 - Identical</td>
<td>M10-1-1</td>
<td>-</td>
</tr>
<tr>
<td>10-1-2 (Required) A base class shall only be declared virtual if it is used in a diamond hierarchy.</td>
<td>4 - Rejected</td>
<td>-</td>
<td>Multiple inheritance is not allowed.</td>
</tr>
<tr>
<td>10-1-3 (Required) An accessible base class shall not be both virtual and non-virtual in the same hierarchy.</td>
<td>1 - Identical</td>
<td>M10-1-3</td>
<td>-</td>
</tr>
<tr>
<td>10-2-1 (Advisory) All accessible entity names within a multiple inheritance hierarchy should be unique.</td>
<td>1 - Identical</td>
<td>M10-2-1</td>
<td>-</td>
</tr>
<tr>
<td>10-3-1 (Required) There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.</td>
<td>4 - Rejected</td>
<td>-</td>
<td>Rule already covered by A10-1-1 and A10-3-1</td>
</tr>
<tr>
<td>10-3-2 (Required) Each overriding virtual function shall be declared with the virtual keyword.</td>
<td>3 - Significant differences</td>
<td>A10-3-2</td>
<td>Rule and rationale reformulated so the override specifier should be used instead of virtual keyword. Example reworked.</td>
</tr>
<tr>
<td>10-3-3 (Required) A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.</td>
<td>1 - Identical</td>
<td>M10-3-3</td>
<td>-</td>
</tr>
<tr>
<td>11-0-1 (Required) Member data in non-POD class types shall be private.</td>
<td>1 - Identical</td>
<td>M11-0-1</td>
<td>-</td>
</tr>
<tr>
<td>12-1-1 (Required) An object’s dynamic type shall not be used from the body of its constructor or destructor.</td>
<td>1 - Identical</td>
<td>M12-1-1</td>
<td>-</td>
</tr>
<tr>
<td>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.</td>
<td>3 - Significant differences</td>
<td>A12-1-1</td>
<td>Obligation level changed to “Required”. Rule reformulated to cover non-static class data members. Rationale reformulated. Example reworked.</td>
</tr>
<tr>
<td>12-1-3 (Required) All constructors that are callable with a single argument of fundamental type shall be declared explicit.</td>
<td>2 - Small differences</td>
<td>A12-1-4</td>
<td>Example reworked.</td>
</tr>
<tr>
<td>Rule</td>
<td>Description</td>
<td>Significant Differences</td>
<td>Rejected</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td>-------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>A12-8-1</td>
<td>A copy constructor shall only initialize its base classes and the non-static members of the class of which it is a member.</td>
<td>3 - Significant differences</td>
<td>A12-8-1</td>
</tr>
<tr>
<td>A12-8-1</td>
<td>The copy assignment operator shall be declared protected or private in an abstract class.</td>
<td>3 - Significant differences</td>
<td>A12-8-1</td>
</tr>
<tr>
<td>A12-8-2</td>
<td>The copy assignment operator shall be declared protected or private in an abstract class.</td>
<td>3 - Significant differences</td>
<td>A12-8-6</td>
</tr>
<tr>
<td>A14-5-1</td>
<td>A non-member generic function shall only be declared in a namespace that is not an associated namespace.</td>
<td>4 - Rejected</td>
<td>-</td>
</tr>
<tr>
<td>M14-5-2</td>
<td>A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter.</td>
<td>1 - Identical</td>
<td>M14-5-2</td>
</tr>
<tr>
<td>M14-5-3</td>
<td>A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.</td>
<td>1 - Identical</td>
<td>M14-5-3</td>
</tr>
<tr>
<td>M14-6-1</td>
<td>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;.</td>
<td>1 - Identical</td>
<td>M14-6-1</td>
</tr>
<tr>
<td>Rule</td>
<td>Description</td>
<td>Level</td>
<td>Status</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td>14-6-2</td>
<td>The function chosen by overload resolution shall resolve to a function declared previously in the translation unit.</td>
<td>4 - Rejected</td>
<td>-</td>
</tr>
<tr>
<td>14-7-1</td>
<td>All class templates, function templates, class template member functions and class template static members shall be instantiated at least once.</td>
<td>4 - Rejected</td>
<td>-</td>
</tr>
<tr>
<td>14-7-2</td>
<td>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.</td>
<td>3 - Significant differences</td>
<td>A14-7-1</td>
</tr>
<tr>
<td>14-7-3</td>
<td>All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template.</td>
<td>1 - Identical</td>
<td>M14-7-3</td>
</tr>
<tr>
<td>14-8-1</td>
<td>Overloaded function templates shall not be explicitly specialized.</td>
<td>1 - Identical</td>
<td>M14-8-1</td>
</tr>
<tr>
<td>14-8-2</td>
<td>The viable function set for a function call should either contain no function specializations, or only contain function specializations.</td>
<td>3 - Small differences</td>
<td>A14-8-1</td>
</tr>
<tr>
<td>15-0-1</td>
<td>Exceptions shall only be used for error handling.</td>
<td>3 - Significant differences</td>
<td>A15-0-1</td>
</tr>
<tr>
<td>15-0-2</td>
<td>An exception object should not have pointer type.</td>
<td>3 - Significant differences</td>
<td>A15-1-2</td>
</tr>
<tr>
<td>15-0-3</td>
<td>Control shall not be transferred into a try or catch block using a goto or a switch statement.</td>
<td>1 - Identical</td>
<td>M15-0-3</td>
</tr>
<tr>
<td>15-1-1</td>
<td>The assignment-expression of a throw statement shall not itself cause an exception.</td>
<td>1 - Identical</td>
<td>M15-1-1</td>
</tr>
<tr>
<td>15-1-2</td>
<td>NULL shall not be thrown explicitly.</td>
<td>1 - Identical</td>
<td>M15-1-2</td>
</tr>
<tr>
<td>15-1-3</td>
<td>An empty throw (throw;) shall only be used in the compound-statement of a catch handler.</td>
<td>1 - Identical</td>
<td>M15-1-3</td>
</tr>
<tr>
<td>15-3-1 (Required) Exceptions shall be raised only after start-up and before termination of the program.</td>
<td>1 - Identical</td>
<td>M15-3-1</td>
<td>-</td>
</tr>
<tr>
<td>15-3-2 (Advisory) There should be at least one exception handler to catch all otherwise unhandled exceptions.</td>
<td>2 - Small differences</td>
<td>A15-3-3</td>
<td>Obligation level changed. Rule extended to cover multi-threading.</td>
</tr>
<tr>
<td>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.</td>
<td>1 - Identical</td>
<td>M15-3-3</td>
<td>-</td>
</tr>
<tr>
<td>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.</td>
<td>1 - Identical</td>
<td>M15-3-4</td>
<td>-</td>
</tr>
<tr>
<td>15-3-5 (Required) A class type exception shall always be caught by reference.</td>
<td>2 - Small differences</td>
<td>A15-3-5</td>
<td>Possibility to catch by const reference added</td>
</tr>
<tr>
<td>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.</td>
<td>1 - Identical</td>
<td>M15-3-6</td>
<td>-</td>
</tr>
<tr>
<td>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.</td>
<td>1 - Identical</td>
<td>M15-3-7</td>
<td>-</td>
</tr>
<tr>
<td>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.</td>
<td>4 - Rejected</td>
<td>-</td>
<td>Dynamic exception specification was prohibited. Noexcept specifier shall be used instead.</td>
</tr>
<tr>
<td>15-5-1 (Required) A class destructor shall not exit with an exception.</td>
<td>3 - Significant differences</td>
<td>A15-5-1</td>
<td>Rule significantly extended with other special functions and operators.</td>
</tr>
<tr>
<td>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).</td>
<td>4-Rejected</td>
<td>-</td>
<td>Dynamic exception specification was prohibited.</td>
</tr>
<tr>
<td>15-5-3 (Required) The std::terminate() function shall not be called implicitly.</td>
<td>2 - Small differences</td>
<td>A15-5-3</td>
<td>Rationale and example extended.</td>
</tr>
<tr>
<td>Rule</td>
<td>Description</td>
<td>Recommendation</td>
<td>Rule Notation</td>
</tr>
<tr>
<td>------</td>
<td>------------------------------------------------------------------------------</td>
<td>----------------</td>
<td>--------------</td>
</tr>
<tr>
<td>16-0-1</td>
<td>(Required) #include directives in a file shall only be preceded by other preprocessor directives or comments.</td>
<td>1 - Identical</td>
<td>M16-0-1</td>
</tr>
<tr>
<td>16-0-2</td>
<td>(Required) Macros shall only be #define’d or #undef’d in the global namespace.</td>
<td>1 - Identical</td>
<td>M16-0-2</td>
</tr>
<tr>
<td>16-0-3</td>
<td>(Required) #undef shall not be used.</td>
<td>4 - Rejected</td>
<td>-</td>
</tr>
<tr>
<td>16-0-4</td>
<td>(Required) Function-like macros shall not be defined.</td>
<td>4 - Rejected</td>
<td>-</td>
</tr>
<tr>
<td>16-0-5</td>
<td>(Required) Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.</td>
<td>1 - Identical</td>
<td>M16-0-5</td>
</tr>
<tr>
<td>16-0-6</td>
<td>(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 ##.</td>
<td>1 - Identical</td>
<td>M16-0-6</td>
</tr>
<tr>
<td>16-0-7</td>
<td>(Required) Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.</td>
<td>1 - Identical</td>
<td>M16-0-7</td>
</tr>
<tr>
<td>16-0-8</td>
<td>(Required) If the # token appears as the first token on a line, then it shall be immediately followed by a pre-processing token.</td>
<td>1 - Identical</td>
<td>M16-0-8</td>
</tr>
<tr>
<td>16-1-1</td>
<td>(Required) The defined preprocessor operator shall only be used in one of the two standard forms.</td>
<td>1 - Identical</td>
<td>M16-1-1</td>
</tr>
<tr>
<td>16-1-2</td>
<td>(Required) All #else, #elif and #endif pre-processor directives shall reside in the same file as the #if or ifndef directive to which they are related.</td>
<td>1 - Identical</td>
<td>M16-1-2</td>
</tr>
<tr>
<td>16-2-1</td>
<td>(Required) The pre-processor shall only be used for file inclusion and include guards.</td>
<td>4 - Rejected</td>
<td>-</td>
</tr>
<tr>
<td>16-2-2</td>
<td>(Required) C++ macros shall only be used for include guards, type qualifiers, or storage class specifiers.</td>
<td>4 - Rejected</td>
<td>-</td>
</tr>
<tr>
<td>16-2-3</td>
<td>(Required) Include guards shall be provided</td>
<td>1 - Identical</td>
<td>M16-2-3</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Changes</td>
<td>Related MISRA</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>---------</td>
<td>---------------</td>
</tr>
<tr>
<td>16-2-4</td>
<td>(Required) The <code>, </code> <code>, </code> / or // characters shall not occur in a header file name.</td>
<td>2 - Small differences</td>
<td>A16-2-1</td>
</tr>
<tr>
<td>16-2-5</td>
<td>(Advisory) The character <code>\</code> should not occur in a header file name.</td>
<td>2 - Small differences</td>
<td>A16-2-1</td>
</tr>
<tr>
<td>16-2-6</td>
<td>(Required) The <code>#include</code> directive shall be followed by either a <code>&lt;filename&gt;</code> or “filename” sequence.</td>
<td>4 - Rejected</td>
<td>-</td>
</tr>
<tr>
<td>16-3-1</td>
<td>(Required) There shall be at most one occurrence of the <code>#</code> or <code>##</code> operators in a single macro definition.</td>
<td>1 - Identical</td>
<td>M16-3-1</td>
</tr>
<tr>
<td>16-3-2</td>
<td>(Advisory) The <code>#</code> and <code>##</code> operators should not be used.</td>
<td>1 - Identical</td>
<td>M16-3-2</td>
</tr>
<tr>
<td>16-6-1</td>
<td>(Required) All uses of the <code>#pragma</code> directive shall be documented.</td>
<td>4 - Rejected</td>
<td>-</td>
</tr>
<tr>
<td>17-0-1</td>
<td>(Required) Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.</td>
<td>2 - Small differences</td>
<td>A17-0-1</td>
</tr>
<tr>
<td>17-0-2</td>
<td>(Required) The names of standard library macros and objects shall not be reused.</td>
<td>1 - Identical</td>
<td>M17-0-2</td>
</tr>
<tr>
<td>17-0-3</td>
<td>(Required) The names of standard library functions shall not be overridden.</td>
<td>1 - Identical</td>
<td>M17-0-3</td>
</tr>
<tr>
<td>17-0-4</td>
<td>(Required) All library code shall conform to MISRA C++.</td>
<td>4 - Rejected</td>
<td>-</td>
</tr>
<tr>
<td>17-0-5</td>
<td>(Required) The <code>setjmp</code> macro and the <code>longjmp</code> function shall not be used.</td>
<td>1 - Identical</td>
<td>M17-0-5</td>
</tr>
<tr>
<td>18-0-1</td>
<td>(Required) The C library shall not be used.</td>
<td>2 - Small differences</td>
<td>A18-0-1</td>
</tr>
<tr>
<td>18-0-2</td>
<td>(Required) The library functions <code>atof</code>, <code>atoi</code> and <code>atol</code> from library <code>&lt;cstdlib&gt;</code> shall not be used.</td>
<td>2 - Small differences</td>
<td>A18-0-2</td>
</tr>
</tbody>
</table>
18-0-3 (Required) The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.

18-0-4 (Required) The time handling functions of library <ctime> shall not be used.

18-0-5 (Required) The unbounded functions of library <cstring> shall not be used.

18-2-1 (Required) The macro offsetof shall not be used.

18-4-1 (Required) Dynamic heap memory allocation shall not be used.

18-7-1 (Required) The signal handling facilities of <csignal> shall not be used.

19-3-1 (Required) The error indicator errno shall not be used.

27-0-1 (Required) The stream input/output library <cstdio> shall not be used.

<table>
<thead>
<tr>
<th>HIC++ Rule:</th>
<th>Relation type:</th>
<th>Related rule:</th>
<th>Comment:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.1 Ensure that code complies with the 2011 ISO C++ Language Standard.</td>
<td>2 - Small differences</td>
<td>A1-1-1</td>
<td>Specified that the code shall conform to ISO/IEC 14882:2014</td>
</tr>
<tr>
<td>1.2.1 Ensure that all statements are reachable.</td>
<td>2 - Small differences</td>
<td>M0-1-1</td>
<td></td>
</tr>
<tr>
<td>1.2.2 Ensure that no expression or sub-expression is redundant.</td>
<td>2 - Small differences</td>
<td>M0-1-9</td>
<td></td>
</tr>
<tr>
<td>1.3.1 Do not use the increment operator (++) on a variable of type bool</td>
<td>2 - Small differences</td>
<td>M4-5-1</td>
<td></td>
</tr>
<tr>
<td>1.3.2 Do not use the register keyword.</td>
<td>2 - Small differences</td>
<td>A7-1-4</td>
<td></td>
</tr>
</tbody>
</table>

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.
<table>
<thead>
<tr>
<th>Section</th>
<th>Rule</th>
<th>Differences</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.3.3</td>
<td>Do not use the C Standard Library .h headers</td>
<td>2 - Small differences</td>
<td>A18-0-1</td>
</tr>
<tr>
<td>1.3.4</td>
<td>Do not use deprecated STL library features</td>
<td>2 - Small differences</td>
<td>A1-1-1, A18-1-3, A18-9-1</td>
</tr>
<tr>
<td>1.3.5</td>
<td>Do not use throw exception specifications.</td>
<td>2 - Small differences</td>
<td>A15-4-1</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Do not use tab characters in source files.</td>
<td>4 - Rejected</td>
<td>AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Do not use digraphs or trigraphs.</td>
<td>2 - Small differences</td>
<td>A2-5-1, A2-6-1</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Do not use the C comment delimiters /* ... */.</td>
<td>2 - Small differences</td>
<td>A2-8-4</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Do not comment out code.</td>
<td>2 - Small differences</td>
<td>A2-8-2</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Ensure that each identifier is distinct from any other visible identifier.</td>
<td>2 - Small differences</td>
<td>M2-10-1</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Do not concatenate strings with different encoding prefixes.</td>
<td>2 - Small differences</td>
<td>A2-14-2</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Do not use octal constants (other than zero).</td>
<td>2 - Small differences</td>
<td>M2-13-2</td>
</tr>
<tr>
<td>2.5.3</td>
<td>Use nullptr for the null pointer constant.</td>
<td>2 - Small differences</td>
<td>A4-10-1</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Do not hide declarations.</td>
<td>2 - Small differences</td>
<td>A2-11-1</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Do not declare functions at block scope.</td>
<td>2 - Small differences</td>
<td>M3-1-2</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Do not use variables with static storage duration.</td>
<td>3 - Significant differences</td>
<td>A3-3-2</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Do not return a reference or a pointer to an automatic variable defined within the function.</td>
<td>2 - Small differences</td>
<td>M7-5-1</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Do not assign the address of a variable to a pointer with a greater lifetime.</td>
<td>2 - Small differences</td>
<td>M7-5-2</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Use RAII for resources.</td>
<td>4 - Rejected</td>
<td>AUTOSAR C++ Coding Guidelines does not define rules for coding patterns. Note that usage of RAII is recommended, see: A15-1-4.</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Do not make any assumptions about the internal representation of a value or object.</td>
<td>2 - Small differences</td>
<td>A3-9-1, M3-9-3, M5-0-15, M5-0-21, M9-5-1, M18-2-1</td>
</tr>
<tr>
<td>Section</td>
<td>Text</td>
<td>Notes</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>4.1.1</td>
<td>Ensure that a function argument does not undergo an array-to-pointer conversion.</td>
<td>2 - Small differences M5-2-12</td>
<td></td>
</tr>
<tr>
<td>4.2.1</td>
<td>Ensure that the U suffix is applied to a literal used in a context requiring an unsigned integral expression.</td>
<td>2 - Small differences M2-13-2</td>
<td></td>
</tr>
<tr>
<td>4.2.2</td>
<td>Ensure that data loss does not demonstrably occur in an integral expression.</td>
<td>2 - Small differences A4-7-1, M5-0-4, M5-0-6, M5-0-9</td>
<td></td>
</tr>
<tr>
<td>4.3.1</td>
<td>Do not convert an expression of wider floating point type to a narrower floating point type.</td>
<td>2 - Small differences A4-7-1, M5-0-6</td>
<td></td>
</tr>
<tr>
<td>4.4.1</td>
<td>Do not convert floating values to integral types except through use of standard library functions.</td>
<td>4 - Rejected Rules that are related: M5-0-3, M5-0-5, M5-0-6, M5-0-7</td>
<td></td>
</tr>
<tr>
<td>5.1.1</td>
<td>Use symbolic names instead of literal values in code.</td>
<td>2 - Small differences A5-1-1</td>
<td></td>
</tr>
<tr>
<td>5.1.2</td>
<td>Do not rely on the sequence of evaluation within an expression.</td>
<td>2 - Small differences A5-0-1</td>
<td></td>
</tr>
<tr>
<td>5.1.3</td>
<td>Use parentheses in expressions to specify the intent of the expression.</td>
<td>2 - Small differences A5-0-1, M5-2-1, M5-2-10</td>
<td></td>
</tr>
<tr>
<td>5.1.4</td>
<td>Do not capture variables implicitly in a lambda.</td>
<td>2 - Small differences A5-1-2</td>
<td></td>
</tr>
<tr>
<td>5.1.5</td>
<td>Include a (possibly empty) parameter list in every lambda expression.</td>
<td>2 - Small differences A5-1-3</td>
<td></td>
</tr>
<tr>
<td>5.1.6</td>
<td>Do not code side effects into the right-hand operands of: &amp;&amp;,</td>
<td></td>
<td>, sizeof, typeid or a function passed to condition_variable::wait.</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Ensure that pointer or array access is demonstrably within bounds of a valid object.</td>
<td>2 - Small differences A5-2-5</td>
<td></td>
</tr>
<tr>
<td>5.2.2</td>
<td>Ensure that functions do not call themselves, either directly or indirectly.</td>
<td>2 - Small differences A7-5-2</td>
<td></td>
</tr>
<tr>
<td>5.3.1</td>
<td>Do not apply unary minus to operands of unsigned type.</td>
<td>2 - Small differences M5-3-2</td>
<td></td>
</tr>
<tr>
<td>5.3.2</td>
<td>Allocate memory using new and release it using delete.</td>
<td>2 - Small differences A18-5-1 Note that operators new and delete shall not be used explicitly, see: A18-5-2.</td>
<td></td>
</tr>
<tr>
<td>Section</td>
<td>Rule</td>
<td>Difference</td>
<td>Note</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td>------------</td>
<td>------</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Ensure that the form of delete matches the form of new used to allocate the memory.</td>
<td>2 - Small differences</td>
<td>A18-5-3 Note that operators new and delete shall not be used explicitly, see: A18-5-2.</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Only use casting forms: static_cast (excl. void*), dynamic_cast or explicit constructor call.</td>
<td>2 - Small differences</td>
<td>A5-2-1, A5-2-2, A5-2-3, A5-2-4</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Do not cast an expression to an enumeration type.</td>
<td>4 - Rejected</td>
<td>It is allowed to cast an expression to an enumeration type, but an expression shall have a value that corresponds to an enumerator of the enumeration, see: A7-2-1.</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Do not convert from a base class to a derived class.</td>
<td>3 - Small differences</td>
<td>M5-2-2, M5-2-3, A5-2-1 Note that the dynamic_cast is unsuitable for use with real-time systems.</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Ensure that the right hand operand of the division or remainder operators is demonstrably non-zero.</td>
<td>2 - Small differences</td>
<td>A5-5-1</td>
</tr>
<tr>
<td>5.6.1</td>
<td>Do not use bitwise operators with signed operands.</td>
<td>2 - Small differences</td>
<td>M5-0-21</td>
</tr>
<tr>
<td>5.7.1</td>
<td>Do not write code that expects floating point calculations to yield exact results.</td>
<td>2 - Small differences</td>
<td>M6-2-2</td>
</tr>
<tr>
<td>5.7.2</td>
<td>Ensure that a pointer to member that is a virtual function is only compared (==) with nullptr.</td>
<td>2 - Small differences</td>
<td>A5-10-1</td>
</tr>
<tr>
<td>5.8.1</td>
<td>Do not use the conditional operator (?:) as a sub-expression.</td>
<td>2 - Small differences</td>
<td>A5-16-1</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Enclose the body of a selection or an iteration statement in a compound statement.</td>
<td>2 - Small differences</td>
<td>M6-3-1, M6-4-1</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Explicitly cover all paths through multi-way selection statements.</td>
<td>2 - Small differences</td>
<td>M6-4-2</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Ensure that a non-empty case statement block does not fall through to the next label.</td>
<td>2 - Small differences</td>
<td>M6-4-5</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Ensure that a switch statement has at least two case labels, distinct from the default label.</td>
<td>2 - Small differences</td>
<td>A6-4-1</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Implement a loop that only uses element values as a range-based loop.</td>
<td>2 - Small differences</td>
<td>A6-5-1</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Ensure that a loop has a single loop counter, an optional control variable, and is not degenerate.</td>
<td>2 - Small differences</td>
<td>A6-5-2</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Differences</td>
<td>Code</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Do not alter a control or counter variable more than once in a loop.</td>
<td>3 - Significant differences</td>
<td>M6-5-3</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Only modify a for loop counter in the for expression.</td>
<td>2 - Small differences</td>
<td>M6-5-3</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Ensure that the label(s) for a jump statement or a switch condition appear later, in the same or an enclosing block.</td>
<td>2 - Small differences</td>
<td>M6-6-1</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Ensure that execution of a function with a non-void return type ends in a return statement with a value.</td>
<td>2 - Small differences</td>
<td>A8-4-2</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Postpone variable definitions as long as possible.</td>
<td>2 - Small differences</td>
<td>M3-4-1</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Declare each identifier on a separate line in a separate declaration.</td>
<td>2 - Small differences</td>
<td>A7-1-7</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Use const whenever possible.</td>
<td>2 - Small differences</td>
<td>A7-1-1, A7-1-2</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Do not place type specifiers before non-type specifiers in a declaration.</td>
<td>2 - Small differences</td>
<td>A7-1-8</td>
</tr>
<tr>
<td>7.1.4</td>
<td>Place CV-qualifiers on the right hand side of the type they apply to.</td>
<td>2 - Small differences</td>
<td>A7-1-3</td>
</tr>
<tr>
<td>7.1.5</td>
<td>Do not inline large functions.</td>
<td>4 - Rejected</td>
<td></td>
</tr>
<tr>
<td>7.1.6</td>
<td>Use class types or typedefs to abstract scalar quantities and standard integer types.</td>
<td>3 - Significant differences</td>
<td>A3-9-1</td>
</tr>
<tr>
<td>7.1.7</td>
<td>Use a trailing return type in preference to type disambiguation using typename.</td>
<td>2 - Small differences</td>
<td>A8-2-1</td>
</tr>
<tr>
<td>7.1.8</td>
<td>Use auto id = expr when declaring a variable to have the same type as its initializer function call.</td>
<td>3 - Significant differences</td>
<td>A7-1-5</td>
</tr>
<tr>
<td>7.1.9</td>
<td>Do not explicitly specify the return type of a lambda.</td>
<td>4 - Rejected</td>
<td></td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Differences</td>
<td>Reference</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>-------------</td>
<td>-----------</td>
</tr>
<tr>
<td>7.1.10</td>
<td>Use <code>static_assert</code> for assertions involving compile time constants.</td>
<td>3 - Significant differences</td>
<td>A16-6-1</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Use an explicit enumeration base and ensure that it is large enough to store all enumerators.</td>
<td>2 - Small differences</td>
<td>A7-2-2</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Initialize none, the first only or all enumerators in an enumeration.</td>
<td>2 - Small differences</td>
<td>A7-2-4</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Do not use using directives.</td>
<td>2 - Small differences</td>
<td>M7-3-4</td>
</tr>
<tr>
<td>7.4.1</td>
<td>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.</td>
<td>2 - Small differences</td>
<td>A3-1-1, M3-2-2</td>
</tr>
<tr>
<td>7.4.2</td>
<td>Ensure that an inline function, a function template, or a type used from multiple translation units is defined in a single header file.</td>
<td>2 - Small differences</td>
<td>A3-1-1, M3-2-4</td>
</tr>
<tr>
<td>7.4.3</td>
<td>Ensure that an object or a function used from multiple translation units is declared in a single header file.</td>
<td>2 - Small differences</td>
<td>A7-4-1</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Do not use multiple levels of pointer indirection.</td>
<td>3 - Significant differences</td>
<td>A5-0-3</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Make parameter names absent or identical in all declarations.</td>
<td>2 - Small differences</td>
<td>M3-9-1</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Do not declare functions with an excessive number of parameters.</td>
<td>4 - Rejected</td>
<td>Code metrics are not covered by AUTOSAR C++ Coding Guidelines.</td>
</tr>
<tr>
<td>8.2.3</td>
<td>Pass small objects with a trivial copy constructor by value.</td>
<td>4 - Rejected</td>
<td>The rule is vague, “small” has no technical meaning.</td>
</tr>
<tr>
<td>8.2.4</td>
<td>Do not pass <code>std::unique_ptr</code> by const reference.</td>
<td>2 - Small differences</td>
<td>A18-1-5</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Do not write functions with an excessive McCabe Cyclomatic Complexity.</td>
<td>4 - Rejected</td>
<td>Code metrics are not covered by AUTOSAR C++ Coding Guidelines.</td>
</tr>
<tr>
<td>8.3.2</td>
<td>Do not write functions with a high static program path count.</td>
<td>4 - Rejected</td>
<td>Code metrics are not covered by AUTOSAR C++ Coding Guidelines.</td>
</tr>
<tr>
<td>Section</td>
<td>Rule Description</td>
<td>Level</td>
<td>Comments</td>
</tr>
<tr>
<td>---------</td>
<td>------------------</td>
<td>-------</td>
<td>----------</td>
</tr>
<tr>
<td>8.3.3</td>
<td>Do not use default arguments.</td>
<td>4 - Rejected</td>
<td>Using default arguments is allowed with some restrictions, see e.g. M8-3-1.</td>
</tr>
<tr>
<td>8.3.4</td>
<td>Define =delete functions with parameters of type rvalue reference to const.</td>
<td>2 - Small differences</td>
<td>A13-3-1, A18-9-3</td>
</tr>
<tr>
<td>8.4.1</td>
<td>Do not access an invalid object or an object with indeterminate value.</td>
<td>2 - Small differences</td>
<td>M8-5-1, A12-8-3</td>
</tr>
<tr>
<td>8.4.2</td>
<td>Ensure that a braced aggregate initializer matches the layout of the aggregate object.</td>
<td>2 - Small differences</td>
<td>M8-5-2</td>
</tr>
<tr>
<td>9.1.1</td>
<td>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.</td>
<td>2 - Small differences</td>
<td>M9-3-3</td>
</tr>
<tr>
<td>9.1.2</td>
<td>Make default arguments the same or absent when overriding a virtual function.</td>
<td>2 - Small differences</td>
<td>M8-3-1</td>
</tr>
<tr>
<td>9.1.3</td>
<td>Do not return non-const handles to class data from const member functions.</td>
<td>2 - Small differences</td>
<td>M9-3-1, A9-3-1</td>
</tr>
<tr>
<td>9.1.4</td>
<td>Do not write member functions which return non-const handles to data less accessible than the member function.</td>
<td>3 - Significant differences</td>
<td>A9-3-1</td>
</tr>
<tr>
<td>9.1.5</td>
<td>Do not introduce virtual functions in a final class.</td>
<td>2 - Small differences</td>
<td>A10-3-3</td>
</tr>
<tr>
<td>9.2.1</td>
<td>Declare bit-fields with an explicitly unsigned integral or enumeration type.</td>
<td>2 - Small differences</td>
<td>A9-6-1</td>
</tr>
<tr>
<td>10.1.1</td>
<td>Ensure that access to base class subobjects does not require explicit disambiguation.</td>
<td>3 - Significant differences</td>
<td>A10-1-1</td>
</tr>
<tr>
<td>10.2.1</td>
<td>Use the override special identifier when overriding a virtual function.</td>
<td>2 - Small differences</td>
<td>A10-3-2</td>
</tr>
<tr>
<td>10.3.1</td>
<td>Ensure that a derived class has at most one base class which is not an interface class.</td>
<td>2 - Small differences</td>
<td>A10-1-1</td>
</tr>
<tr>
<td>11.1.1</td>
<td>Declare all data members private.</td>
<td>2 - Small differences</td>
<td>M11-0-1</td>
</tr>
<tr>
<td>11.2.1</td>
<td>Do not use friend declarations.</td>
<td>2 - Small differences</td>
<td>A11-3-1</td>
</tr>
<tr>
<td>Section</td>
<td>Rule Description</td>
<td>Differences</td>
<td>Rule ID</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>12.1.1</td>
<td>Do not declare implicit user-defined conversions.</td>
<td>3 - Significant differences</td>
<td>A12-1-4</td>
</tr>
<tr>
<td>12.2.1</td>
<td>Declare virtual, private or protected the destructor of a type used as a base class.</td>
<td>3 - Significant differences</td>
<td>A12-4-1</td>
</tr>
<tr>
<td>12.3.1</td>
<td>Correctly declare overloads for operator new and delete.</td>
<td>4 - Rejected</td>
<td></td>
</tr>
<tr>
<td>12.4.1</td>
<td>Do not use the dynamic type of an object unless the object is fully constructed.</td>
<td>2 - Small differences</td>
<td>M12-1-1</td>
</tr>
<tr>
<td>12.4.2</td>
<td>Ensure that a constructor initializes explicitly all base classes and non-static data members.</td>
<td>2 - Small differences</td>
<td>A12-1-1</td>
</tr>
<tr>
<td>12.4.3</td>
<td>Do not specify both an NSDMI and a member initializer in a constructor for the same non-static member.</td>
<td>2 - Significant differences</td>
<td>A12-1-2</td>
</tr>
<tr>
<td>12.4.4</td>
<td>Write members in an initialization list in the order in which they are declared.</td>
<td>2 - Small differences</td>
<td>A8-5-1</td>
</tr>
<tr>
<td>12.4.5</td>
<td>Use delegating constructors to reduce code duplication.</td>
<td>4 - Rejected</td>
<td>Non-generic design principle; There is no need for a new rule.</td>
</tr>
<tr>
<td>12.5.1</td>
<td>Define explicitly =default or =delete implicit special member functions of concrete classes.</td>
<td>3 - Significant differences</td>
<td>A12-0-1</td>
</tr>
<tr>
<td>12.5.2</td>
<td>Define special members =default if the behavior is equivalent.</td>
<td>2 - Small differences</td>
<td>A12-7-1</td>
</tr>
<tr>
<td>12.5.3</td>
<td>Ensure that a user defined move/copy constructor only moves/copies base and member objects.</td>
<td>2 - Small differences</td>
<td>A12-8-1</td>
</tr>
<tr>
<td>12.5.4</td>
<td>Declare noexcept the move constructor and move assignment operator.</td>
<td>3 - Significant differences</td>
<td>A15-5-1</td>
</tr>
<tr>
<td>12.5.5</td>
<td>Correctly reset moved-from handles to resources in the move constructor.</td>
<td>2 - Small differences</td>
<td>A12-8-1</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Difference Level</td>
<td>Rule Code</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>-----------------</td>
<td>-----------</td>
</tr>
<tr>
<td>12.5.6</td>
<td>Use an atomic, non-throwing swap operation to implement the copy and move assignment operators.</td>
<td>2 - Small differences</td>
<td>A12-8-2</td>
</tr>
<tr>
<td>12.5.7</td>
<td>Declare assignment operators with the ref-qualifier &amp;.</td>
<td>2 - Small differences</td>
<td>A12-8-7</td>
</tr>
<tr>
<td>12.5.8</td>
<td>Make the copy assignment operator of an abstract class protected or define it =delete.</td>
<td>3 - Significant differences</td>
<td>A12-8-6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AUTOSAR C++ Coding Guidelines requires additional functions to be comply with this rule.</td>
<td>A12-8-6</td>
</tr>
<tr>
<td>13.1.1</td>
<td>Ensure that all overloads of a function are visible from where it is called.</td>
<td>2 - Small differences</td>
<td>M7-3-5</td>
</tr>
<tr>
<td>13.1.2</td>
<td>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.</td>
<td>3 - Significant differences</td>
<td>A13-3-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A function taking “forwarding reference” shall not be overloaded.</td>
<td>A13-3-1</td>
</tr>
<tr>
<td>13.2.1</td>
<td>Do not overload operators with special semantics.</td>
<td>2 - Small differences</td>
<td>M5-2-11, M5-3-3</td>
</tr>
<tr>
<td>13.2.2</td>
<td>Ensure that the return type of an overloaded binary operator matches the built-in counterparts.</td>
<td>2 - Small differences</td>
<td>A13-2-1, A13-2-2, A13-2-3</td>
</tr>
<tr>
<td>13.2.3</td>
<td>Declare binary arithmetic and bitwise operators as non-members.</td>
<td>4 - Rejected</td>
<td>Non-generic design principle; There is no need for a new rule.</td>
</tr>
<tr>
<td>13.2.4</td>
<td>When overloading the subscript operator (operator[]) implement both const and non-const versions.</td>
<td>2 - Small differences</td>
<td>A13-5-1</td>
</tr>
<tr>
<td>13.2.5</td>
<td>Implement a minimal set of operators and use them to implement all other related operators.</td>
<td>4 - Rejected</td>
<td>Non-generic design principle; There is no need for a new rule.</td>
</tr>
<tr>
<td>14.1.1</td>
<td>Use variadic templates rather than an ellipsis.</td>
<td>3 - Significant differences</td>
<td>A8-4-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AUTOSAR C++ Coding Guidelines prohibits usage of variadic arguments.</td>
<td>A8-4-1</td>
</tr>
<tr>
<td>14.2.1</td>
<td>Declare template specializations in the same file as the primary template they specialize.</td>
<td>2 - Small differences</td>
<td>M14-7-3</td>
</tr>
<tr>
<td>14.2.2</td>
<td>Do not explicitly specialize a function template that is overloaded with other templates.</td>
<td>2 - Small differences</td>
<td>M14-8-1</td>
</tr>
<tr>
<td>14.2.3</td>
<td>Declare extern an explicitly instantiated template.</td>
<td>4 - Rejected</td>
<td></td>
</tr>
</tbody>
</table>

---

305 of 371

Document ID 839: AUTOSAR_RS_CPP14Guidelines

— AUTOSAR CONFIDENTIAL —
<table>
<thead>
<tr>
<th>Section</th>
<th>Rule</th>
<th>Type of Difference</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.1.1</td>
<td>Only use instances of std::exception for exceptions.</td>
<td>2 - Small differences</td>
<td>A15-1-1</td>
</tr>
<tr>
<td>15.2.1</td>
<td>Do not throw an exception from a destructor.</td>
<td>2 - Small differences</td>
<td>A15-5-1</td>
</tr>
<tr>
<td>15.3.1</td>
<td>Do not access non-static members from a catch handler of constructor/destructor function try block.</td>
<td>2 - Small differences</td>
<td>M15-3-3</td>
</tr>
<tr>
<td>15.3.2</td>
<td>Ensure that a program does not result in a call to std::terminate.</td>
<td>2 - Small differences</td>
<td>A15-5-2, A15-5-3</td>
</tr>
<tr>
<td>16.1.1</td>
<td>Use the preprocessor only for implementing include guards, and including header files with include guards.</td>
<td>3 - Significant differences</td>
<td>A16-0-1</td>
</tr>
<tr>
<td>16.1.2</td>
<td>Do not include a path specifier in filenames supplied in #include directives.</td>
<td>3 - Significant differences</td>
<td>A16-2-1</td>
</tr>
<tr>
<td>16.1.3</td>
<td>Include directly the minimum number of headers required for compilation.</td>
<td>4 - Rejected</td>
<td></td>
</tr>
<tr>
<td>16.1.4</td>
<td>Use &lt;&gt; brackets for system and standard library headers. Use quotes for all other headers.</td>
<td>4 - Rejected</td>
<td>The rule defines a coding style. Anyway, these are the only forms allowed by the C++ Language Standard. No need for a new rule.</td>
</tr>
<tr>
<td>16.1.5</td>
<td>Use std::forward to forward universal references.</td>
<td>2 - Small differences</td>
<td>A18-9-3</td>
</tr>
<tr>
<td>17.1.1</td>
<td>Do not use std::vector&lt;bool&gt;.</td>
<td>2 - Small differences</td>
<td>A18-1-2</td>
</tr>
<tr>
<td>17.2.1</td>
<td>Wrap use of the C Standard Library.</td>
<td>2 - Small differences</td>
<td>A17-1-1</td>
</tr>
<tr>
<td>17.3.1</td>
<td>Do not use std::move on objects declared with const or const &amp; type.</td>
<td>2 - Small differences</td>
<td>A18-9-3</td>
</tr>
<tr>
<td>17.3.2</td>
<td>Use std::forward to forward universal references.</td>
<td>2 - Small differences</td>
<td>A18-9-2</td>
</tr>
<tr>
<td>17.3.3</td>
<td>Do not subsequently use the argument to std::forward.</td>
<td>2 - Small differences</td>
<td>A18-9-4</td>
</tr>
<tr>
<td>Section</td>
<td>Rule</td>
<td>Difference Level</td>
<td>Reason</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td>------------------</td>
<td>--------</td>
</tr>
<tr>
<td>17.3.4</td>
<td>Do not create smart pointers of array type.</td>
<td>3 - Significant differences</td>
<td>This especially concerns <code>std::shared_ptr</code>, because <code>std::unique_ptr</code> provides partial specialization for array types.</td>
</tr>
<tr>
<td>17.3.5</td>
<td>Do not create an rvalue reference of <code>std::array</code>.</td>
<td>4 - Rejected</td>
<td>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.</td>
</tr>
<tr>
<td>17.4.1</td>
<td>Use <code>const</code> container calls when result is immediately converted to a <code>const</code> iterator.</td>
<td>2 - Small differences</td>
<td>A23-0-1</td>
</tr>
<tr>
<td>17.4.2</td>
<td>Use API calls that construct objects in place.</td>
<td>3 - Significant differences</td>
<td>A18-5-2</td>
</tr>
<tr>
<td>17.5.1</td>
<td>Do not ignore the result of <code>std::remove</code>, <code>std::remove_if</code> or <code>std::unique</code>.</td>
<td>2 - Small differences</td>
<td>A0-1-2</td>
</tr>
<tr>
<td>18.1.1</td>
<td>Do not use platform specific multi-threading facilities.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>18.2.1</td>
<td>Use <code>high_integrity::thread</code> in place of <code>std::thread</code>.</td>
<td>4 - Rejected</td>
<td>The <code>high_integrity::thread</code> is not part of the C++ Language Standard.</td>
</tr>
<tr>
<td>Section</td>
<td>Guidelines</td>
<td>Analysis</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------------------------------------------------</td>
<td>-------------------</td>
<td></td>
</tr>
<tr>
<td>18.2.2</td>
<td>Synchronize access to data shared between threads using a single lock.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>18.2.3</td>
<td>Do not share volatile data between threads.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>18.2.4</td>
<td>Use <code>std::call_once</code> rather than the Double-Checked Locking pattern.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>18.3.1</td>
<td>Within the scope of a lock, ensure that no static path results in a lock of the same mutex.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>18.3.2</td>
<td>Ensure that order of nesting of locks in a project forms a DAG.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>18.3.3</td>
<td>Do not use <code>std::recursive_mutex</code>.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>18.3.4</td>
<td>Only use <code>std::unique_lock</code> when <code>std::lock_guard</code> cannot be used.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>18.3.5</td>
<td>Do not access the members of <code>std::mutex</code> directly.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>18.3.6</td>
<td>Do not use relaxed atomics.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency” chapter is not yet covered, this will be addressed in future.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>JSF Rule:</th>
<th>Relation type:</th>
<th>Related rule:</th>
<th>Comment:</th>
</tr>
</thead>
<tbody>
<tr>
<td>AV Rule 8 All code shall conform to ISO/IEC 14882:2002(E) standard C++.</td>
<td>2 - Small differences</td>
<td>A1-1-1</td>
<td></td>
</tr>
<tr>
<td>AV Rule 9 Only those characters specified in the C++ basic source character set will be used. [...]</td>
<td>2 - Small differences</td>
<td>A2-2-1</td>
<td></td>
</tr>
<tr>
<td>AV Rule 10 Values of character types will be restricted to a defined and documented subset of ISO 10646-1.</td>
<td>5 - Not yet analyzed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AV Rule 11 Trigraphs will not be used.</td>
<td>2 - Small differences</td>
<td>A2-5-1</td>
<td></td>
</tr>
<tr>
<td>AV Rule 12 The following digraphs will not be used [...].</td>
<td>2 - Small differences</td>
<td>A2-6-1</td>
<td></td>
</tr>
<tr>
<td>AV Rule 13 Multi-byte characters and wide string literals will not be used.</td>
<td>4 - Rejected</td>
<td></td>
<td>Agreed for wchar_t type only, A2-14-3.</td>
</tr>
<tr>
<td>AV Rule 14 Literal suffixes shall use uppercase rather than lowercase letters.</td>
<td>2 - Small differences</td>
<td>M2-13-4</td>
<td></td>
</tr>
<tr>
<td>AV Rule 15 Provision shall be made for run-time checking (defensive programming).</td>
<td>2 - Small differences</td>
<td>M0-3-1</td>
<td></td>
</tr>
<tr>
<td>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.</td>
<td>4 - Rejected</td>
<td></td>
<td>JSF-specific rule.</td>
</tr>
<tr>
<td>AV Rule 17 The error indicator errno shall not be used.</td>
<td>2 - Small differences</td>
<td>M19-3-1</td>
<td></td>
</tr>
<tr>
<td>AV Rule 18 The macro offsetof, in library &lt;stddef.h&gt;, shall not be used.</td>
<td>2 - Small differences</td>
<td>M18-2-1</td>
<td></td>
</tr>
<tr>
<td>AV Rule 19 &lt;locale.h&gt; and the setlocale function shall not be used.</td>
<td>2 - Small differences</td>
<td>A18-0-3</td>
<td></td>
</tr>
<tr>
<td>AV Rule 20 The setjmp macro and the longjmp function shall not be used.</td>
<td>2 - Small differences</td>
<td>M17-0-5</td>
<td></td>
</tr>
<tr>
<td>AV Rule</td>
<td>Description</td>
<td>Differences</td>
<td>Code</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------------------------------------------------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>21</td>
<td>The signal handling facilities of <code>&lt;signal.h&gt;</code> shall not be used.</td>
<td>2 - Small differences</td>
<td>M18-7-1</td>
</tr>
<tr>
<td>22</td>
<td>The input/output library <code>&lt;stdio.h&gt;</code> shall not be used.</td>
<td>2 - Small differences</td>
<td>M27-0-1</td>
</tr>
<tr>
<td>23</td>
<td>The library functions <code>atof</code>, <code>atoi</code> and <code>atol</code> from library <code>&lt;stdlib.h&gt;</code> shall not be used.</td>
<td>2 - Small differences</td>
<td>A18-0-2</td>
</tr>
<tr>
<td>24</td>
<td>The library functions <code>abort</code>, <code>exit</code>, <code>getenv</code> and <code>system</code> from library <code>&lt;stdlib.h&gt;</code> shall not be used.</td>
<td>2 - Small differences</td>
<td>M18-0-3</td>
</tr>
<tr>
<td>25</td>
<td>The time handling functions of library <code>&lt;time.h&gt;</code> shall not be used.</td>
<td>2 - Small differences</td>
<td>M18-0-4</td>
</tr>
<tr>
<td>26</td>
<td>Only the following preprocessor directives shall be used: 1. <code>#ifndef</code> 2. <code>#define</code> 3. <code>#endif</code> 4. <code>#include</code>.</td>
<td>2 - Small differences</td>
<td>A16-0-1</td>
</tr>
<tr>
<td>27</td>
<td><code>#ifndef</code>, <code>#define</code> and <code>#endif</code> 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.</td>
<td>2 - Small differences</td>
<td>A16-0-1, M16-2-3</td>
</tr>
<tr>
<td>28</td>
<td>The <code>#ifndef</code> and <code>#endif</code> preprocessor directives will only be used as defined in AV Rule 27 to prevent multiple inclusions of the same header file.</td>
<td>2 - Small differences</td>
<td>A16-0-1</td>
</tr>
<tr>
<td>29</td>
<td>The <code>#define</code> preprocessor directive shall not be used to create inline macros. Inline functions shall be used instead.</td>
<td>2 - Small differences</td>
<td>A16-0-1</td>
</tr>
<tr>
<td>30</td>
<td>The <code>#define</code> preprocessor directive shall not be used to define constant values. Instead, the <code>const</code> qualifier shall be applied to variable declarations to specify constant values.</td>
<td>2 - Small differences</td>
<td>A16-0-1</td>
</tr>
<tr>
<td>31</td>
<td>The <code>#define</code> preprocessor directive will only be used as part of the technique to prevent multiple inclusions of the same header file.</td>
<td>2 - Small differences</td>
<td>A16-0-1</td>
</tr>
<tr>
<td>32</td>
<td>The <code>#include</code> preprocessor directive will only be used to include header (<code>*.h</code>) files.</td>
<td>2 - Small differences</td>
<td>A16-0-1</td>
</tr>
<tr>
<td>33</td>
<td>The <code>#include</code> directive shall use the <code>&lt;filename.h&gt;</code> notation to include header files.</td>
<td>4 - Rejected</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>Header files should contain logically related declarations only.</td>
<td>2 - Small differences</td>
<td>A3-3-1</td>
</tr>
<tr>
<td>AV Rule</td>
<td>Description</td>
<td>Status</td>
<td>Notes</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>AV 35</td>
<td>A header file will contain a mechanism that prevents multiple inclusions of itself.</td>
<td>2 - Small differences</td>
<td>M16-2-3</td>
</tr>
<tr>
<td>AV 36</td>
<td>Compilation dependencies should be minimized when possible.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>AV 37</td>
<td>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.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>AV 38</td>
<td>Declarations of classes that are only accessed via pointers (*) or references (&amp;) should be supplied by forward headers that contain only forward declarations.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>AV 39</td>
<td>Header files (*.h) will not contain non-const variable definitions or function definitions.</td>
<td>2 - Small differences</td>
<td>M3-2-4, A3-3-1</td>
</tr>
<tr>
<td>AV 40</td>
<td>Every implementation file shall include the header files that uniquely define the inline functions, types, and templates used.</td>
<td>2 - Small differences</td>
<td>M3-2-4, A3-3-1</td>
</tr>
<tr>
<td>AV 41</td>
<td>Source lines will be kept to a length of 120 characters or less.</td>
<td>4 - Rejected</td>
<td>Coding style is not covered by AUTOSAR C++ Coding Guidelines.</td>
</tr>
<tr>
<td>AV 42</td>
<td>Each expression-statement will be on a separate line.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>AV 43</td>
<td>Tabs should be avoided.</td>
<td>4 - Rejected</td>
<td>Coding style is not covered by AUTOSAR C++ Coding Guidelines.</td>
</tr>
<tr>
<td>AV 44</td>
<td>All indentations will be at least two spaces and be consistent within the same source file.</td>
<td>4 - Rejected</td>
<td>Coding style is not covered by AUTOSAR C++ Coding Guidelines.</td>
</tr>
<tr>
<td>AV 45</td>
<td>All words in an identifier will be separated by the &quot;_&quot; character.</td>
<td>4 - Rejected</td>
<td>Coding style is not covered by AUTOSAR C++ Coding Guidelines.</td>
</tr>
<tr>
<td>AV 46</td>
<td>User-specified identifiers (internal and external) will not rely on significance of more than 64 characters.</td>
<td>4 - Rejected</td>
<td>Coding style is not covered by AUTOSAR C++ Coding Guidelines.</td>
</tr>
<tr>
<td>AV 47</td>
<td>Identifiers will not begin with the underscore character &quot;_&quot;.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
</tbody>
</table>

Coding style is not covered by AUTOSAR C++ Coding Guidelines.
<table>
<thead>
<tr>
<th>AV Rule</th>
<th>Description</th>
<th>Rejection Code</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>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 “i”, (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”.</td>
<td>4 - Rejected</td>
<td>Coding style is not covered by AUTOSAR C++ Coding Guidelines.</td>
</tr>
<tr>
<td>49</td>
<td>All acronyms in an identifier will be composed of uppercase letters.</td>
<td>4 - Rejected</td>
<td>Coding style is not covered by AUTOSAR C++ Coding Guidelines.</td>
</tr>
<tr>
<td>50</td>
<td>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.</td>
<td>4 - Rejected</td>
<td>Coding style is not covered by AUTOSAR C++ Coding Guidelines.</td>
</tr>
<tr>
<td>51</td>
<td>All letters contained in function and variable names will be composed entirely of lowercase letters.</td>
<td>4 - Rejected</td>
<td>Coding style is not covered by AUTOSAR C++ Coding Guidelines.</td>
</tr>
<tr>
<td>52</td>
<td>Identifiers for constant and enumerator values shall be lowercase.</td>
<td>4 - Rejected</td>
<td>Coding style is not covered by AUTOSAR C++ Coding Guidelines.</td>
</tr>
<tr>
<td>53</td>
<td>Header files will always have a file name extension of “.h”.</td>
<td>3 - Significant differences A3-1-2</td>
<td></td>
</tr>
<tr>
<td>53.1</td>
<td>The following character sequences shall not appear in header file names: <code>.</code>, <code>\</code>, <code>/</code>, <code>//</code>, or <code>. </code>.</td>
<td>2 - Small differences A16-2-1</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>Implementation files will always have a file name extension of “.cpp”.</td>
<td>2 - Small differences A3-1-3</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>The name of a header file should reflect the logical entity for which it provides declarations.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>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.)</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>AV Rule</td>
<td>Description</td>
<td>Rejected</td>
<td>Notes</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>----------</td>
<td>-------</td>
</tr>
<tr>
<td>AV 57</td>
<td>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).</td>
<td>4 - Rejected</td>
<td>Coding style is not covered by AUTOSAR C++ Coding Guidelines.</td>
</tr>
<tr>
<td>AV 58</td>
<td>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).</td>
<td>4 - Rejected</td>
<td>Coding style is not covered by AUTOSAR C++ Coding Guidelines.</td>
</tr>
<tr>
<td>AV 59</td>
<td>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.</td>
<td>2 - Small differences</td>
<td>M6-3-1</td>
</tr>
<tr>
<td>AV 60</td>
<td>Braces (&quot;{&quot; ) which enclose a block will be placed in the same column, on separate lines directly before and after the block.</td>
<td>4 - Rejected</td>
<td>Coding style is not covered by AUTOSAR C++ Coding Guidelines.</td>
</tr>
<tr>
<td>AV 61</td>
<td>Braces (&quot;{&quot; ) which enclose a block will have nothing else on the line except comments (if necessary).</td>
<td>4 - Rejected</td>
<td>Coding style is not covered by AUTOSAR C++ Coding Guidelines.</td>
</tr>
<tr>
<td>AV 62</td>
<td>The dereference operator &quot;*&quot; and the address-of operator &quot;&amp;&quot; will be directly connected with the type-specifier.</td>
<td>4 - Rejected</td>
<td>Coding style is not covered by AUTOSAR C++ Coding Guidelines.</td>
</tr>
<tr>
<td>AV 63</td>
<td>Spaces will not be used around &quot;.&quot; or &quot;-&gt;&quot;, nor between unary operators and operands.</td>
<td>4 - Rejected</td>
<td>Coding style is not covered by AUTOSAR C++ Coding Guidelines.</td>
</tr>
<tr>
<td>AV 64</td>
<td>A class interface should be complete and minimal.</td>
<td>4 - Rejected</td>
<td>Code metrics are not covered by AUTOSAR C++ Coding Guidelines.</td>
</tr>
<tr>
<td>AV 65</td>
<td>A structure should be used to model an entity that does not require an invariant.</td>
<td>3 - Significant differences</td>
<td>A11-0-2</td>
</tr>
<tr>
<td>AV 66</td>
<td>A class should be used to model an entity that maintains an invariant.</td>
<td>3 - Significant differences</td>
<td>A11-0-1</td>
</tr>
<tr>
<td>AV 67</td>
<td>Public and protected data should only be used in structs - not classes.</td>
<td>2 - Small differences</td>
<td>M11-0-1</td>
</tr>
<tr>
<td>AV Rule</td>
<td>Description</td>
<td>Analysis</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>----------</td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>Unneeded implicitly generated member functions shall be explicitly disallowed.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>A member function that does not affect the state of an object (its instance variables) will be declared const.</td>
<td>2 - Small differences M9-3-3</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>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.</td>
<td>4 - Rejected Friend declarations are prohibited, see: A11-3-1.</td>
<td></td>
</tr>
<tr>
<td>70.1</td>
<td>An object shall not be improperly used before its lifetime begins or after its lifetime ends.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>Calls to an externally visible operation of an object, other than its constructors, shall not be allowed until the object has been fully initialized.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>71.1</td>
<td>A class's virtual functions shall not be invoked from its destructor or any of its constructors.</td>
<td>2 - Small differences M12-1-1</td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>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.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>73</td>
<td>Unnecessary default constructors shall not be defined.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>Initialization of nonstatic class members will be performed through the member initialization list rather than through assignment in the body of a constructor.</td>
<td>2 - Small differences A12-6-1</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>Members of the initialization list shall be listed in the order in which they are declared in the class.</td>
<td>2 - Small differences A8-5-1</td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>A copy constructor and an assignment operator shall be declared for classes that contain pointers to data items or nontrivial destructors.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>77</td>
<td>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).</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>Rule</td>
<td>Description</td>
<td>Annotations</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>AV Rule 77.1</td>
<td>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).</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>AV Rule 78</td>
<td>All base classes with a virtual function shall define a virtual destructor.</td>
<td>2 - Small differences</td>
<td></td>
</tr>
<tr>
<td>AV Rule 79</td>
<td>All resources acquired by a class shall be released by the class's destructor.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>AV Rule 80</td>
<td>The default copy and assignment operators will be used for classes when those operators offer reasonable semantics.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>AV Rule 81</td>
<td>The assignment operator shall handle self-assignment correctly.</td>
<td>2 - Small differences</td>
<td></td>
</tr>
<tr>
<td>AV Rule 82</td>
<td>An assignment operator shall return a reference to &quot;this.&quot;</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>AV Rule 83</td>
<td>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).</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>AV Rule 84</td>
<td>Operator overloading will be used sparingly and in a conventional manner.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>AV Rule 85</td>
<td>When two operators are opposites (such as <code>==</code> and <code>!=</code>), both will be defined and one will be defined in terms of the other.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>AV Rule 86</td>
<td>Concrete types should be used to represent simple independent concepts.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>AV Rule 87</td>
<td>Hierarchies should be based on abstract classes.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>AV Rule 88</td>
<td>Multiple inheritance shall only be allowed in the following restricted form: n interfaces plus m private implementations, plus at most one protected implementation.</td>
<td>3 - Significant differences</td>
<td></td>
</tr>
<tr>
<td>AV Rule 88.1</td>
<td>A stateful virtual base shall be explicitly declared in each derived class that accesses it.</td>
<td>4 - Rejected</td>
<td></td>
</tr>
<tr>
<td>AV Rule 89</td>
<td>A base class shall not be both virtual and non-virtual in the same hierarchy.</td>
<td>2 - Small differences</td>
<td></td>
</tr>
<tr>
<td>AV Rule 90</td>
<td>Heavily used interfaces should be minimal, general and abstract.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>AV Rule 91</td>
<td>Public inheritance will be used to implement &quot;is-a&quot; relationships.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
</tbody>
</table>
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.

AV Rule 93 “has-a” or “is-implemented-in-terms-of” relationships will be modeled through membership or non-public inheritance.

AV Rule 94 An inherited nonvirtual function shall not be redefined in a derived class.

AV Rule 95 An inherited default parameter shall never be redefined.

AV Rule 97 Arrays shall not be used in interfaces. Instead, the Array class should be used.

AV Rule 97.1 Neither operand of an equality operator (== or !=) shall be a pointer to a virtual member function.

AV Rule 98 Every nonlocal name, except main(), should be placed in some namespace.

AV Rule 99 Namespaces will not be nested more than two levels deep.

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.

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.
<table>
<thead>
<tr>
<th>AV Rule</th>
<th>Description</th>
<th>Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>AV Rule 102</td>
<td>Template tests shall be created to cover all actual template instantiations.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>AV Rule 103</td>
<td>Constraint checks should be applied to template arguments.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>AV Rule 104</td>
<td>A template specialization shall be declared before its use.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>AV Rule 105</td>
<td>A template definition's dependence on its instantiation contexts should be minimized.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>AV Rule 106</td>
<td>Specializations for pointer types should be made where appropriate.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>AV Rule 107</td>
<td>Functions shall always be declared at file scope.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>AV Rule 108</td>
<td>Functions with variable numbers of arguments shall not be used.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>AV Rule 109</td>
<td>A function definition should not be placed in a class specification unless the function is intended to be inlined.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>AV Rule 110</td>
<td>Functions with more than 7 arguments will not be used.</td>
<td>4 - Rejected</td>
</tr>
<tr>
<td>AV Rule 111</td>
<td>A function shall not return a pointer or reference to a non-static local object.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>AV Rule 112</td>
<td>Function return values should not obscure resource ownership.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>AV Rule 113</td>
<td>Functions will have a single exit point.</td>
<td>4 - Rejected</td>
</tr>
<tr>
<td>AV Rule 114</td>
<td>All exit points of value-returning functions shall be through return statements.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>AV Rule 115</td>
<td>If a function returns error information, then that error information will be tested.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>AV Rule 116</td>
<td>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.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>AV Rule 117</td>
<td>Arguments should be passed by reference if NULL values are not possible.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>AV Rule 117.1</td>
<td>An object should be passed as <code>const T&amp;</code> if the function should not change the value of the object.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>AV Rule 117.2</td>
<td>An object should be passed as <code>T&amp;</code> if the function may change the value of the object.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>AV Rule 118</td>
<td>Arguments should be passed via pointers if NULL values are possible.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>AV Rule 118.1</td>
<td>An object should be passed as <code>const T*</code> if its value should not be modified.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>AV Rule 118.2</td>
<td>An object should be passed as <code>T*</code> if its value may be modified.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>AV Rule 119</td>
<td>Functions shall not call themselves, either directly or indirectly (i.e. recursion shall not be allowed).</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>AV Rule 120</td>
<td>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.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>AV Rule 121</td>
<td>Only functions with 1 or 2 statements should be considered candidates for inline functions.</td>
<td>4 - Rejected</td>
</tr>
<tr>
<td>AV Rule 122</td>
<td>Trivial accessor and mutator functions should be inlined.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>AV Rule 123</td>
<td>The number of accessor and mutator functions should be minimized.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>AV Rule 124</td>
<td>Trivial forwarding functions should be inlined.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>AV Rule 125</td>
<td>Unnecessary temporary objects should be avoided.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>AV Rule 126</td>
<td>Only valid C++ style comments (<code>//</code>) shall be used.</td>
<td>4 - Rejected</td>
</tr>
<tr>
<td>AV Rule 127</td>
<td>Code that is not used (commented out) shall be deleted.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>AV Rule 128</td>
<td>Comments that document actions or sources (e.g. tables, figures, paragraphs, etc.) outside of the file being documented will not be allowed.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>AV Rule 129</td>
<td>Comments in header files should describe the externally visible behavior of the functions or classes being documented.</td>
<td>3 - Significant differences</td>
</tr>
<tr>
<td>AV Rule</td>
<td>Description</td>
<td>Rejected/Accepted/Not yet analyzed</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>AV Rule 130</td>
<td>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.</td>
<td>4 - Rejected</td>
</tr>
<tr>
<td>AV Rule 131</td>
<td>One should avoid stating in comments what is better stated in code (i.e. do not simply repeat what is in the code).</td>
<td>4 - Rejected</td>
</tr>
<tr>
<td>AV Rule 132</td>
<td>Each variable declaration, typedef, enumeration value, and structure member will be commented.</td>
<td>4 - Rejected</td>
</tr>
<tr>
<td>AV Rule 133</td>
<td>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).</td>
<td>4 - Rejected</td>
</tr>
<tr>
<td>AV Rule 134</td>
<td>Assumptions (limitations) made by functions should be documented in the function's preamble.</td>
<td>3 - Significant differences</td>
</tr>
<tr>
<td>AV Rule 135</td>
<td>Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>AV Rule 136</td>
<td>Declarations should be at the smallest feasible scope.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>AV Rule 137</td>
<td>All declarations at file scope should be static where possible.</td>
<td>3 - Significant differences</td>
</tr>
<tr>
<td>AV Rule 138</td>
<td>Identifiers shall not simultaneously have both internal and external linkage in the same translation unit.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>AV Rule 139</td>
<td>External objects will not be declared in more than one file.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>AV Rule 140</td>
<td>The register storage class specifier shall not be used.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>AV Rule 141</td>
<td>A class, structure, or enumeration will not be declared in the definition of its type.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>AV Rule 142</td>
<td>All variables shall be initialized before use.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>AV Rule 143</td>
<td>Variables will not be introduced until they can be initialized with meaningful values.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>AV Rule 144</td>
<td>Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>AV Rule</td>
<td>Description</td>
<td>Analysis</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>----------</td>
</tr>
<tr>
<td>Rule 145</td>
<td>In an enumerator list, the “=” construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>Rule 146</td>
<td>Floating point implementations shall comply with a defined floating point standard. The standard that will be used is the ANSI/IEEE Std 754</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>Rule 147</td>
<td>The underlying bit representations of floating point numbers shall not be used in any way by the programmer.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>Rule 148</td>
<td>Enumeration types shall be used instead of integer types (and constants) to select from a limited series of choices.</td>
<td></td>
</tr>
<tr>
<td>Rule 149</td>
<td>Octal constants (other than zero) shall not be used.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>Rule 150</td>
<td>Hexadecimal constants will be represented using all uppercase letters.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>Rule 151</td>
<td>Numeric values in code will not be used; symbolic values will be used instead.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>Rule 151.1</td>
<td>A string literal shall not be modified.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>Rule 152</td>
<td>Multiple variable declarations shall not be allowed on the same line.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>Rule 153</td>
<td>Unions shall not be used.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>Rule 154</td>
<td>Bit-fields shall have explicitly unsigned integral or enumeration types only.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>Rule 155</td>
<td>Bit-fields will not be used to pack data into a word for the sole purpose of saving space.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>Rule 156</td>
<td>All the members of a structure (or class) shall be named and shall only be accessed via their names.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>Rule 157</td>
<td>The right hand operand of a &amp;&amp; or</td>
<td></td>
</tr>
<tr>
<td>Rule 158</td>
<td>The operands of a logical &amp;&amp; or</td>
<td></td>
</tr>
<tr>
<td>Rule 159</td>
<td>Operators</td>
<td></td>
</tr>
<tr>
<td>Rule 160</td>
<td>An assignment expression shall be used only as the expression in an expression statement.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>AV Rule</td>
<td>Description</td>
<td>Score</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>-------</td>
</tr>
<tr>
<td>162</td>
<td>Signed and unsigned values shall not be mixed in arithmetic or comparison operations.</td>
<td>5</td>
</tr>
<tr>
<td>163</td>
<td>Unsigned arithmetic shall not be used.</td>
<td>5</td>
</tr>
<tr>
<td>164</td>
<td>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).</td>
<td>2</td>
</tr>
<tr>
<td>164.1</td>
<td>The left-hand operand of a right-shift operator shall not have a negative value.</td>
<td>5</td>
</tr>
<tr>
<td>165</td>
<td>The unary minus operator shall not be applied to an unsigned expression.</td>
<td>2</td>
</tr>
<tr>
<td>166</td>
<td>The sizeof operator will not be used on expressions that contain side effects.</td>
<td>2</td>
</tr>
<tr>
<td>167</td>
<td>The implementation of integer division in the chosen compiler shall be determined, documented and taken into account.</td>
<td>3</td>
</tr>
<tr>
<td>168</td>
<td>The comma operator shall not be used.</td>
<td>2</td>
</tr>
<tr>
<td>169</td>
<td>Pointers to pointers should be avoided when possible.</td>
<td>3</td>
</tr>
<tr>
<td>170</td>
<td>More than 2 levels of pointer indirection shall not be used.</td>
<td>3</td>
</tr>
<tr>
<td>171</td>
<td>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).</td>
<td>2</td>
</tr>
<tr>
<td>173</td>
<td>The address of an object with automatic storage shall not be assigned to an object which persists after the object has ceased to exist.</td>
<td>2</td>
</tr>
<tr>
<td>174</td>
<td>The null pointer shall not be de-referenced.</td>
<td>5</td>
</tr>
<tr>
<td>175</td>
<td>A pointer shall not be compared to NULL or be assigned NULL; use plain 0 instead.</td>
<td>4</td>
</tr>
<tr>
<td>176</td>
<td>A typedef will be used to simplify program syntax when declaring function pointers.</td>
<td>5</td>
</tr>
<tr>
<td>177</td>
<td>User-defined conversion functions should be avoided.</td>
<td>5</td>
</tr>
</tbody>
</table>
### 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

### 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

### AV Rule 186
There shall be no unreachable code.

2 - Small differences

### AV Rule 187
All non-null statements shall potentially have a side-effect.

2 - Small differences

### AV Rule 188
Labels will not be used, except in switch statements.

3 - Significant differences

### AV Rule 189
The goto statement shall not be used.

2 - Small differences

### AV Rule 190
The continue statement shall not be used.

4 - Rejected

The continue statement usage is allowed within for-loops, see: M6-6-3.

### AV Rule 191
The break statement shall not be used (except to terminate the cases of a switch statement).

2 - Small differences

### 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
<table>
<thead>
<tr>
<th>AV Rule</th>
<th>Description</th>
<th>Differences</th>
<th>Related</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>193</td>
<td>Every non-empty case clause in a switch statement shall be terminated with a break statement.</td>
<td>2 - Small differences</td>
<td>M6-4-3, M6-4-5</td>
<td></td>
</tr>
<tr>
<td>194</td>
<td>All switch statements that do not intend to test for every enumeration value shall contain a final default clause.</td>
<td>2 - Small differences</td>
<td>M6-4-6</td>
<td></td>
</tr>
<tr>
<td>195</td>
<td>A switch expression will not represent a Boolean value.</td>
<td>2 - Small differences</td>
<td>M6-4-7</td>
<td></td>
</tr>
<tr>
<td>196</td>
<td>Every switch statement will have at least two cases and a potential default.</td>
<td>2 - Small differences</td>
<td>A6-4-1</td>
<td></td>
</tr>
<tr>
<td>197</td>
<td>Floating point variables shall not be used as loop counters.</td>
<td>2 - Small differences</td>
<td>M6-5-2</td>
<td></td>
</tr>
<tr>
<td>198</td>
<td>The initialization expression in a for loop will perform no actions other than to initialize the value of a single for loop parameter.</td>
<td>5 - Not yet analyzed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>199</td>
<td>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.</td>
<td>5 - Not yet analyzed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>Null initialize or increment expressions in for loops will not be used; a while loop will be used instead.</td>
<td>3 - Significant differences</td>
<td>A6-5-2</td>
<td></td>
</tr>
<tr>
<td>201</td>
<td>Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop.</td>
<td>2 - Small differences</td>
<td>M6-5-3</td>
<td></td>
</tr>
<tr>
<td>202</td>
<td>Floating point variables shall not be tested for exact equality or inequality.</td>
<td>2 - Small differences</td>
<td>M6-2-2</td>
<td></td>
</tr>
<tr>
<td>203</td>
<td>Evaluation of expressions shall not lead to overflow/underflow (unless required algorithmically and then should be heavily documented).</td>
<td>2 - Small differences</td>
<td>M5-19-1, A7-1-2</td>
<td></td>
</tr>
<tr>
<td>204</td>
<td>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.</td>
<td>5 - Not yet analyzed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>204.1</td>
<td>The value of an expression shall be the same under any order of evaluation that the standard permits.</td>
<td>2 - Small differences</td>
<td>A5-0-1</td>
<td></td>
</tr>
<tr>
<td>205</td>
<td>The volatile keyword shall not be used unless directly interfacing with hardware.</td>
<td>5 - Not yet analyzed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AV Rule</td>
<td>Description</td>
<td>Compliance</td>
<td>Notes</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>------------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>206</td>
<td>Allocation/deallocation from/to the free store (heap) shall not occur after initialization.</td>
<td>2 - Small differences</td>
<td>A18-5-3</td>
<td></td>
</tr>
<tr>
<td>207</td>
<td>Unencapsulated global data will be avoided.</td>
<td>5 - Not yet analyzed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>208</td>
<td>C++ exceptions shall not be used (i.e. throw, catch and try shall not be used.)</td>
<td>4 - Rejected</td>
<td>C++ exceptions may be used conditionally.</td>
<td></td>
</tr>
<tr>
<td>209</td>
<td>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.</td>
<td>2 - Small differences</td>
<td>A3-9-1</td>
<td></td>
</tr>
<tr>
<td>210</td>
<td>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.)</td>
<td>5 - Not yet analyzed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>210.1</td>
<td>Algorithms shall not make assumptions concerning the order of allocation of nonstatic data members separated by an access specifier.</td>
<td>5 - Not yet analyzed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>211</td>
<td>Algorithms shall not assume that shorts, ints, longs, floats, doubles or long doubles begin at particular addresses.</td>
<td>5 - Not yet analyzed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>212</td>
<td>Underflow or overflow functioning shall not be depended on in any special way.</td>
<td>5 - Not yet analyzed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>213</td>
<td>No dependence shall be placed on C++’s operator precedence rules, below arithmetic operators, in expressions.</td>
<td>2 - Small differences</td>
<td>A5-0-1</td>
<td></td>
</tr>
<tr>
<td>214</td>
<td>Assuming that non-local static objects, in separate translation units, are initialized in a special order shall not be done.</td>
<td>5 - Not yet analyzed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>215</td>
<td>Pointer arithmetic will not be used.</td>
<td>3 - Significant differences</td>
<td>M5-0-15 Pointer arithmetic may be used for array indexing.</td>
<td></td>
</tr>
<tr>
<td>216</td>
<td>Programmers should not attempt to prematurely optimize code.</td>
<td>5 - Not yet analyzed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>217</td>
<td>Compile-time and link-time errors should be preferred over run-time errors.</td>
<td>5 - Not yet analyzed</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### AV Rule 218 Compiler warning levels will be set in compliance with project policies.

- 3 - Significant differences
- A1-1-2

### AV Rule 219 All tests applied to a base class interface shall be applied to all derived class interfaces as well. If the derived class poses stronger postconditions/invariants, then the new postconditions/invariants shall be substituted in the derived class tests.

- 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.

<table>
<thead>
<tr>
<th>SEI CERT Rule:</th>
<th>Relation type:</th>
<th>Related rule:</th>
<th>Comment:</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCL30-C. Declare objects with appropriate storage durations.</td>
<td>2 - Small differences</td>
<td>M7-5-2</td>
<td></td>
</tr>
<tr>
<td>DCL40-C. Do not create incompatible declarations of the same function or object.</td>
<td>2 - Small differences</td>
<td>M3-9-1</td>
<td></td>
</tr>
<tr>
<td>DCL50-CPP. Do not define a C-style variadic function.</td>
<td>2 - Small differences</td>
<td>A8-4-1</td>
<td></td>
</tr>
<tr>
<td>DCL51-CPP. Do not declare or define a reserved identifier.</td>
<td>2 - Small differences</td>
<td>A13-1-2, A17-0-1</td>
<td></td>
</tr>
<tr>
<td>DCL52-CPP. Never qualify a reference type with const or volatile.</td>
<td>5 - Not yet analyzed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DCL53-CPP. Do not write syntactically ambiguous declarations.</td>
<td>2 - Small differences</td>
<td>A8-5-2</td>
<td></td>
</tr>
<tr>
<td>DCL54-CPP. Overload allocation and deallocation functions as a pair in the same scope.</td>
<td>3 - Significant differences</td>
<td>A18-5-3, A18-5-4</td>
<td></td>
</tr>
<tr>
<td>DCL55-CPP. Avoid information leakage when passing a class object across a trust boundary.</td>
<td>5 - Not yet analyzed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Analyzed</td>
<td>Status</td>
</tr>
<tr>
<td>--------</td>
<td>----------------------------------------------------------------------------</td>
<td>----------</td>
<td>--------</td>
</tr>
<tr>
<td>DCL56-CPP</td>
<td>Avoid cycles during initialization of static objects.</td>
<td>5</td>
<td>Not yet analyzed</td>
</tr>
<tr>
<td>DCL57-CPP</td>
<td>Do not let exceptions escape from destructors or deallocation functions.</td>
<td>2</td>
<td>Significant differences</td>
</tr>
<tr>
<td>DCL58-CPP</td>
<td>Do not modify the standard namespaces.</td>
<td>5</td>
<td>Not yet analyzed</td>
</tr>
<tr>
<td>DCL59-CPP</td>
<td>Do not define an unnamed namespace in a header file.</td>
<td>2</td>
<td>Small differences</td>
</tr>
<tr>
<td>DCL60-CPP</td>
<td>Obey the one-definition rule.</td>
<td>2</td>
<td>Small differences</td>
</tr>
<tr>
<td>EXP34-C</td>
<td>Do not dereference null pointers.</td>
<td>5</td>
<td>Not yet analyzed</td>
</tr>
<tr>
<td>EXP35-C</td>
<td>Do not modify objects with temporary lifetime.</td>
<td>5</td>
<td>Not yet analyzed</td>
</tr>
<tr>
<td>EXP36-C</td>
<td>Do not cast pointers into more strictly aligned pointer types.</td>
<td>5</td>
<td>Not yet analyzed</td>
</tr>
<tr>
<td>EXP37-C</td>
<td>Call functions with the correct number and type of arguments.</td>
<td>5</td>
<td>Not yet analyzed</td>
</tr>
<tr>
<td>EXP39-C</td>
<td>Do not access a variable through a pointer of an incompatible type.</td>
<td>5</td>
<td>Not yet analyzed</td>
</tr>
<tr>
<td>EXP42-C</td>
<td>Do not compare padding data.</td>
<td>5</td>
<td>Not yet analyzed</td>
</tr>
<tr>
<td>EXP45-C</td>
<td>Do not perform assignments in selection statements.</td>
<td>2</td>
<td>Small differences</td>
</tr>
<tr>
<td>EXP46-C</td>
<td>Do not use a bitwise operator with a Boolean-like operand.</td>
<td>4</td>
<td>Rejected</td>
</tr>
<tr>
<td>EXP47-C</td>
<td>Do not call va_arg with an argument of the incorrect type.</td>
<td>4</td>
<td>Rejected</td>
</tr>
<tr>
<td>EXP50-CPP</td>
<td>Do not depend on the order of evaluation for side effects.</td>
<td>2</td>
<td>Small differences</td>
</tr>
<tr>
<td>EXP51-CPP</td>
<td>Do not delete an array through a pointer of the incorrect type.</td>
<td>5</td>
<td>Not yet analyzed</td>
</tr>
<tr>
<td>EXP52-CPP</td>
<td>Do not rely on side effects in unevaluated operands.</td>
<td>3</td>
<td>Significant differences</td>
</tr>
<tr>
<td>EXP53-CPP</td>
<td>Do not read uninitialized memory.</td>
<td>5</td>
<td>Not yet analyzed</td>
</tr>
<tr>
<td>EXP54-CPP</td>
<td>Do not access an object outside of its lifetime.</td>
<td>5</td>
<td>Not yet analyzed</td>
</tr>
<tr>
<td>EXP55-CPP</td>
<td>Do not access a cv-qualified object through a cv-unqualified type.</td>
<td>2 - Small differences</td>
<td>A5-2-3</td>
</tr>
<tr>
<td>EXP56-CPP</td>
<td>Do not call a function with a mismatched language linkage.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>EXP57-CPP</td>
<td>Do not cast or delete pointers to incomplete classes.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>EXP58-CPP</td>
<td>Pass an object of the correct type to va_start.</td>
<td>4 - Rejected</td>
<td></td>
</tr>
<tr>
<td>EXP59-CPP</td>
<td>Use offsetof() on valid types and members.</td>
<td>4 - Rejected</td>
<td></td>
</tr>
<tr>
<td>EXP60-CPP</td>
<td>Do not pass a nonstandard-layout type object across execution boundaries.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>EXP61-CPP</td>
<td>A lambda object must not outlive any of its reference captured objects.</td>
<td>2 - Small differences</td>
<td>A5-1-4</td>
</tr>
<tr>
<td>EXP62-CPP</td>
<td>Do not access the bits of an object representation that are not part of the object’s value representation.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>EXP63-CPP</td>
<td>Do not rely on the value of a moved-from object.</td>
<td>2 - Small differences</td>
<td>A12-8-3</td>
</tr>
<tr>
<td>INT30-C</td>
<td>Ensure that unsigned integer operations do not wrap.</td>
<td>2 - Small differences</td>
<td>A4-7-1, M5-19-1</td>
</tr>
<tr>
<td>INT31-C</td>
<td>Ensure that integer conversions do not result in lost or misinterpreted data.</td>
<td>3 - Significant differences</td>
<td>A4-7-1, M5-0-15</td>
</tr>
<tr>
<td>INT32-C</td>
<td>Ensure that operations on signed integers do not result in overflow.</td>
<td>2 - Small differences</td>
<td>A4-7-1</td>
</tr>
<tr>
<td>INT33-C</td>
<td>Ensure that division and remainder operations do not result in divide-by-zero errors.</td>
<td>2 - Small differences</td>
<td>A5-5-1</td>
</tr>
<tr>
<td>INT34-C</td>
<td>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.</td>
<td>2 - Small differences</td>
<td>M5-8-1</td>
</tr>
<tr>
<td>INT35-C</td>
<td>Use correct integer precisions.</td>
<td>3 - Significant differences</td>
<td>A3-9-1</td>
</tr>
<tr>
<td>INT36-C</td>
<td>Converting a pointer to integer or integer to pointer.</td>
<td>2 - Small differences</td>
<td>M5-2-8, M5-2-9</td>
</tr>
<tr>
<td>INT50-CPP</td>
<td>Do not cast to an out-of-range enumeration value.</td>
<td>2 - Small differences</td>
<td>A7-2-1</td>
</tr>
<tr>
<td>CTR50-CPP</td>
<td>Guarantee that container indices and iterators are within the valid range.</td>
<td>3 - Significant differences</td>
<td>A5-2-5</td>
</tr>
<tr>
<td>CTR51-CPP</td>
<td>Use valid references, pointers, and iterators to reference elements of a container.</td>
<td>3 - Significant differences</td>
<td>M5-0-16, M5-0-17</td>
</tr>
<tr>
<td>CTR52-CPP</td>
<td>Guarantee that library functions do not overflow.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>CTR53-CPP</td>
<td>Use valid iterator ranges.</td>
<td>3 - Significant differences</td>
<td>M5-0-16, M5-0-17</td>
</tr>
<tr>
<td>CTR54-CPP</td>
<td>Do not subtract iterators that do not refer to the same container.</td>
<td>3 - Significant differences</td>
<td>M5-0-16, M5-0-17</td>
</tr>
<tr>
<td>CTR55-CPP</td>
<td>Do not use an additive operator on an iterator if the result would overflow.</td>
<td>3 - Significant differences</td>
<td>M5-0-16, M5-0-17</td>
</tr>
<tr>
<td>CTR56-CPP</td>
<td>Do not use pointer arithmetic on polymorphic objects.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>CTR57-CPP</td>
<td>Provide a valid ordering predicate.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>CTR58-CPP</td>
<td>Predicate function objects should not be mutable.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>ARR30-C</td>
<td>Do not form or use out-of-bounds pointers or array subscripts.</td>
<td>3 - Significant differences</td>
<td>A5-2-5</td>
</tr>
<tr>
<td>ARR37-C</td>
<td>Do not add or subtract an integer to a pointer to a non-array object.</td>
<td>3 - Significant differences</td>
<td>M5-0-15</td>
</tr>
<tr>
<td>ARR38-C</td>
<td>Guarantee that library functions do not form invalid pointers.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>ARR39-C</td>
<td>Do not add or subtract a scaled integer to a pointer.</td>
<td>3 - Significant differences</td>
<td>M5-0-15</td>
</tr>
<tr>
<td>STR30-C</td>
<td>Do not attempt to modify string literals.</td>
<td>4 - Rejected</td>
<td>Use of C-style arrays, apart from static constexpr members, is prohibited. See: A18-1-1.</td>
</tr>
<tr>
<td>STR32-C</td>
<td>Do not pass a non-null-terminated character sequence to a library function that expects a string</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>STR34-C</td>
<td>Cast characters to unsigned char before converting to larger integer sizes.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>STR37-C</td>
<td>Arguments to character-handling functions must be representable as an unsigned char.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>STR38-C</td>
<td>Do not confuse narrow and wide character strings and functions.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>STR51-CPP</td>
<td>Do not attempt to create a std::string from a null pointer.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>STR52-CPP</td>
<td>Use valid references, pointers, and iterators to reference elements of a basic_string.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>Guideline</td>
<td>Category</td>
<td>Description</td>
<td>Recommendation</td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
<td>-------------</td>
<td>----------------</td>
</tr>
<tr>
<td>STR53-CPP</td>
<td>3 - Significant differences</td>
<td>Range check element access.</td>
<td>A5-2-5</td>
</tr>
<tr>
<td>MEM31-C</td>
<td>5 - Not yet analyzed</td>
<td>Free dynamically allocated memory when no longer needed.</td>
<td></td>
</tr>
<tr>
<td>MEM34-C</td>
<td>5 - Not yet analyzed</td>
<td>Only free memory allocated dynamically.</td>
<td></td>
</tr>
<tr>
<td>MEM35-C</td>
<td>4 - Rejected</td>
<td>Allocate sufficient memory for an object.</td>
<td>Use of malloc, calloc and realloc functions is prohibited, see: A18-5-1.</td>
</tr>
<tr>
<td>MEM36-C</td>
<td>4 - Rejected</td>
<td>Do not modify the alignment of objects by calling realloc().</td>
<td>Use of malloc, calloc and realloc functions is prohibited, see: A18-5-1.</td>
</tr>
<tr>
<td>MEM50-CPP</td>
<td>5 - Not yet analyzed</td>
<td>Do not access freed memory.</td>
<td></td>
</tr>
<tr>
<td>MEM51-CPP</td>
<td>3 - Significant differences</td>
<td>Properly deallocate dynamically allocated resources.</td>
<td>A18-5-3</td>
</tr>
<tr>
<td>MEM52-CPP</td>
<td>3 - Significant differences</td>
<td>Detect and handle memory allocation errors.</td>
<td>A15-0-2, A15-2-2, A15-3-3, A15-5-3</td>
</tr>
<tr>
<td>MEM53-CPP</td>
<td>4 - Rejected</td>
<td>Explicitly construct and destruct objects when manually managing object lifetime.</td>
<td>A18-5-2</td>
</tr>
<tr>
<td>MEM54-CPP</td>
<td>5 - Not yet analyzed</td>
<td>Provide placement new with properly aligned pointers to sufficient storage capacity.</td>
<td></td>
</tr>
<tr>
<td>MEM55-CPP</td>
<td>5 - Not yet analyzed</td>
<td>Honor replacement dynamic storage management requirements.</td>
<td></td>
</tr>
<tr>
<td>MEM56-CPP</td>
<td>5 - Not yet analyzed</td>
<td>Do not store an already-owned pointer value in an unrelated smart pointer.</td>
<td></td>
</tr>
<tr>
<td>MEM57-CPP</td>
<td>5 - Not yet analyzed</td>
<td>Avoid using default operator new for over-aligned types.</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Level</td>
<td>Notes</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
<td>-------</td>
<td>---------------------</td>
</tr>
<tr>
<td>FIO50-CPP</td>
<td>Do not alternately input and output from a file stream without an intervening positioning call.</td>
<td>5</td>
<td>Not yet analyzed</td>
</tr>
<tr>
<td>FIO51-CPP</td>
<td>Close files when they are no longer needed.</td>
<td>5</td>
<td>Not yet analyzed</td>
</tr>
<tr>
<td>FIO30-C</td>
<td>Exclude user input from format strings.</td>
<td>2</td>
<td>Small differences A27-0-1</td>
</tr>
<tr>
<td>FIO32-C</td>
<td>Do not perform operations on devices that are only appropriate for files.</td>
<td>5</td>
<td>Not yet analyzed</td>
</tr>
<tr>
<td>FIO34-C</td>
<td>Distinguish between characters read from a file and EOF or WEOF.</td>
<td>5</td>
<td>Not yet analyzed</td>
</tr>
<tr>
<td>FIO37-C</td>
<td>Do not assume that fgets() or fgetws() returns a nonempty string when successful.</td>
<td>5</td>
<td>Not yet analyzed</td>
</tr>
<tr>
<td>FIO38-C</td>
<td>Do not copy a FILE object.</td>
<td>5</td>
<td>Not yet analyzed</td>
</tr>
<tr>
<td>FIO39-C</td>
<td>Do not alternately input and output from a stream without an intervening flush or positioning call.</td>
<td>5</td>
<td>Not yet analyzed</td>
</tr>
<tr>
<td>FIO40-C</td>
<td>Reset strings on fgets() or fgetws() failure.</td>
<td>5</td>
<td>Not yet analyzed</td>
</tr>
<tr>
<td>FIO41-C</td>
<td>Do not call getc(), putc(), getwc(), or putwc() with a stream argument that has side effects.</td>
<td>5</td>
<td>Not yet analyzed</td>
</tr>
<tr>
<td>FIO42-C</td>
<td>Close files when they are no longer needed.</td>
<td>5</td>
<td>Not yet analyzed</td>
</tr>
<tr>
<td>FIO44-C</td>
<td>Only use values for fsetpos() that are returned from fgetpos().</td>
<td>5</td>
<td>Not yet analyzed</td>
</tr>
<tr>
<td>FIO45-C</td>
<td>Avoid TOCTOU race conditions while accessing files.</td>
<td>5</td>
<td>Not yet analyzed</td>
</tr>
<tr>
<td>FIO46-C</td>
<td>Do not access a closed file.</td>
<td>5</td>
<td>Not yet analyzed</td>
</tr>
<tr>
<td>FIO47-C</td>
<td>Use valid format strings.</td>
<td>5</td>
<td>Not yet analyzed</td>
</tr>
<tr>
<td>ERR30-C</td>
<td>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.</td>
<td>4</td>
<td>Rejected Use of the errno is prohibited, see: M19-3-1.</td>
</tr>
<tr>
<td>ERR32-C</td>
<td>Do not rely on indeterminate values of errno.</td>
<td>4</td>
<td>Rejected Use of the errno is prohibited, see: M19-3-1.</td>
</tr>
<tr>
<td>ERR33-C</td>
<td>Detect and handle standard library errors.</td>
<td>3</td>
<td>Small differences M0-3-2, A15-0-3</td>
</tr>
<tr>
<td>ERR50-CPP</td>
<td>Do not abruptly terminate the program.</td>
<td>2</td>
<td>Small differences A15-5-2, A15-5-3</td>
</tr>
<tr>
<td>ERR51-CPP</td>
<td>Handle all exceptions.</td>
<td>2</td>
<td>Small differences A15-3-3, A15-5-3</td>
</tr>
<tr>
<td>ERR52-CPP</td>
<td>Do not use setjmp() or longjmp().</td>
<td>2</td>
<td>Small differences M17-0-5</td>
</tr>
<tr>
<td>ID</td>
<td>Text</td>
<td>Category</td>
<td>Number</td>
</tr>
<tr>
<td>-------------</td>
<td>----------------------------------------------------------------------</td>
<td>----------</td>
<td>--------</td>
</tr>
<tr>
<td>ERR53-CPP</td>
<td>Do not reference base classes or class data members in a constructor or destructor function-try-block handler.</td>
<td>3 - Significant differences</td>
<td>M15-3-3</td>
</tr>
<tr>
<td>ERR54-CPP</td>
<td>Catch handlers should order their parameter types from most derived to least derived.</td>
<td>2 - Small differences</td>
<td>M15-3-6, M15-3-7</td>
</tr>
<tr>
<td>ERR55-CPP</td>
<td>Honor exception specifications.</td>
<td>3 - Significant differences</td>
<td>A15-4-2</td>
</tr>
<tr>
<td>ERR56-CPP</td>
<td>Guarantee exception safety.</td>
<td>2 - Small differences</td>
<td>A15-0-2</td>
</tr>
<tr>
<td>ERR57-CPP</td>
<td>Do not leak resources when handling exceptions.</td>
<td>3 - Significant differences</td>
<td>A15-0-2, A15-1-2, A15-1-4</td>
</tr>
<tr>
<td>ERR58-CPP</td>
<td>Handle all exceptions thrown before main() begins executing.</td>
<td>2 - Small differences</td>
<td>A15-2-1</td>
</tr>
<tr>
<td>ERR59-CPP</td>
<td>Do not throw an exception across execution boundaries.</td>
<td>2 - Small differences</td>
<td>A15-1-5</td>
</tr>
<tr>
<td>ERR60-CPP</td>
<td>Exception objects must be nothrow copy constructible.</td>
<td>3 - Significant differences</td>
<td>A15-5-3</td>
</tr>
<tr>
<td>ERR61-CPP</td>
<td>Catch exceptions by lvalue reference.</td>
<td>2 - Small differences</td>
<td>A15-3-5</td>
</tr>
<tr>
<td>ERR62-CPP</td>
<td>Detect errors when converting a string to a number.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OOP50-CPP</td>
<td>Do not invoke virtual functions from constructors or destructors.</td>
<td>2 - Small differences</td>
<td>M12-1-1</td>
</tr>
<tr>
<td>OOP51-CPP</td>
<td>Do not slice derived objects.</td>
<td>2 - Small differences</td>
<td>A12-8-6, A15-3-5</td>
</tr>
<tr>
<td>OOP52-CPP</td>
<td>Do not delete a polymorphic object without a virtual destructor.</td>
<td>2 - Small differences</td>
<td>A12-4-1, A12-4-2</td>
</tr>
<tr>
<td>OOP53-CPP</td>
<td>Write constructor member initializers in the canonical order.</td>
<td>2 - Small differences</td>
<td>A8-5-1</td>
</tr>
<tr>
<td>OOP54-CPP</td>
<td>Gracefully handle self-copy assignment.</td>
<td>2 - Small differences</td>
<td>A12-8-5</td>
</tr>
<tr>
<td>OOP55-CPP</td>
<td>Do not use pointer-to-member operators to access nonexistent members.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OOP56-CPP</td>
<td>Honor replacement handler requirements.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Guideline</td>
<td>Description</td>
<td>Analysis</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>---------</td>
<td></td>
</tr>
<tr>
<td>OOP57-CPP</td>
<td>Prefer special member functions and overloaded operators to C Standard Library functions.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>OOP58-CPP</td>
<td>Copy operations must not mutate the source object.</td>
<td>2 - Small differences</td>
<td>A12-8-1</td>
</tr>
<tr>
<td>CON50-CPP</td>
<td>Do not destroy a mutex while it is locked.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>CON51-CPP</td>
<td>Ensure actively held locks are released on exceptional conditions.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>CON52-CPP</td>
<td>Prevent data races when accessing bit-fields from multiple threads.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>CON53-CPP</td>
<td>Avoid deadlock by locking in a predefined order.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>CON54-CPP</td>
<td>Wrap functions that can spuriously wake up in a loop.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>CON55-CPP</td>
<td>Preserve thread safety and liveness when using condition variables.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>CON56-CPP</td>
<td>Do not speculatively lock a non-recursive mutex that is already owned by the calling thread.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>CON33-C</td>
<td>Avoid race conditions when using library functions.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>CON37-C</td>
<td>Do not call signal() in a multithreaded program</td>
<td>4 - Rejected</td>
<td>Use of signal handling facilities of &lt;signal&gt; is prohibited, see: M18-7-1.</td>
</tr>
<tr>
<td>CON40-C</td>
<td>Do not refer to an atomic variable twice in an expression.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>CON41-C</td>
<td>Wrap functions that can fail spuriously in a loop.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>CON43-C</td>
<td>Do not allow data races in multithreaded code.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>MSC33-C</td>
<td>Do not pass invalid data to the asctime() function.</td>
<td>4 - Rejected</td>
<td>Use of time handling functions of &lt;ctime&gt; is prohibited, see: M18-0-4.</td>
</tr>
<tr>
<td>MSC38-C</td>
<td>Do not treat a predefined identifier as an object if it might only be implemented as a macro.</td>
<td>4 - Rejected</td>
<td>Error indicator errno, setjmp() and variadic arguments shall not be used, see: M19-3-1, M17-0-5, A8-4-1.</td>
</tr>
<tr>
<td>MSC39-C</td>
<td>Do not call va_arg() on a va_list that has an indeterminate value.</td>
<td>4 - Rejected</td>
<td>Use of variadic arguments is prohibited, see: A8-4-1.</td>
</tr>
<tr>
<td>MSC40-C</td>
<td>Do not violate constraints.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>MSC50-CPP</td>
<td>Do not use std::rand() for generating pseudorandom numbers.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>MSC51-CPP</td>
<td>Ensure your random number generator is properly seeded.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>MSC52-CPP</td>
<td>Value-returning functions must return a value from all exit paths.</td>
<td>2 - Small differences</td>
<td>A8-4-2</td>
</tr>
<tr>
<td>MSC53-CPP</td>
<td>Do not return from a function declared [[noreturn]].</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>MSC54-CPP</td>
<td>A signal handler must be a plain old function.</td>
<td>4 - Rejected</td>
<td>Use of signal handling facilities of &lt;csignal&gt; is prohibited, see: M18-7-1.</td>
</tr>
<tr>
<td>FLP30-C</td>
<td>Do not use floating-point variables as loop counters.</td>
<td>2 - Small differences</td>
<td>A6-5-2</td>
</tr>
<tr>
<td>FLP32-C</td>
<td>Prevent or detect domain and range errors in math functions.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>FLP34-C</td>
<td>Ensure that floating-point conversions are within range of the new type.</td>
<td>3 - Significant differences</td>
<td>M5-0-5, M5-0-6, M5-0-7</td>
</tr>
<tr>
<td>FLP36-C</td>
<td>Preserve precision when converting integral values to floating-point type.</td>
<td>3 - Significant differences</td>
<td>M5-0-5, M5-0-6, M5-0-7</td>
</tr>
<tr>
<td>FLP37-C</td>
<td>Do not use object representations to compare floating-point values.</td>
<td>2 - Small differences</td>
<td>M3-9-3</td>
</tr>
<tr>
<td>ENV30-C</td>
<td>Do not modify the object referenced by the return value of certain functions.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>ENV31-C</td>
<td>Do not rely on an environment pointer following an operation that may invalidate it.</td>
<td>4 - Rejected</td>
<td>In general, a project shall not rely on environment-specific implementations.</td>
</tr>
<tr>
<td>ENV32-C</td>
<td>All exit handlers must return normally.</td>
<td>3 - Significant differences</td>
<td>A15-5-2, A15-5-3</td>
</tr>
<tr>
<td>ENV33-C</td>
<td>Do not call system().</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>ENV34-C</td>
<td>Do not store pointers returned by certain functions.</td>
<td>3 - Significant differences</td>
<td>A18-0-3, M19-3-1</td>
</tr>
<tr>
<td>Guidelines</td>
<td>Rule Description</td>
<td>Requirement Level</td>
<td>Reason</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>SIG31-C</td>
<td>Do not access shared objects in signal handlers.</td>
<td>4 - Rejected</td>
<td></td>
</tr>
<tr>
<td>SIG34-C</td>
<td>Do not call signal() from within interruptible signal handlers.</td>
<td>4 - Rejected</td>
<td></td>
</tr>
<tr>
<td>SIG35-C</td>
<td>Do not return from a computational exception signal handler.</td>
<td>4 - Rejected</td>
<td></td>
</tr>
<tr>
<td>PRE30-C</td>
<td>Do not create a universal character name through concatenation.</td>
<td>5 - Not yet analyzed</td>
<td></td>
</tr>
<tr>
<td>PRE31-C</td>
<td>Avoid side effects in arguments to unsafe macros.</td>
<td>3 - Significant differences</td>
<td></td>
</tr>
<tr>
<td>PRE32-C</td>
<td>Do not use preprocessor directives in invocations of function-like macros.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>C++ Core Guidelines Rule:</th>
<th>Relation type:</th>
<th>Related rule:</th>
<th>Comment:</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.1: Express ideas directly in code.</td>
<td>4 - Rejected</td>
<td></td>
<td>The rule is vague.</td>
</tr>
<tr>
<td>P.2: Write in ISO Standard C++.</td>
<td>2 - Small differences</td>
<td>A0-4-3</td>
<td></td>
</tr>
<tr>
<td>P.3: Express intent.</td>
<td>4 - Rejected</td>
<td></td>
<td>The rule is vague.</td>
</tr>
<tr>
<td>P4:</td>
<td>Ideally, a program should be statically type safe.</td>
<td>3 - Significant differences</td>
<td>The rule is covered by: A5-2-1, A5-2-2, A5-2-4, M5-2-12, A8-5-2, M9-5-1</td>
</tr>
<tr>
<td>P5:</td>
<td>Prefer compile-time checking to run-time checking.</td>
<td>3 - Significant differences</td>
<td>M0-3-1</td>
</tr>
<tr>
<td>P6:</td>
<td>What cannot be checked at compile time should be checkable at run time.</td>
<td>3 - Significant differences</td>
<td>A0-1-2, M0-3-2</td>
</tr>
<tr>
<td>P7:</td>
<td>Catch run-time errors early.</td>
<td>3 - Significant differences</td>
<td>A0-1-2, M0-3-2, A5-2-5, A15-0-4, A15-0-5</td>
</tr>
<tr>
<td>P8:</td>
<td>Don’t leak any resources.</td>
<td>3 - Significant differences</td>
<td>A18-5-1, A18-5-2, A15-1-4</td>
</tr>
<tr>
<td>P9:</td>
<td>Don’t waste time or space.</td>
<td>3 - Significant differences</td>
<td>M0-1-1, A0-1-1, M0-1-8, M0-1-9</td>
</tr>
<tr>
<td>P10:</td>
<td>Prefer immutable data to mutable data.</td>
<td>2 - Small differences</td>
<td>A7-1-1</td>
</tr>
<tr>
<td>P11:</td>
<td>Encapsulate messy constructs, rather than spreading through the code.</td>
<td>4 - Rejected</td>
<td>The rule is vague.</td>
</tr>
<tr>
<td>I.1:</td>
<td>Make interfaces explicit.</td>
<td>4 - Rejected</td>
<td>The rule is vague.</td>
</tr>
<tr>
<td>I.2:</td>
<td>Avoid global variables.</td>
<td>2 - Small differences</td>
<td>A3-3-2</td>
</tr>
<tr>
<td>I.3:</td>
<td>Avoid singletons.</td>
<td>2 - Small differences</td>
<td>A3-3-2</td>
</tr>
<tr>
<td>I.4:</td>
<td>Make interfaces precisely and strongly typed.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
</tr>
<tr>
<td>I.5:</td>
<td>State preconditions (if any).</td>
<td>4 - Rejected</td>
<td>The rule is vague.</td>
</tr>
<tr>
<td>I.6:</td>
<td>Prefer Expects() for expressing preconditions.</td>
<td>4 - Rejected</td>
<td>Expects() is not part of Language Standard.</td>
</tr>
<tr>
<td>I.7:</td>
<td>State postconditions.</td>
<td>4 - Rejected</td>
<td>The rule is vague.</td>
</tr>
<tr>
<td>I.8:</td>
<td>Prefer Ensures() for expressing postconditions.</td>
<td>4 - Rejected</td>
<td>Ensures() is not part of Language Standard.</td>
</tr>
<tr>
<td>I.9:</td>
<td>If an interface is a template, document its parameters using concepts.</td>
<td>3 - Significant differences</td>
<td>A14-1-1</td>
</tr>
<tr>
<td>I.10:</td>
<td>Use exceptions to signal a failure to perform a required task.</td>
<td>2 - Small differences</td>
<td>A15-0-1</td>
</tr>
<tr>
<td>I.11:</td>
<td>Never transfer ownership by a raw pointer (T*).</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
</tr>
<tr>
<td>I.12:</td>
<td>Declare a pointer that must not be null as not_null.</td>
<td>4 - Rejected</td>
<td>The not_null is not part of Language Standard.</td>
</tr>
<tr>
<td>I.13: Do not pass an array as a single pointer.</td>
<td>2 - Small differences</td>
<td>M5-2-12</td>
<td></td>
</tr>
<tr>
<td>I.22: Avoid complex initialization of global objects.</td>
<td>3 - Significant differences</td>
<td>A3-3-2</td>
<td></td>
</tr>
<tr>
<td>I.23: Keep the number of function arguments low.</td>
<td>4 - Rejected</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I.24: Avoid adjacent unrelated parameters of the same type.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>I.25: Prefer abstract classes as interfaces to class hierarchies.</td>
<td>3 - Significant differences</td>
<td>A10-1-1</td>
<td></td>
</tr>
<tr>
<td>I.26: If you want a cross-compiler ABI, use a C-style subset.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>F.1: “Package” meaningful operations as carefully named functions.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>F.2: A function should perform a single logical operation.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>F.3: Keep functions short and simple.</td>
<td>4 - Rejected</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.4: If a function may have to be evaluated at compile time, declare it constexpr.</td>
<td>2 - Small differences</td>
<td>A7-1-2</td>
<td></td>
</tr>
<tr>
<td>F.5: If a function is very small and time-critical, declare it inline.</td>
<td>4 - Rejected</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.6: If your function may not throw, declare it noexcept.</td>
<td>2 - Small differences</td>
<td>A15-4-4</td>
<td></td>
</tr>
<tr>
<td>F.7: For general use, take T* or T&amp; arguments rather than smart pointers.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>F.8: Prefer pure functions.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>F.15: Prefer simple and conventional ways of passing information.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>F.16: For “in” parameters, pass cheaply-copied types by value and others by reference to const.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>F.17: For “in-out” parameters, pass by reference to non-const.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>F.18: For “consume” parameters, pass by X&amp;&amp; and std::move the parameter.</td>
<td>5 - Not yet analyzed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.19: For “forward” parameters, pass by TP&amp;&amp; and only std::forward the parameter.</td>
<td>5 - Not yet analyzed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.20: For “out” output values, prefer return values to output parameters.</td>
<td>5 - Not yet analyzed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.21: To return multiple “out” values, prefer returning a tuple or struct.</td>
<td>5 - Not yet analyzed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.22: Use T* or owner&lt;T*&gt; to designate a single object.</td>
<td>3 - Significant differences</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.23: Use a not_null&lt;T&gt; to indicate that “null” is not a valid value.</td>
<td>4 - Rejected</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.24: Use a span&lt;T&gt; or a span_p&lt;T&gt; to designate a half-open sequence.</td>
<td>4 - Rejected</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.25: Use a zstring or a not_null&lt;zstring&gt; to designate a C-style string.</td>
<td>4 - Rejected</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.26: Use a unique_ptr&lt;T&gt; to transfer ownership where a pointer is needed.</td>
<td>4 - R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.27: Use a shared_ptr&lt;T&gt; to share ownership.</td>
<td>5 - Not yet analyzed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.60: Prefer T* over T&amp; when no argument is a valid option.</td>
<td>5 - Not yet analyzed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.42: Return a T* to indicate a position (only).</td>
<td>5 - Not yet analyzed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.43: Never (directly or indirectly) return a pointer or a reference to a local object.</td>
<td>2 - Small differences</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.44: Return a T&amp; when copy is undesirable and returning no object isn’t needed.</td>
<td>5 - Not yet analyzed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.45: Don’t return a T&amp;&amp;.</td>
<td>5 - Not yet analyzed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.46: int is the return type for main().</td>
<td>5 - Not yet analyzed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F.47: Return T&amp; from assignment operators.</td>
<td>5 - Not yet analyzed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Guideline</td>
<td>Description</td>
<td>Analysis</td>
<td>Notes</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>----------</td>
<td>-------</td>
</tr>
<tr>
<td>F.50</td>
<td>Use a lambda when a function won’t do (to capture local variables, or to write a local function).</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
</tr>
<tr>
<td>F.51</td>
<td>Where there is a choice, prefer default arguments over overloading.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
</tr>
<tr>
<td>F.52</td>
<td>Prefer capturing by reference in lambdas that will be used locally, including passed to algorithms.</td>
<td>2 - Small differences</td>
<td>A5-1-5</td>
</tr>
<tr>
<td>F.53</td>
<td>Avoid capturing by reference in lambdas that will be used nonlocally, including returned, stored on the heap, or passed to another thread.</td>
<td>2 - Small differences</td>
<td>A5-1-4</td>
</tr>
<tr>
<td>F.54</td>
<td>If you capture this, capture all variables explicitly (no default capture).</td>
<td>3 - Significant differences</td>
<td>A5-1-2</td>
</tr>
<tr>
<td>C.1</td>
<td>Organize related data into structures (structs or classes).</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
</tr>
<tr>
<td>C.2</td>
<td>Use class if the class has an invariant; use struct if the data members can vary independently.</td>
<td>3 - Significant differences</td>
<td>Class shall be used for all non-POD types (see: A11-0-1), and a struct for types defined in A11-0-2.</td>
</tr>
<tr>
<td>C.3</td>
<td>Represent the distinction between an interface and an implementation using a class.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
</tr>
<tr>
<td>C.4</td>
<td>Make a function a member only if it needs direct access to the representation of a class.</td>
<td>2 - Small differences</td>
<td>M9-3-3</td>
</tr>
<tr>
<td>C.5</td>
<td>Place helper functions in the same namespace as the class they support.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
</tr>
<tr>
<td>C.7</td>
<td>Don’t define a class or enum and declare a variable of its type in the same statement.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
</tr>
<tr>
<td>C.8</td>
<td>Use class rather than struct if any member is non-public.</td>
<td>2 - Small differences</td>
<td>M11-0-1, A11-0-1</td>
</tr>
<tr>
<td>C.9</td>
<td>Minimize exposure of members.</td>
<td>3 - Significant differences</td>
<td>M9-3-1, A9-3-1, M11-0-1</td>
</tr>
<tr>
<td>C.10</td>
<td>Prefer concrete types over class hierarchies.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
</tr>
<tr>
<td>C.11</td>
<td>Make concrete types regular.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
</tr>
<tr>
<td>C.20</td>
<td>If you can avoid defining default operations, do.</td>
<td>2 - Small differences</td>
<td>A12-0-1</td>
</tr>
</tbody>
</table>

Following “the rule of zero” is permitted if no special member functions need to be defined.
| C.21: If you define or `=delete` any default operation, define or `=delete` them all. | 2 - Small differences | A12-0-1 |
| C.22: Make default operations consistent. | 2 - Small differences | A12-1-1, A12-8-1 |
| C.30: Define a destructor if a class needs an explicit action at object destruction. | 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 | A12-4-1 |
| C.36: A destructor may not fail. | 2 - Small differences | A15-5-1 |
| C.37: Make destructors noexcept. | 2 - Small differences | A15-5-1 |
| C.40: Define a constructor if a class has an invariant. | 5 - Not yet analyzed | - |
| C.41: A constructor should create a fully initialized object. | 2 - Small differences | A15-2-2 |
| C.42: If a constructor cannot construct a valid object, throw an exception. | 2 - Small differences | A15-2-2 |
| C.43: Ensure that a class has a default constructor. | 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 | A12-1-3, A12-7-1 |
| C.46: By default, declare single-argument constructors explicit. | 2 - Small differences | A12-1-4 |
| C.47: Define and initialize member variables in the order of member declaration. | 2 - Small differences | A8-5-1 |
| C.48 | Prefer in-class initializers to member initializers in constructors for constant initializers. | 3 - Significant differences | A12-1-3 | AUTOSAR C++ Coding Guidelines states that NSDMI shall not be mixed with member initializer list of constructors, see: A12-1-2. |
| C.49 | Prefer initialization to assignment in constructors. | 2 - Small differences | A12-6-1 | |
| C.50 | Use a factory function if you need “virtual behavior” during initialization. | 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 | A10-3-5, A13-2-1 | |
| C.61 | A copy operation should copy. | 2 - Small differences | A12-8-1, A12-8-2 | |
| C.62 | Make copy assignment safe for self-assignment. | 2 - Small differences | A12-8-5 | |
| C.63 | Make move assignment non-virtual, take the parameter by &&, and return by non-const&. | 2 - Small differences | A10-3-5, A13-2-1 | |
| C.64 | A move operation should move and leave its source in a valid state. | 2 - Small differences | A12-8-1, A12-8-4 | |
| C.65 | Make move assignment safe for self-assignment. | 2 - Small differences | A12-8-5 | |
| C.66 | Make move operations noexcept. | 2 - Small differences | A15-5-1 | |
| C.67 | A base class should suppress copying, and provide a virtual clone instead if copying is desired. | 2 - Small differences | A12-8-6 | |
| C.80 | Use =default if you have to be explicit about using the default semantics. | 2 - Small differences | A12-7-1 | |
| C.81 | Use =delete when you want to disable default behavior (without wanting an alternative). | 5 - Not yet analyzed | - | - |
| C.82 | Don’t call virtual functions in constructors and destructors. | 2 - Small differences | M12-1-1 |
### Guidelines for the use of the C++14 language in critical and safety-related systems

**AUTOSAR AP Release 17-03**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
<th>Type</th>
<th>Analysis</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>C.83:</strong></td>
<td>For value-like types, consider providing a noexcept swap function.</td>
<td>3 - Significant differences</td>
<td>A12-8-2</td>
<td>The swap function is explicitly recommended for copy and move assignment operators only.</td>
</tr>
<tr>
<td><strong>C.84:</strong></td>
<td>A swap function may not fail.</td>
<td>2 - Small differences</td>
<td>A15-5-1</td>
<td></td>
</tr>
<tr>
<td><strong>C.85:</strong></td>
<td>Make swap noexcept.</td>
<td>2 - Small differences</td>
<td>A15-5-1</td>
<td></td>
</tr>
<tr>
<td><strong>C.86:</strong></td>
<td>Make == symmetric with respect to operand types and noexcept.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>C.87:</strong></td>
<td>Beware of == on base classes.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>C.88:</strong></td>
<td>Make a hash noexcept.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>C.120:</strong></td>
<td>Use class hierarchies to represent concepts with inherent hierarchical structure (only).</td>
<td>4 - Rejected</td>
<td>Non-generic design principle; There is no need for a new rule.</td>
<td></td>
</tr>
<tr>
<td><strong>C.121:</strong></td>
<td>If a base class is used as an interface, make it a pure abstract class.</td>
<td>2 - Small differences</td>
<td>AUTOSAR C++ Coding Guidelines defines an interface class definition, see: Interface-Class.</td>
<td></td>
</tr>
<tr>
<td><strong>C.122:</strong></td>
<td>Use abstract classes as interfaces when complete separation of interface and implementation is needed.</td>
<td>4 - Rejected</td>
<td>Non-generic design principle; There is no need for a new rule.</td>
<td></td>
</tr>
<tr>
<td><strong>C.126:</strong></td>
<td>An abstract class typically doesn’t need a constructor.</td>
<td>4 - Rejected</td>
<td>Non-generic design principle; There is no need for a new rule.</td>
<td></td>
</tr>
<tr>
<td><strong>C.127:</strong></td>
<td>A class with a virtual function should have a virtual or protected destructor.</td>
<td>2 - Small differences</td>
<td>A12-4-1</td>
<td></td>
</tr>
<tr>
<td><strong>C.128:</strong></td>
<td>Virtual functions should specify exactly one of virtual, override, or final.</td>
<td>2 - Small differences</td>
<td>A10-3-1</td>
<td></td>
</tr>
<tr>
<td><strong>C.129:</strong></td>
<td>When designing a class hierarchy, distinguish between implementation inheritance and interface inheritance.</td>
<td>4 - Rejected</td>
<td>Non-generic design principle; There is no need for a new rule.</td>
<td></td>
</tr>
<tr>
<td>C.130: Redefine or prohibit copying for a base class; prefer a virtual clone function instead.</td>
<td>2 - Small differences</td>
<td>A12-8-6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------------------------------------------------------</td>
<td>-----------------------</td>
<td>---------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C.131: Avoid trivial getters and setters.</td>
<td>4 - Rejected</td>
<td>All members in non-POD types shall be private.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C.132: Don't make a function virtual without reason.</td>
<td>4 - Rejected</td>
<td>Non-generic design principle; There is no need for a new rule.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C.133: Avoid protected data.</td>
<td>3 - Significant differences</td>
<td>All members in non-POD types shall be private.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C.134: Ensure all non-const data members have the same access level.</td>
<td>2 - Small differences</td>
<td>M11-0-1, A11-0-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C.135: Use multiple inheritance to represent multiple distinct interfaces.</td>
<td>2 - Small differences</td>
<td>A10-1-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C.136: Use multiple inheritance to represent the union of implementation attributes.</td>
<td>4 - Rejected</td>
<td>Multiple implementation inheritance is prohibited by AUTOSAR C++ Coding Guidelines, it allows only multiple interface inheritance.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C.137: Use virtual bases to avoid overly general base classes.</td>
<td>4 - Rejected</td>
<td>It is allowed to use virtual inheritance only in a diamond hierarchy, see: M10-1-1, M10-1-2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C.138: Create an overload set for a derived class and its bases with using.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C.139: Use final sparingly.</td>
<td>4 - Rejected</td>
<td>Non-generic design principle; There is no need for a new rule.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C.140: Do not provide different default arguments for a virtual function and an overrider.</td>
<td>2 - Small differences</td>
<td>M8-3-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C.145: Access polymorphic objects through pointers and references.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C.146: Use dynamic_cast where class hierarchy navigation is unavoidable.</td>
<td>2 - Small differences</td>
<td>A5-2-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clause</td>
<td>Description</td>
<td>Rejection</td>
<td>Explanation</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------</td>
<td>-------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>C.147</td>
<td>Use dynamic_cast to a reference type when failure to find the required class is considered an error.</td>
<td>4 - Rejected</td>
<td>The dynamic_cast should not be used, see: A5-2-1.</td>
<td></td>
</tr>
<tr>
<td>C.148</td>
<td>Use dynamic_cast to a pointer type when failure to find the required class is considered a valid alternative.</td>
<td>4 - Rejected</td>
<td>The dynamic_cast should not be used, see: A5-2-1.</td>
<td></td>
</tr>
<tr>
<td>C.149</td>
<td>Use unique_ptr or shared_ptr to avoid forgetting to delete objects created using new.</td>
<td>2 - Small differences</td>
<td>A18-5-2</td>
<td></td>
</tr>
<tr>
<td>C.150</td>
<td>Use make_unique() to construct objects owned by unique_ptrs.</td>
<td>2 - Small differences</td>
<td>A18-5-2</td>
<td></td>
</tr>
<tr>
<td>C.151</td>
<td>Use make_shared() to construct objects owned by shared_ptrs.</td>
<td>2 - Small differences</td>
<td>A18-5-2</td>
<td></td>
</tr>
<tr>
<td>C.152</td>
<td>Never assign a pointer to an array of derived class objects to a pointer to its base.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>C.160</td>
<td>Define operators primarily to mimic conventional usage.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>C.161</td>
<td>Use nonmember functions for symmetric operators.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>C.162</td>
<td>Overload operations that are roughly equivalent.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>C.163</td>
<td>Overload only for operations that are roughly equivalent.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>C.164</td>
<td>Avoid conversion operators.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>C.165</td>
<td>Use using for customization points.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>C.166</td>
<td>Overload unary &amp; only as part of a system of smart pointers and references.</td>
<td>3 - Significant differences</td>
<td>M5-3-3 The unary &amp; operator shall not be overloaded.</td>
<td></td>
</tr>
<tr>
<td>C.168</td>
<td>Define overloaded operators in the namespace of their operands.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>C.167</td>
<td>Use an operator for an operation with its conventional meaning.</td>
<td>4 - Rejected</td>
<td>Design principle; There is no need for a new rule.</td>
<td></td>
</tr>
<tr>
<td>C.170</td>
<td>If you feel like overloading a lambda, use a generic lambda.</td>
<td>4 - Rejected</td>
<td>Design principle; There is no need for a new rule. Creating generic lambda expressions is allowed, see: A7-1-5.</td>
<td></td>
</tr>
<tr>
<td>C.180: Use unions to save memory.</td>
<td>4 - Rejected</td>
<td>Unions shall not be used, see: M9-5-1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------------------</td>
<td>--------------</td>
<td>----------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C.181: Avoid &quot;naked&quot; unions.</td>
<td>2 - Small differences</td>
<td>M9-5-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C.182: Use anonymous unions to implement tagged unions.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C.183: Don’t use a union for type punning.</td>
<td>4 - Rejected</td>
<td>Unions shall not be used, see: M9-5-1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enum.1: Prefer enumerations over macros.</td>
<td>3 - Significant differences</td>
<td>A16-0-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enum.2: Use enumerations to represent sets of related named constants.</td>
<td>4 - Rejected</td>
<td>Design principle; There is no need for a new rule.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enum.3: Prefer class enums over &quot;plain&quot; enums.</td>
<td>2 - Small differences</td>
<td>A7-2-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enum.4: Define operations on enumerations for safe and simple use.</td>
<td>4 - Rejected</td>
<td>Design principle; There is no need for a new rule.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enum.5: Don’t use ALL_CAPS for enumerators.</td>
<td>4 - Rejected</td>
<td>AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enum.6: Avoid unnamed enumerations.</td>
<td>3 - Significant differences</td>
<td>A7-2-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enum.7: Specify the underlying type of an enumeration only when necessary.</td>
<td>4 - Rejected</td>
<td>Enum classes shall be used instead of enums; it is not allowed to declare unnamed enum class.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AUTOSAR C++ Coding Guidelines forces a programmer to specify the underlying base type explicitly, as only fixed-width numeric types shall be used. See: A3-9-1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enum.8: Specify enumerator values only when necessary.</td>
<td>3 - Significant differences</td>
<td>A7-2-4</td>
<td>It is defined how enumerators values should be specified.</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------</td>
<td>-----------------------------</td>
<td>--------</td>
<td>----------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>R.1: Manage resources automatically using resource handles and RAII (Resource Acquisition Is Initialization).</td>
<td>4 - Rejected</td>
<td></td>
<td>Design principle; There is no need for a new rule.</td>
<td></td>
</tr>
<tr>
<td>R.2: In interfaces, use raw pointers to denote individual objects (only).</td>
<td>4 - Rejected</td>
<td></td>
<td>Design principle; There is no need for a new rule.</td>
<td></td>
</tr>
<tr>
<td>R.3: A raw pointer (a $T^*$) is non-owning.</td>
<td>4 - Rejected</td>
<td></td>
<td>Design principle; There is no need for a new rule.</td>
<td></td>
</tr>
<tr>
<td>R.4: A raw reference (a $T$&amp;$*$) is non-owning.</td>
<td>4 - Rejected</td>
<td></td>
<td>Design principle; There is no need for a new rule.</td>
<td></td>
</tr>
<tr>
<td>R.5: Don’t heap-allocate unnecessarily.</td>
<td>4 - Rejected</td>
<td></td>
<td>Design principle; There is no need for a new rule.</td>
<td></td>
</tr>
<tr>
<td>R.6: Avoid non-const global variables.</td>
<td>3 - Significant differences</td>
<td>A3-3-2</td>
<td>There shall be no non-POD type objects with static storage duration besides static constexpr variables.</td>
<td></td>
</tr>
<tr>
<td>R.10: Avoid malloc() and free().</td>
<td>2 - Small differences</td>
<td>A18-5-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R.11: Avoid calling new and delete explicitly.</td>
<td>2 - Small differences</td>
<td>A18-5-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R.12: Immediately give the result of an explicit resource allocation to a manager object.</td>
<td>2 - Small differences</td>
<td>A18-5-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R.13: Perform at most one explicit resource allocation in a single expression statement.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>R.14: ??? array vs. pointer parameter.</td>
<td>2 - Small differences</td>
<td>M5-2-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R.15: Always overload matched allocation/deallocation pairs.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>R.20: Use unique_ptr or shared_ptr to represent ownership.</td>
<td>4 - Rejected</td>
<td>Design principle: There is no need for a new rule.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R.21: Prefer unique_ptr over shared_ptr unless you need to share ownership.</td>
<td>4 - Rejected</td>
<td>Design principle: There is no need for a new rule.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R.22: Use make_shared() to make shared_ptrs.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R.23: Use make_unique() to make unique_ptrs.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R.24: Use std::weak_ptr to break cycles of shared_ptrs.</td>
<td>4 - Rejected</td>
<td>Design principle: There is no need for a new rule.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R.30: Take smart pointers as parameters only to explicitly express lifetime semantics.</td>
<td>4 - Rejected</td>
<td>Design principle: There is no need for a new rule.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R.31: If you have non-std smart pointers, follow the basic pattern from std.</td>
<td>4 - Rejected</td>
<td>Design principle: There is no need for a new rule.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R.32: Take a unique_ptr&lt;widget&gt; parameter to express that a function assumes ownership of a widget.</td>
<td>4 - Rejected</td>
<td>Design principle: There is no need for a new rule.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R.33: Take a unique_ptr&lt;widget&gt;&amp; parameter to express that a function reseats the widget.</td>
<td>4 - Rejected</td>
<td>Design principle: There is no need for a new rule.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R.34: Take a shared_ptr&lt;widget&gt; parameter to express that a function is part owner.</td>
<td>4 - Rejected</td>
<td>Design principle: There is no need for a new rule.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R.35: Take a shared_ptr&lt;widget&gt;&amp; parameter to express that a function might reseat the shared pointer.</td>
<td>4 - Rejected</td>
<td>Design principle: There is no need for a new rule.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R.36: Take a const shared_ptr&lt;widget&gt;&amp; parameter to express that it might retain a reference count to the object ???</td>
<td>4 - Rejected</td>
<td>Design principle: There is no need for a new rule.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ES</td>
<td>Description</td>
<td>Rating</td>
<td>Notes</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-------------</td>
<td>--------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>R.37</td>
<td>Do not pass a pointer or reference obtained from an aliased smart pointer.</td>
<td>4 - Rejected</td>
<td>Design principle: There is no need for a new rule.</td>
<td></td>
</tr>
<tr>
<td>ES.1</td>
<td>Prefer the standard library to other libraries and to “handcrafted code”.</td>
<td>4 - Rejected</td>
<td>Design principle: There is no need for a new rule.</td>
<td></td>
</tr>
<tr>
<td>ES.2</td>
<td>Prefer suitable abstractions to direct use of language features.</td>
<td>4 - Rejected</td>
<td>Design principle: There is no need for a new rule.</td>
<td></td>
</tr>
<tr>
<td>ES.5</td>
<td>Keep scopes small.</td>
<td>2 - Small differences</td>
<td>M3-4-1</td>
<td></td>
</tr>
<tr>
<td>ES.6</td>
<td>Declare names in for-statement initializers and conditions to limit scope.</td>
<td>2 - Small differences</td>
<td>M3-4-1</td>
<td></td>
</tr>
<tr>
<td>ES.7</td>
<td>Keep common and local names short, and keep uncommon and nonlocal names longer.</td>
<td>4 - Rejected</td>
<td>AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.</td>
<td></td>
</tr>
<tr>
<td>ES.8</td>
<td>Avoid similar-looking names.</td>
<td>2 - Small differences</td>
<td>M2-10-1</td>
<td></td>
</tr>
<tr>
<td>ES.9</td>
<td>Avoid ALL_CAPS names.</td>
<td>4 - Rejected</td>
<td>AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.</td>
<td></td>
</tr>
<tr>
<td>ES.10</td>
<td>Declare one name (only) per declaration.</td>
<td>2 - Small differences</td>
<td>A7-1-7</td>
<td></td>
</tr>
<tr>
<td>ES.11</td>
<td>Use auto to avoid redundant repetition of type names.</td>
<td>3 - Significant differences</td>
<td>A7-1-5</td>
<td></td>
</tr>
<tr>
<td>ES.12</td>
<td>Do not reuse names in nested scopes.</td>
<td>2 - Small differences</td>
<td>A2-11-1</td>
<td></td>
</tr>
<tr>
<td>ES.20</td>
<td>Always initialize an object.</td>
<td>2 - Small differences</td>
<td>M8-5-1</td>
<td></td>
</tr>
<tr>
<td>ES.21</td>
<td>Don’t introduce a variable (or constant) before you need to use it.</td>
<td>2 - Small differences</td>
<td>M3-4-1</td>
<td></td>
</tr>
<tr>
<td>ES.22:</td>
<td>Don’t declare a variable until you have a value to initialize it with.</td>
<td>2 - Small differences</td>
<td>M3-4-1, M8-5-1</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>---------------------------------------------------------------------</td>
<td>----------------------</td>
<td>----------------</td>
<td></td>
</tr>
<tr>
<td>ES.23:</td>
<td>Prefer the () initializer syntax.</td>
<td>2 - Small differences</td>
<td>A8-5-2</td>
<td></td>
</tr>
<tr>
<td>ES.24:</td>
<td>Use a unique_ptr&lt;T&gt; to hold pointers.</td>
<td>3 - Significant differences</td>
<td>A18-5-2, A15-1-4, A18-1-3</td>
<td></td>
</tr>
<tr>
<td>ES.25:</td>
<td>Declare an object const or constexpr unless you want to modify its value later on.</td>
<td>2 - Small differences</td>
<td>A7-1-1</td>
<td></td>
</tr>
<tr>
<td>ES.26:</td>
<td>Don’t use a variable for two unrelated purposes.</td>
<td>4 - Rejected</td>
<td>Implementation principle: There is no need for a new rule.</td>
<td></td>
</tr>
<tr>
<td>ES.27:</td>
<td>Use std::array or stack_array for arrays on the stack.</td>
<td>3 - Significant differences</td>
<td>A18-1-1</td>
<td></td>
</tr>
<tr>
<td>ES.28:</td>
<td>Use lambdas for complex initialization, especially of const variables.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>ES.30:</td>
<td>Don’t use macros for program text manipulation.</td>
<td>2 - Small differences</td>
<td>A16-0-1</td>
<td></td>
</tr>
<tr>
<td>ES.31:</td>
<td>Don’t use macros for constants or “functions”.</td>
<td>2 - Small differences</td>
<td>A16-0-1</td>
<td></td>
</tr>
<tr>
<td>ES.32:</td>
<td>Use ALLCAPS for all macro names.</td>
<td>4 - Rejected</td>
<td>Usage of macros is prohibited.</td>
<td></td>
</tr>
<tr>
<td>ES.33:</td>
<td>If you must use macros, give them unique names.</td>
<td>2 - Small differences</td>
<td>M2-10-1</td>
<td></td>
</tr>
<tr>
<td>ES.34:</td>
<td>Don’t define a (C-style) variadic function.</td>
<td>2 - Small differences</td>
<td>A8-4-1</td>
<td></td>
</tr>
<tr>
<td>ES.70: Prefer a switch-statement to an if-statement when there is a choice.</td>
<td>4 - Rejected</td>
<td>Design principle: The switch statement shall have at least two case-clauses, distinct from the default label. See: A6-4-1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ES.71: Prefer a range-for-statement to a for-statement when there is a choice.</td>
<td>3 - Significant differences</td>
<td>A6-5-1</td>
<td>It is recommended to use range-based for statement to replace equivalent for-statements.</td>
<td></td>
</tr>
<tr>
<td>ES.72: Prefer a for-statement to a while-statement when there is an obvious loop variable.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>ES.73: Prefer a while-statement to a for-statement when there is no obvious loop variable.</td>
<td>3 - Significant differences</td>
<td>A6-5-2</td>
<td>It is required that a for-loop contains a loop-counter.</td>
<td></td>
</tr>
<tr>
<td>ES.74: Prefer to declare a loop variable in the initializer part of a for-statement.</td>
<td>3 - Significant differences</td>
<td>M3-4-1</td>
<td>It is required that each identifier is defined in a block that minimizes its visibility.</td>
<td></td>
</tr>
<tr>
<td>ES.75: Avoid do-statements.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>ES.76: Avoid goto.</td>
<td>2 - Small differences</td>
<td>A6-6-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ES.78: Always end a non-empty case with a break.</td>
<td>2 - Small differences</td>
<td>M6-4-5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ES.85: Make empty statements visible.</td>
<td>2 - Small differences</td>
<td>M6-3-1, M6-4-1, M6-4-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ES.88: Avoid modifying loop control variables inside the body of raw for-loops.</td>
<td>2 - Small differences</td>
<td>M6-5-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ES.40: Avoid complicated expressions.</td>
<td>4 - Rejected</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ES.41: If in doubt about operator precedence, parenthesize.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>ES.42: Keep use of pointers simple and straightforward.</td>
<td>4 - Rejected</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ES.43: Avoid expressions with undefined order of evaluation.</td>
<td>2 - Small differences</td>
<td>A5-0-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ES.44: Don’t depend on order of evaluation of function arguments.</td>
<td>2 - Small differences</td>
<td>A5-0-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ES.45: Avoid &quot;magic constants&quot;; use symbolic constants.</td>
<td>2 - Small differences</td>
<td>A5-1-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ES.48: Avoid lossy (narrowing, truncating) arithmetic conversions.</td>
<td>2 - Small differences</td>
<td>A4-7-1, M5-0-6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ES.47: Use nullptr rather than 0 or NULL.</td>
<td>2 - Small differences</td>
<td>A4-10-1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Guidelines for the use of the C++14 language in critical and safety-related systems

**AUTOSAR AP Release 17-03**

#### Guidelines

<table>
<thead>
<tr>
<th>ES.48: Avoid casts.</th>
<th>2 - Small differences</th>
<th>A5-2-1, A5-2-2, A5-2-3, A5-2-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ES.49: If you must use a cast, use a named cast.</td>
<td>2 - Small differences</td>
<td>A5-2-2</td>
</tr>
<tr>
<td>ES.50: Don’t cast away const.</td>
<td>2 - Small differences</td>
<td>A5-2-3</td>
</tr>
<tr>
<td>ES.55: Avoid the need for range checking.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
</tr>
<tr>
<td>ES.56: Write std::move() only when you need to explicitly move an object to another scope.</td>
<td>3 - Significant differences</td>
<td>A12-8-3, A18-9-2, A18-9-3</td>
</tr>
<tr>
<td>ES.60: Avoid new and delete outside resource management functions.</td>
<td>2 - Small differences</td>
<td>A18-5-2</td>
</tr>
<tr>
<td>ES.61: Delete arrays using delete[] and non-arrays using delete.</td>
<td>2 - Small differences</td>
<td>A18-5-3</td>
</tr>
<tr>
<td>ES.62: Don’t compare pointers into different arrays.</td>
<td>2 - Small differences</td>
<td>M5-0-16</td>
</tr>
<tr>
<td>ES.63: Don’t slice.</td>
<td>3 - Significant differences</td>
<td>A12-8-6, A15-3-5</td>
</tr>
<tr>
<td>ES.100: Don’t mix signed and unsigned arithmetic.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
</tr>
<tr>
<td>ES.101: Use unsigned types for bit manipulation.</td>
<td>2 - Small differences</td>
<td>M5-0-21</td>
</tr>
<tr>
<td>ES.102: Use signed types for arithmetic.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
</tr>
<tr>
<td>ES.103: Don’t overflow.</td>
<td>2 - Small differences</td>
<td>A4-7-1</td>
</tr>
<tr>
<td>ES.104: Don’t underflow.</td>
<td>2 - Small differences</td>
<td>A4-7-1</td>
</tr>
<tr>
<td>ES.105: Don’t divide by zero.</td>
<td>2 - Small differences</td>
<td>A5-5-1</td>
</tr>
<tr>
<td>Per.1: Don’t optimize without reason.</td>
<td>4 - Rejected</td>
<td>Implementation principle: There is no need for a new rule.</td>
</tr>
<tr>
<td>Per.2: Don’t optimize prematurely.</td>
<td>4 - Rejected</td>
<td>Implementation principle: There is no need for a new rule.</td>
</tr>
<tr>
<td>Per.3: Don’t optimize something that’s not performance critical.</td>
<td>4 - Rejected</td>
<td>Implementation principle: There is no need for a new rule.</td>
</tr>
<tr>
<td>Per.4: Don’t assume that complicated code is necessarily faster than simple code.</td>
<td>4 - Rejected</td>
<td>Implementation principle: There is no need for a new rule.</td>
</tr>
<tr>
<td>Per.5: Don’t assume that low-level code is necessarily faster than high-level code.</td>
<td>4 - Rejected</td>
<td>Implementation principle: There is no need for a new rule.</td>
</tr>
<tr>
<td>Per.6: Don’t make claims about performance without measurements.</td>
<td>4 - Rejected</td>
<td>Implementation principle: There is no need for a new rule.</td>
</tr>
<tr>
<td>Per.7: Design to enable optimization.</td>
<td>4 - Rejected</td>
<td>Design principle: There is no need for a new rule.</td>
</tr>
<tr>
<td>Per.10: Rely on the static type system.</td>
<td>4 - Rejected</td>
<td>Implementation principle: There is no need for a new rule.</td>
</tr>
<tr>
<td>Per.19: Access memory predictably.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
</tr>
<tr>
<td>CP.1: Assume that your code will run as part of a multi-threaded program.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>CP.2: Avoid data races.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>CP.3: Minimize explicit sharing of writable data.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>CP.4: Think in terms of tasks, rather than threads.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>CP8: Don’t try to use volatile for synchronization.</td>
<td>55 - Not yet analyzed</td>
<td>The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>CP20: Use RAII, never plain lock()/unlock().</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>CP21: Use std::lock() to acquire multiple mutexes.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>CP22: Never call unknown code while holding a lock (e.g., a callback).</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>CP23: Think of a joining thread as a scoped container.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>CP24: Think of a detached thread as a global container.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>CP25: Prefer gsl::raii_thread over std::thread unless you plan to detach().</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>CP</td>
<td>Guidelines</td>
<td>Analysis Note</td>
</tr>
<tr>
<td>----</td>
<td>------------</td>
<td>---------------</td>
</tr>
<tr>
<td>CP26:</td>
<td>Prefer <code>gsl::detached_thread</code> over <code>std::thread</code> if you plan to <code>detach()</code></td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>CP27:</td>
<td>Use plain <code>std::thread</code> for threads that detach based on a run-time condition (only).</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>CP28:</td>
<td>Remember to join scoped threads that are not <code>detach()</code>ed.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>CP30:</td>
<td>Do not pass pointers to local variables to non-RAII threads.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>CP31:</td>
<td>Pass small amounts of data between threads by value, rather than by reference or pointer.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>CP32:</td>
<td>To share ownership between unrelated threads use <code>shared_ptr</code>.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>CP40:</td>
<td>Minimize context switching.</td>
<td>5 - Not yet analyzed</td>
</tr>
<tr>
<td>CP.41: Minimize thread creation and destruction.</td>
<td>55 - Not yet analyzed</td>
<td>The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>CP.42: Don’t wait without a condition.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>CP.43: Minimize time spent in a critical section.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>CP.44: Remember to name your lock guards and unique_locks.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>P.50: Define a mutex together with the data it guards.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>CP.60: Use a future to return a value from a concurrent task.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>CP.61: Use a async() to spawn a concurrent task.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>CP.100: Don't use lock-free programming unless you absolutely have to.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------</td>
<td>---------------------------------------------------------------------</td>
</tr>
<tr>
<td>CP.101: Distrust your hardware/compiler combination.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>CP.102: Carefully study the literature.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>CP.110: Do not write your own double-checked locking for initialization.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>CP.111: Use a conventional pattern if you really need double-checked locking.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>CP.200: Use volatile only to talk to non-C++ memory.</td>
<td>5 - Not yet analyzed</td>
<td>The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.</td>
</tr>
<tr>
<td>E.1: Develop an error-handling strategy early in a design.</td>
<td>4 - Rejected</td>
<td>Design principle; There is no need for a new rule.</td>
</tr>
<tr>
<td>E.2: Throw an exception to signal that a function can't perform its assigned task.</td>
<td>2 - Small differences</td>
<td>A15-0-1</td>
</tr>
<tr>
<td>E.3: Use exceptions for error handling only.</td>
<td>2 - Small differences</td>
<td>A15-0-1</td>
</tr>
<tr>
<td>E.4:</td>
<td>Design your error-handling strategy around invariants.</td>
<td>4 - Rejected</td>
</tr>
<tr>
<td>-------</td>
<td>--------------------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>E.5:</td>
<td>Let a constructor establish an invariant, and throw if it cannot.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>E.6:</td>
<td>Use RAII to prevent leaks.</td>
<td>4 - Rejected</td>
</tr>
<tr>
<td>E.7:</td>
<td>State your preconditions.</td>
<td>4 - Rejected</td>
</tr>
<tr>
<td>E.8:</td>
<td>State your postconditions.</td>
<td>4 - Rejected</td>
</tr>
<tr>
<td>E.12:</td>
<td>Use noexcept when exiting a function because of a throw is impossible or unacceptable.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>E.13:</td>
<td>Never throw while being the direct owner of an object.</td>
<td>3 - Significant differences</td>
</tr>
<tr>
<td>E.14:</td>
<td>Use purpose-designed user-defined types as exceptions (not built-in types).</td>
<td>3 - Significant differences</td>
</tr>
<tr>
<td>E.15:</td>
<td>Catch exceptions from a hierarchy by reference.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>E.16:</td>
<td>Destructors, deallocation, and swap must never fail.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>E.17: Don't try to catch every exception in every function.</td>
<td>2 - Small differences</td>
<td>A15-3-1, A15-3-2</td>
</tr>
<tr>
<td>E.18: Minimize the use of explicit try/catch.</td>
<td>4 - Rejected</td>
<td>Implementation principle: There is no need for a new rule.</td>
</tr>
<tr>
<td>E.19: Use a final_action object to express cleanup if no suitable resource handle is available.</td>
<td>4 - Rejected</td>
<td>The finally is not part of the C++ Language Standard.</td>
</tr>
<tr>
<td>E.25: If you can't throw exceptions, simulate RAII for resource management.</td>
<td>3 - Rejected</td>
<td>the RAII is a coding pattern; There is no need for a new rule. On the other hand, usage of RAII is recommended in the example of the A15-1-4.</td>
</tr>
<tr>
<td>E.26: If you can't throw exceptions, consider failing fast.</td>
<td>4 - Rejected</td>
<td>Implementation principle: There is no need for a new rule.</td>
</tr>
<tr>
<td>E.27: If you can't throw exceptions, use error codes systematically.</td>
<td>4 - Rejected</td>
<td>AUTOSAR C++ Coding Guidelines does not force any specific error handling mechanism. It requires that every error information will be tested, see: M0-3-2.</td>
</tr>
<tr>
<td>E.28: Avoid error handling based on global state (e.g. errno).</td>
<td>2 - Small differences</td>
<td>M19-3-1</td>
</tr>
<tr>
<td>Con.1: By default, make objects immutable.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
</tr>
<tr>
<td>Con.2: By default, make member functions const.</td>
<td>2 - Small differences</td>
<td>M9-3-3</td>
</tr>
<tr>
<td>Con.3: By default, pass pointers and references to const.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
</tr>
<tr>
<td>Con.4: Use const to define objects with values that do not change after construction.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
</tr>
<tr>
<td>Con.5: Use constexpr for values that can be computed at compile time.</td>
<td>2 - Small differences</td>
<td>A7-1-2</td>
</tr>
<tr>
<td>T.1: Use templates to raise the level of abstraction of code.</td>
<td>4 - Rejected</td>
<td>Design principle; There is no need for a new rule.</td>
</tr>
<tr>
<td>T.2: Use templates to express algorithms that apply to many argument types.</td>
<td>4 - Rejected</td>
<td>Design principle; There is no need for a new rule.</td>
</tr>
<tr>
<td>T.3: Use templates to express containers and ranges.</td>
<td>4 - Rejected</td>
<td>Design principle; There is no need for a new rule.</td>
</tr>
<tr>
<td>T.4: Use templates to express syntax tree manipulation.</td>
<td>4 - Rejected</td>
<td>Design principle; There is no need for a new rule.</td>
</tr>
<tr>
<td>T.5: Combine generic and OO techniques to amplify their strengths, not their costs.</td>
<td>4 - Rejected</td>
<td>Design principle; There is no need for a new rule.</td>
</tr>
<tr>
<td>T.10: Specify concepts for all template arguments.</td>
<td>4 - Rejected</td>
<td>Concepts are not part of the C++14 Language Standard.</td>
</tr>
<tr>
<td>T.11: Whenever possible use standard concepts.</td>
<td>4 - Rejected</td>
<td>Concepts are not part of the C++14 Language Standard.</td>
</tr>
<tr>
<td>T.12: Prefer concept names over auto for local variables.</td>
<td>4 - Rejected</td>
<td>Concepts are not part of the C++14 Language Standard.</td>
</tr>
<tr>
<td>T.13: Prefer the shorthand notation for simple, single-type argument concepts.</td>
<td>4 - Rejected</td>
<td>Concepts are not part of the C++14 Language Standard.</td>
</tr>
</tbody>
</table>
### Guidelines for the use of the C++14 language in critical and safety-related systems

**AUTOSAR AP Release 17-03**

| 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 | A7-1-6 |
| 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 | A14-1-1, A14-7-1 | |
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 | A8-5-2 | |
T.69: Inside a template, don’t make an unqualified nonmember function call unless you intend it to be a customization point. | 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 | A8-4-1 | |
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 | - | - |
<p>| 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 | A14-8-1 |  |
| T.150: | Check that a class matches a concept using static_assert. | 2 - Small differences | A14-1-1 |  |
| CPL.1: | Prefer C++ to C. | 2 - Small differences | A17-1-1, A18-0-1 |  |
| CPL.2: | If you must use C, use the common subset of C and C++, and compile the C code as C++. | 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 .cpp suffix for code files and .h for interface files if your project doesn’t already follow another convention. | 3 - Significant differences | A3-1-2, A3-1-3 | For header file names, AUTOSAR C++ Coding Guidelines allows either “.h”, “.hpp” or “.hxx” extension. |
| SF.2: | A .h file may not contain object definitions or non-inline function definitions. | 2 - Small differences | A3-1-1 |  |
| SF.3: | Use .h files for all declarations used in multiple source files. | 2 - Small differences | M3-2-2, A3-3-1 |  |
| SF.4: | Include .h files before other declarations in a file. | 2 - Small differences | M16-0-1 |  |
| SF.5: | A .cpp file must include the .h file(s) that defines its interface. | 5 - Not yet analyzed |  |  |
| SF.7: | Don’t write using namespace in a header file. | 2 - Small differences | M7-3-6 |  |
| SF.8: | Use #include guards for all .h files. | 2 - Small differences | M16-2-3 |  |
| SF.9: | Avoid cyclic dependencies among source files. | 5 - Not yet analyzed |  |  |
| SF.21: | Don’t use an unnamed (anonymous) namespace in a header. | 2 - Small differences | M7-3-3 |  |
| SF.22: | Use an unnamed (anonymous) namespace for all internal/nonexported entities. | 5 - Not yet analyzed |  |  |
| SL.1: | Use libraries wherever possible. | 4 - Rejected | Design principle: There is no need for a new rule. |
| SL.2: | Prefer the standard library to other libraries. | 4 - Rejected | Design principle; There is no need for a new rule. |
| SL.con.1: | Prefer using STL array or vector instead of a C array. | 2 - Small differences | A18-1-1 |
| SL.con.2: | Prefer using STL vector by default unless you have a reason to use a different container. | 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 | A5-2-4 |
| Type.2: | Don’t use static_cast downcasts. Use dynamic_cast instead. | 2 - Small differences | M5-2-2 |
| Type.3: | Don’t use const_cast to cast away const (i.e., at all). | 2 - Small differences | A5-2-3 |
| Type.4: | Don’t use C-style (T) expression casts that would perform a static_cast downcast, const_cast, or reinterpret_cast. | 2 - Small differences | A5-2-2 |
| Type.5: | Don’t use a variable before it has been initialized. | 2 - Small differences | M8-5-1 |
| Type.6: | Always initialize a member variable. | 2 - Small differences | M8-5-1 |
| Type.7: | Avoid accessing members of raw unions. Prefer variant instead. | 2 - Small differences | M9-5-1 |
| Type.8: | Avoid reading from varargs or passing vararg arguments. Prefer variadic template parameters instead. | 2 - Small differences | A8-4-1 |
| Bounds.1: | Don’t use pointer arithmetic. Use span instead. | 2 - Small differences | M5-0-15 |
| Bounds.2: | Only index into arrays using constant expressions. | 2 - Small differences | A5-2-5 |
| Bounds.3: | No array-to-pointer decay. | 2 - Small differences | M5-2-12 |
| Bounds.4: | Don’t use standard library functions and types that are not bounds-checked. | 5 - Not yet analyzed | - |
| NL.1: | Don’t say in comments what can be clearly stated in code. | 4 - Rejected | AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention. |</p>
<table>
<thead>
<tr>
<th>Guideline</th>
<th>Decision</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>NL.2: State intent in comments.</td>
<td>4 - Rejected</td>
<td>AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.</td>
</tr>
<tr>
<td>NL.3: Keep comments crisp.</td>
<td>4 - Rejected</td>
<td>AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.</td>
</tr>
<tr>
<td>NL.4: Maintain a consistent indentation style.</td>
<td>4 - Rejected</td>
<td>AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.</td>
</tr>
<tr>
<td>NL.5: Don’t encode type information in names.</td>
<td>4 - Rejected</td>
<td>AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.</td>
</tr>
<tr>
<td>NL.7: Make the length of a name roughly proportional to the length of its scope.</td>
<td>4 - Rejected</td>
<td>AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.</td>
</tr>
<tr>
<td>NL.8: Use a consistent naming style.</td>
<td>4 - Rejected</td>
<td>AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.</td>
</tr>
<tr>
<td>Guideline</td>
<td>Original Text</td>
<td>Rejected Reason</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>NL.9</td>
<td>Use ALL_CAPS for macro names only.</td>
<td>4 - Rejected</td>
</tr>
<tr>
<td>NL.10</td>
<td>Avoid CamelCase.</td>
<td>4 - Rejected</td>
</tr>
<tr>
<td>NL.15</td>
<td>Use spaces sparingly.</td>
<td>4 - Rejected</td>
</tr>
<tr>
<td>NL.16</td>
<td>Use a conventional class member declaration order.</td>
<td>4 - Rejected</td>
</tr>
<tr>
<td>NL.17</td>
<td>Use K&amp;R-derived layout.</td>
<td>4 - Rejected</td>
</tr>
<tr>
<td>NL.18</td>
<td>Use C++-style declarator layout.</td>
<td>4 - Rejected</td>
</tr>
<tr>
<td>NL.19</td>
<td>Avoid names that are easily misread.</td>
<td>2 - Small differences</td>
</tr>
<tr>
<td>NL.20</td>
<td>Don’t place two statements on the same line.</td>
<td>3 - Significant differences</td>
</tr>
<tr>
<td>NL.21: Declare one name (only) per declaration.</td>
<td>2 - Small differences</td>
<td>A7-1-7</td>
</tr>
<tr>
<td>NL.25: Don't use void as an argument type.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
</tr>
<tr>
<td>NL.26: Use conventional const notation.</td>
<td>5 - Not yet analyzed</td>
<td>-</td>
</tr>
</tbody>
</table>

Table A.5: ISOCPP
B Glossary

<table>
<thead>
<tr>
<th>Abbreviation / Acronym:</th>
<th>Description:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real-time application (RTA)</td>
<td>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.</td>
</tr>
<tr>
<td>HIC++</td>
<td>High Integrity C++ Coding Standard.</td>
</tr>
<tr>
<td>cvalue expression</td>
<td>An expression that should not undergo further conversions, either implicitly or explicitly, is called a cvalue expression.</td>
</tr>
<tr>
<td>Ownership</td>
<td>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.</td>
</tr>
</tbody>
</table>
### One definition rule

The rule states that:

- There shall be one and only one definition of any variable, function, class type, enumeration type, or template in a translation unit. Some of these may have multiple declarations, but only one definition is allowed.

- There shall be one and only one definition of every non-inline function or variable that is odr-used in the entire program.

- An inline function definition is required in every translation unit where it is odr-used.

- There shall be one and only one definition of a class in any translation unit where the class is used in a way that requires it to be complete.

- There can be more than one definition of any class, enumeration type, inline function with external linkage, class template, non-static function template, static data member of a class template, member function of a class template, partial template specialization in a program, as long as all of the following is true:
  - each definition consists of the same sequence of tokens (typically, appears in the same header file)
  - name lookup from within each definition finds the same entities (after overload-resolution), except that constants with internal or no linkage may refer to different objects as long as they are not ODR-used and have the same values in every definition.
  - overloaded operators, including conversion, allocation, and deallocation functions refer to the same function from each definition (unless referring to one defined within the definition)
  - the language linkage is the same (e.g. the include file isn’t inside an extern “C” block)
  - the three rules above apply to every default argument used in each definition
  - if the definition is for a class with an implicitly-declared constructor, every translation unit where it is odr-used must call the same constructor for the base and members
  - if the definition is for a template, then all these requirements apply to both names at the point of definition and dependent names at the point of instantiation

If all these requirements are satisfied, the program behaves as if there is only one definition in the entire program. Otherwise, the behavior is undefined.

### ODR-use

An object is odr-used if its address is taken, or a reference is bound to it. A function is odr-used if a function call to it is made or its address is taken.
<table>
<thead>
<tr>
<th>POD Type</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trivially Copyable Class</td>
<td>A class that:</td>
</tr>
<tr>
<td></td>
<td>• has no non-trivial default constructors</td>
</tr>
<tr>
<td></td>
<td>• has no non-trivial copy and move constructors</td>
</tr>
<tr>
<td></td>
<td>• has no non-trivial copy and move assignment operators</td>
</tr>
<tr>
<td></td>
<td>• has no virtual functions</td>
</tr>
<tr>
<td></td>
<td>• has no virtual base classes</td>
</tr>
<tr>
<td></td>
<td>• has a trivial destructor</td>
</tr>
<tr>
<td></td>
<td>• has a default constructor</td>
</tr>
<tr>
<td>Standard-Layout Class</td>
<td>A class that:</td>
</tr>
<tr>
<td></td>
<td>• has no non-static data members of type non-standard-layout class (or array of such types) or reference</td>
</tr>
<tr>
<td></td>
<td>• has no virtual functions and no virtual base classes</td>
</tr>
<tr>
<td></td>
<td>• has the same access control for all non-static data members</td>
</tr>
<tr>
<td></td>
<td>• has no non-standard-layout base classes</td>
</tr>
<tr>
<td></td>
<td>• has at most one base class subobject of any given type</td>
</tr>
<tr>
<td></td>
<td>• has all non-static data members and bit-fields in the class and its base classes first declared in the same class</td>
</tr>
<tr>
<td></td>
<td>• has no element of the set M(X) of types as a base class where M(X) is defined as follows:</td>
</tr>
<tr>
<td></td>
<td>• If X is a non-union class type, the set M(X) is empty if X has no (possibly inherited) non-static data members; otherwise, it consists of the type of the first non-static data member of X (where said member may be an anonymous union), X0, and the elements of M(X0).</td>
</tr>
<tr>
<td></td>
<td>• If X is a union type, the set M(X) is the union of all M(Ui) and the set containing all Ui, where each Ui is the type of the i-th non-static data member of X.</td>
</tr>
<tr>
<td></td>
<td>• If X is a non-class type, the set M(X) is empty.</td>
</tr>
</tbody>
</table>
| **Dataflow Anomaly** | The state of a variable at a point in a program can be described using the following terms:
- **Undefined (U):** The value of the variable is indeterminate.
- **Referenced (R):** The variable is used in some way (e.g. in an expression).
- **Defined (D):** The variable is explicitly initialized or assigned a value.

Given the above, the following dataflow anomalies can be defined:
- **UR dataflow anomaly:** Variable not assigned a value before the specified use.
- **DU dataflow anomaly:** Variable is assigned a value that is never subsequently used.
- **DD dataflow anomaly:** Variable is assigned a value twice with no intermediate use. |

| **Dead Code** | Dead code (also known as redundant code) consists of evaluated expressions whose removal would not affect the output program. |

| **Unreachable Code** | Unreachable code is code to which there is no syntactic (control flow) path, e.g. a function which is never called, either directly or indirectly. |

| **Diamond Problem** | The “diamond problem” is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C. If there is a method provided by class A, that is 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** | An interface class is a class which has following properties:
- if there are any, all methods are public pure virtual
- if there are any, all data members are public static constexpr |

| **Extended precision format** | The IEEE Standard for Floating-Point Arithmetic (IEEE 754) specifies extended precision formats, that are recommended for allowing a greater precision format than that provided by the basic formats. For an extended format the exponent range must be as great as that of the next wider basic format. For instance, 64-bit extended precision binary number must have an “exponent max” of at least 16383, which is equal to “exponent max” of 128-bit binary floating-point. The 80-bit extended format meets this requirement. |

| **Fundamental types** | C++ built-in types defined in C++ Language Standard [3] in chapter 3.9.1, e.g. char, signed char, unsigned char, int, long long int, wchar_t, bool, float, double, void, std::nullptr_t, etc. |
Scalar types

A scalar type is a type that provides built-in functionality for the addition operator without overloads. Following types are scalar types:

- integral types
- floating point types
- pointers
- scoped and unscoped enumerations
- std::nullptr_t

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 true, nullptr, etc.) or the result of calling a function whose return type is not a reference is a prvalue.

Implicitly-defined default constructor

Implicitly-defined default constructor calls default constructors of its base classes and non-static data members. It has exactly the same effect as a user-defined constructor with empty body and empty initializer list.

Implicitly-defined copy constructor

Implicitly-defined copy constructor of a class type (class or struct) performs full member-wise copy of the object’s bases and non-static data members, in their initialization order, using direct initialization.

Implicitly-defined move constructor

Implicitly-defined move constructor of a class type (class or struct) performs full member-wise move of the object’s bases and non-static members, in their initialization order, using direct initialization with an xvalue argument.

Implicitly-defined copy assignment operator

Implicitly-defined copy assignment operator of a class type (class or struct) performs full member-wise copy assignment of the object’s bases and non-static data members, in their initialization order, using built-in assignment for the scalars and copy assignment operator for class types.

Implicitly-defined move assignment operator

Implicitly-defined move assignment operator of a class type (class or struct) performs full member-wise move assignment of the object’s direct bases and immediate non-static data members, in their initialization order, using built-in assignment for the scalars, member-wise move-assignment for arrays, and move assignment operator for class types (called non-virtually).
| Implicitly-defined destructor | Implicitly-defined destructor has an empty body. After the body of the destructor is executed, the destructors for all non-static non-variant data members of the class are called, in reverse order of declaration. Then it calls destructors of all direct non-virtual base classes, in reverse order of construction, and then it calls the destructors of all virtual bases. |

Table B.1: table:acronyms