QA-MISRA

Compliance Matrices for

CWE (Common Weakness Enumeration)
SEI CERT C/C++
JSF AV C++
ISO/IEC TS 17961:2013
HIS Metrics

Release 23.04, b13183398

April 18, 2023

QA Systems GmbH
powered by AbsInt Angewandte Informatik GmbH

CONTACT:

QA Systems GmbH
support@qa-systems.com
www.qa-systems.com
www.qa-systems.de/tools/qa-misra/

Known Safety Issues:

www.absint.com/known-issues/qa-misra/23.04.md

Contents

1 Introduction

QA-MISRA is a static analyzer that checks for violations of coding guidelines such as MISRA. It supports the MISRA-C:2004, MISRA C:2012, MISRA C++:2008, AUTOSAR C++14, ISO/IEC TS 17961:2013, CERT, JSF AV C++, and CWE rule sets, as well as rules for coding style and thresholds for code metrics.

Astrée (`https://www.absint.com/astree/index.htm`) is a static code analyzer that proves the absence of runtime errors and invalid concurrent behavior in safety-critical software written or generated in C or C++. Astrée and **QA-MISRA** can be seamlessly integrated. Using **QA-MISRA** in conjunction with the sound semantic analyses offered by Astrée guarantees zero false negatives and minimizes false positives on semantical rules.

1.1 Terms and Definitions

If not stated otherwise for a specific set of guidelines, the degree of rule support is classified as follows.

fully checked A rule is *fully checked (FC)* if the checks adhere exactly to the rule text and the analysis will never miss a rule violation. For fully checked rules, absence of alarms means the tool can prove the absence of violations of this rule. False alarms may be issued.

This degree of support may be raised to *fully checked + exact (FC+E)* if the absence of false alarms can be guaranteed.

partially checked A rule is *partially checked (PC)* if the checks either check only some aspects or a (simplifying) reformulation of the rule (text) and/or the rule may miss rule violations. For partially checked rules, absence of alarms does not imply absence of rule violations. False alarms may be issued.

This degree of support may be raised to *partially checked + soundly supported (PC + S)* if activating **Astrée**'s semantic analysis underpins the rule check by issuing semantic alarms for violations of the rule and by proving the absence of violations of some aspects of the rule or if the analyzer's frontend implicitly checks some aspects of the rule.

(soundly) supported A rule is classified as *(soundly) supported (S)* if there are no dedicated checks, but an analysis run may produce evidence whether or not the rule is broken. This compliance level may require that the user provides appropriate analysis stubs.

For example, the rule "No reliance shall be placed on undefined or unspecified behavior." (MISRA-C:2004, rule 1.2) is supported by **Astrée** because **Astrée** reports undefined and unspecified behavior.

not checked A rule is *not checked (NC)* if there are no dedicated checks and checking the rule is not supported by the analyzer.

Coding guideline checks can be executed with or without runtime error analysis. Coupling **QA-MISRA** with **Astrée**'s runtime error analysis can raise the degree to which a coding guideline is supported.

The following table shows the degree of rule support for each rule assuming no coupling. Detailed information on the degree of support when coupled with **Astrée** can be found in the dedicated a[3] for C/C++ Compliance documentation.

2 Common Weakness Enumeration – CWE

The compliance matrix has been filled with one of the following values corresponding to the Match Accuracy element defined by CWE CCR:

- E = Exact
- MA = CWE-more-abstract
- MS = CWE-more-specific
- P = CWE-partial
- NC = Not-covered

In case the match accuracy for **QA-MISRA** in stand-alone analysis mode differs, it is given in parenthesis.

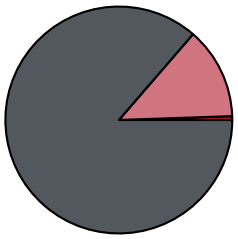
Note that all rules reported by **Astrée** are *sound*, in that if a message or alarm is relevant for a given program, then it will be reported by the tool.

Coding guideline checks can be executed with or without runtime error analysis. Coupling **QA-MISRA** with **Astrée**'s runtime error analysis can raise the degree to which a coding guideline is supported.

The following table shows the degree of rule support for each rule assuming no coupling. Detailed information on the degree of support when coupled with **Astrée** can be found in the dedicated a[3] for C/C++ Compliance documentation.

In total, 25 rules of the rule set – i. e. 14% of all 183 rules – are checked:

	All Rules
fully checked	1 (1 %)
partially checked	24 (13 %)
implicitly checkable	0 (0 %)
not checked	158 (86 %)



Common Weakness Enumeration – CWE		**Support**
14	Compiler Removal of Code to Clear Buffers	*NC*
15	External Control of System or Configuration Setting	*NC*
	Astrée provides the taint analysis that can track untrusted data	
22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	*NC*

continues on the next page…

Common Weakness Enumeration – CWE		Support
…continued		
23	Relative Path Traversal	*NC*
36	Absolute Path Traversal	*NC*
73	External Control of File Name or Path	*NC*
	Astrée provides the taint analysis that can track untrusted data	
77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	*NC*
	Astrée provides the taint analysis that can track untrusted data	
78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	*NC*
	Astrée provides the taint analysis that can track untrusted data	
79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	*NC*
	Astrée provides the taint analysis that can track untrusted data	
88	Argument Injection or Modification	*NC*
	Astrée provides the taint analysis that can track untrusted data	
89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	*NC*
	Astrée provides the taint analysis that can track untrusted data	
90	Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')	*NC*
	Astrée provides the taint analysis that can track untrusted data	
91	XML Injection (aka Blind XPath Injection)	*NC*
	Astrée provides the taint analysis that can track untrusted data	
99	Improper Control of Resource Identifiers ('Resource Injection')	*NC*
	Astrée provides the taint analysis that can track untrusted data	
114	Process Control	*NC*
117	Improper Output Neutralization for Logs	*NC*
	Astrée provides the taint analysis that can track untrusted data	
118	Improper Access of Indexable Resource	*P*

continues on the next page…

Common Weakness Enumeration – CWE		Support
…continued		
	Incorrect field dereference, Out-of-bound array access, Dereference of null or invalid pointer, Possible overflow upon derefenrece	
119	Improper Restriction of Operations within the Bounds of a Memory Buffer	*NC*
	Incorrect field dereference, Possible overflow upon dereference	
120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	*NC*
	Incorrect field dereference, Possible overflow upon dereference	
121	Stack-based Buffer Overflow	*NC*
	Incorrect field dereference, Possible overflow upon dereference	
122	Heap-based Buffer Overflow	*NC*
	Incorrect field dereference, Possible overflow upon dereference	
123	Write-what-where Condition	*P*
	Incorrect field dereference, Out-of-bound array access, Dereference of null or invalid pointer, Possible overflow upon derefenrece	
124	Buffer Underwrite	*P*
	Out-of-bound array index, Invalid dereference	
125	Out-of-bounds Read	*P*
	Incorrect field dereference, Out-of-bound array access, Dereference of null or invalid pointer, Possible overflow upon derefenrece	
126	Buffer Over-read	*P*
	Incorrect field dereference, Out-of-bound array access, Dereference of null or invalid pointer, Possible overflow upon derefenrece	
127	Buffer Under-read	*P*
	Incorrect field dereference, Out-of-bound array access, Dereference of null or invalid pointer, Possible overflow upon derefenrece	
128	Wrap-around Error	*NC*
	Overflow in arithmetic, Overflow in conversion	
129	Improper Validation of Array Index	*P*
	Incorrect field dereference, Out-of-bound array access	

continues on the next page…

Common Weakness Enumeration – CWE		Support
…continued		
130	Improper Handling of Length Parameter Inconsistency	*NC*
131	Incorrect Calculation of Buffer Size	*NC*
	Incorrect field dereference, Possible overflow upon dereference	
134	Uncontrolled Format String	*NC*
	Astrée provides the taint analysis that can track untrusted data	
170	Improper Null Termination	*NC*
176	Improper Handling of Unicode Encoding	*NC*
188	Reliance on Data/Memory Layout	*NC*
	RTE consequence of the invalid assumption	
190	Integer Overflow or Wraparound	*NC*
	Overflow in arithmetic, Overflow in conversion	
191	Integer Underflow or Wraparound	*NC*
	Overflow in arithmetic, Overflow in conversion	
193	Off-by-one Error	*NC*
	RTE consequence of the off-by-one error	
194	Unexpected Sign Extension	*NC*
	Overflow in conversion	
195	Signed to Unsigned Conversion Error	*NC*
	Overflow in conversion	
196	Unsigned to Signed Conversion Error	*NC*
	Overflow in conversion	
197	Numeric Truncation Error	*NC*
	Overflow in conversion	
226	Sensitive Information Uncleared Before Release	*NC*
240	Improper Handling of Inconsistent Structural Elements	*NC*
242	Use of Inherently Dangerous Function	*NC*
243	Creation of chroot Jail Without Changing Working Directory	*NC*

continues on the next page…

Common Weakness Enumeration – CWE		Support
…continued		
244	Improper Clearing of Heap Memory Before Release (’Heap Inspection’)	*NC*
250	Execution with Unnecessary Privileges	*NC*
252	Unchecked Return Value	*NC*
253	Incorrect Check of Function Return Value	*NC*
256	Plaintext Storage of a Password	*NC*
259	Use of Hard-coded Password	*NC*
261	Weak Cryptography for Passwords	*NC*
272	Least Privilege Violation	*NC*
285	Improper Authorization	*NC*
292	DEPRECATED (Duplicate): Trusting Self-reported DNS Name	*NC*
297	Improper Validation of Certificate with Host Mismatch	*NC*
306	Missing Authentication for Critical Function	*NC*
307	Improper Restriction of Excessive Authentication Attempts	*NC*
311	Missing Encryption of Sensitive Data	*NC*
320	Key Management Errors	*NC*
321	Use of Hard-coded Cryptographic Key	*NC*
325	Missing Required Cryptographic Step	*NC*
326	Inadequate Encryption Strength	*NC*
327	Use of a Broken or Risky Cryptographic Algorithm	*NC*
328	Reversible One-Way Hash	*NC*
	Can be checked by suitable stubs for the corresponding functions.	
329	Not Using a Random IV with CBC Mode	*NC*
330	Use of Insufficiently Random Values	*NC*
331	Insufficient Entropy	*NC*
335	PRNG Seed Error	*NC*
336	Same Seed in PRNG	*NC*
337	Predictable Seed in PRNG	*NC*

continues on the next page…

Common Weakness Enumeration – CWE		Support
…continued		
338	Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)	*NC*
350	Reliance on Reverse DNS Resolution for a Security-Critical Action	*NC*
352	Cross-Site Request Forgery (CSRF)	*NC*
359	Exposure of Private Information (’Privacy Violation’)	*NC*
362	Concurrent Execution using Shared Resource with Improper Synchronization (’Race Condition’) Read/Write data race	*NC*
364	Signal Handler Race Condition Read/Write data race	*NC*
365	Race Condition in Switch Read/Write data race	*NC*
366	Race Condition within a Thread Read/Write data race	*NC*
367	Time-of-check Time-of-use (TOCTOU) Race Condition Read/Write data race	*NC*
369	Divde By Zero Float division by zero, Integer division by zero	*NC*
377	Insecure Temporary File	*NC*
398	Indicator of Poor Code Quality Usage of sets of rule checks, such as MISRA	*NC*
400	Uncontrolled Resource Consumption (’Resource Exhaustion’)	*NC*
401	Improper Release of Memory Before Removing Last Reference (’Memory Leak’) Can be partially checked by adapting the stubs for allocation and deallocation functions.	*NC*
404	Improper Resource Shutdown or Release For OSEK, Invalid calls to OSEK system services	*NC*
411	Resource Locking Problems	*NC*

continues on the next page…

Common Weakness Enumeration – CWE		Support
…continued		
	Invalid usage of concurrency primitives	
415	Double Free	*P*
	Invalid pointer for dynamic reallocation or free	
416	Use After Free	*NC*
	Invalid dereference	
426	Untrusted Search Path	*NC*
427	Uncontrolled Search Path Element	*NC*
434	Unrestricted Upload of File with Dangerous Type	*NC*
456	Missing Initialization of a Variable	*P*
	Uninitialized variable	
457	Use of Uninitialized Variable	*P*
	Uninitialized variable	
466	Return of Pointer Value Outside of Expected Range	*NC*
467	Use of sizeof() on a Pointer Type	*P*
468	Incorrect Pointer Scaling	*P*
470	Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection')	*NC*
471	Modification of Assumed-Immutable Data (MAID)	*NC*
	Attempt to write to a constant	
474	Use of Function with Inconsistent Implementations	*NC*
475	Undefined Behavior for Input to API	*NC*
476	NULL Pointer Dereference	*NC*
	Invalid dereference	
477	Use of Obsolete Functions	*NC*
	Can be checked by suitable stubs for the corresponding functions.	
478	Missing Default Case in Switch Statement	*E*
	MISRA rule M.15.0	
481	Assigning instead of Comparing	*P*

continues on the next page…

Common Weakness Enumeration – CWE		Support
…continued		
482	Comparing instead of Assigning	*NC*
494	Download of Code Without Integrity Check	*NC*
497	Exposure of System Data to an Unauthorized Control Sphere	*NC*
	Astrée provides the taint analysis that can track untrusted data	
526	Information Exposure Through Environmental Variables	*NC*
532	Information Exposure Through Log Files	*NC*
535	Information Exposure Through Shell Error Message	*NC*
547	Use of Hard-coded, Security-relevant Constants	*NC*
558	Use of getlogin() in Multithreaded Application	*P*
	Use of bad function	
560	Use of umask() with chmod-style Argument	*NC*
561	Dead Code	*P*
562	Return of Stack Variable Address	*NC*
563	Assignment to Variable without Use ('Unused Variable')	*P*
566	Authorization Bypass Through User-Controlled SQL Primary Key	*NC*
567	Unsynchronized Access to Shared Data in a Multithreaded Context	*NC*
	Read/Write data race, Write/Write data race	
573	Improper Following of Specification by Caller	*NC*
	Specific messages for the C library and OSEK services	
587	Assignment of a Fixed Address to a Pointer	*NC*
	When that pointer is used, Incorrect dereference	
588	Attempt to Access Child of a Non-structure Pointer	*NC*
	Parsing error	
591	Sensitive Data Storage in Improperly Locked Memory	*NC*
601	URL Redirection to Untrusted Site ('Open Redirect')	*NC*
606	Unchecked Input for Loop Condition	*NC*
	Astrée provides the taint analysis that can track untrusted data	
611	Improper Restriction of XML External Entity Reference ('XXE')	*NC*

continues on the next page…

Common Weakness Enumeration – CWE		Support
…continued		
	Astrée provides the taint analysis that can track untrusted data	
615	Information Exposure Through Comments	*NC*
628	Function Call with Incorrectly Specified Arguments	*NC*
639	Authorization Bypass Through User-Controlled Key	*NC*
643	Improper Neutralization of Data within XPath Expressions ('XPath Injection')	*NC*
	Astrée provides the taint analysis that can track untrusted data	
662	Improper Synchronization	*NC*
	Read/Write data race, Write/Write data race	
663	Use of a Non-reentrant Function in a Concurrent Context	*NC*
	Read/Write data race, Write/Write data race	
665	Improper Initialization	*P*
	Use of uninitialized variable, RTE following the wrong initialization	
666	Operation on Resource in Wrong Phase of Lifetime	*NC*
	Can be checked by suitable stubs for the corresponding functions.	
667	Improper Locking	*NC*
	Read/Write data race, Write/Write data race	
672	Operation on a Resource after Expiration or Release	*NC*
	Invalid dereference, Invalid calls to OSEK system services	
676	Use of Potentially Dangerous Function	*P*
	Use of bad function	
680	Integer Overflow to Buffer Overflow	*NC*
	Overflow in arithmetic, followed by Possible overflow upon dereference	
681	Incorrect Conversion between Numeric Types	*NC*
	Overflow in conversion	
682	Incorrect Calculation	*NC*
	RTE consequences of the incorrect calculation	

continues on the next page…

Common Weakness Enumeration – CWE		Support
…continued		
685	Function Call With Incorrect Number of Arguments Function call with wrong number of arguments	*P*
686	Function Call With Incorrect Argument Type unhandled or invalid cast in gen_cast, Wrong parameter type in a function call	*P*
687	Function Call With Incorrectly Specified Argument Value	*NC*
688	Function Call With Incorrect Variable or Reference as Argument	*NC*
690	Unchecked Return Value to NULL Pointer Dereference Invalid dereference	*NC*
691	Insufficient Control Flow Management	*NC*
696	Incorrect Behavior Order	*NC*
704	Incorrect Type Conversion or Cast Overflow in conversion	*NC*
732	Incorrect Permission Assignment for Critical Resource	*NC*
733	Compiler Optimization Removal or Modification of Security-critical Code	*NC*
754	Improper Check for Unusual or Exceptional Conditions	*NC*
755	Improper Handling of Exceptional Conditions	*NC*
759	Use of a One-Way Hash without a Salt Can be checked by suitable stubs for the corresponding functions.	*NC*
760	Use of a One-Way Hash with a Predictable Salt	*NC*
761	Free of Pointer not at Start of Buffer Invalid pointer for dynamic reallocation or free	*P*
763	Release of Invalid Pointer or Reference	*NC*
764	Multiple Locks of a Critical Resource Invalid usage of concurrency primitives	*NC*
765	Multiple Unlocks of a Critical Resource Invalid usage of concurrency primitives	*NC*

continues on the next page…

Common Weakness Enumeration – CWE		Support
…continued		
767	Access to Critical Private Variable via Public Method	*NC*
	Astrée provides the taint analysis that can track untrusted data	
770	Allocation of Resources Without Limits or Throttling	*NC*
780	Use of RSA Algorithm without OAEP	*NC*
783	Operator Precedence Logic Error	*NC*
	Can be checked using the rule Misra M2012.12.1	
785	Use of Path Manipulation Function without Maximum-sized Buffer	*NC*
	message corresponding to the buffer overflow	
786	Access of Memory Location Before Start of Buffer	*NC*
	Possible overflow upon dereference	
787	Out-of-bounds Write	*NC*
	Possible overflow upon dereference	
789	Uncontrolled Memory Allocation	*NC*
	Astrée provides the taint analysis that can track untrusted data	
798	Use of Hard-coded Credentials	*NC*
805	Buffer Access with Incorrect Length Value	*NC*
	Possible overflow upon dereference	
806	Buffer Access Using Size of Source Buffer	*NC*
	Possible overflow upon dereference	
807	Reliance on Untrusted Inputs in a Security Decision	*NC*
	Astrée provides the taint analysis that can track untrusted data	
822	Untrusted Pointer Dereference	*NC*
	Astrée provides the taint analysis that can track untrusted data	
823	Use of Out-of-range Pointer Offset	*P*
	Incorrect field dereference, Out-of-bound array access, Dereference of null or invalid pointer, Possible overflow upon derefenrece	
824	Access of Uninitialized Pointer	*P*

continues on the next page…

Common Weakness Enumeration – CWE		Support
…continued		
	Invalid dereference, Uninitialized variable	
825	Expired Pointer Dereference	*NC*
	Invalid dereference	
829	Inclusion of Functionality from Untrusted Control Sphere	*NC*
831	Signal Handler Function Associated with Multiple Signals	*NC*
	Can be checked by suitable stubs for the corresponding functions.	
832	Unlock of a Resource that is not Locked	*NC*
	Invalid usage of concurrency primitives	
833	Deadlock	*NC*
	Astrée reports all potential data races and deadlocks	
835	Loop with Unreachable Exit Condition (’Infinite Loop’)	*NC*
	Loop never terminates	
862	Missing Authorization	*NC*
863	Incorrect Authorization	*NC*
908	Use of Uninitialized Resource	*P*
	Uninitialized variable	
916	Use of Password Hash With Insufficient Computational Effort	*NC*

3 SEI CERT C/C++ Coding Standard

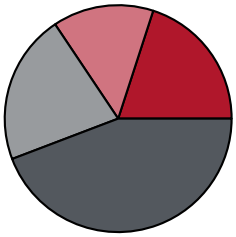
3.1 SEI CERT C Coding Standard

Coding guideline checks can be executed with or without runtime error analysis. Coupling **QA-MISRA** with **Astrée**'s runtime error analysis can raise the degree to which a coding guideline is supported.

The following table shows the degree of rule support for each rule assuming no coupling. Detailed information on the degree of support when coupled with **Astrée** can be found in the dedicated a[3] for C/C++ Compliance documentation.

In total, 103 rules of the rule set – i. e. 36% of all 285 rules – are checked:

	All Rules	Rules	Recommendations
fully checked	44 (15 %)	18 (18 %)	26 (13 %)
partially checked	47 (16 %)	29 (29 %)	18 (9 %)
implicitly checkable	12 (4 %)	6 (6 %)	6 (3 %)
not checked	182 (63 %)	46 (46 %)	136 (73 %)



Preprocessor (PRE)		**Support**
PRE.0	Prefer inline or static functions to function-like macros The exceptions (PRE00-C-EX1..5) stated in this rule are not considered and will cause false alarms.	*FC*
PRE.1	Use parentheses within macros around parameter names Exception PRE01-C-EX1 is not considered by this check and will cause false alarms.	*FC*
PRE.2	Macro replacement lists should be parenthesized Similar to MISRA-C:2004 Rule 19.4	*NC*
PRE.3	Prefer typedefs to defines for encoding non-pointer types	*NC*
PRE.4	Do not reuse a standard header file name	*NC*
PRE.5	Understand macro replacement when concatenating tokens or performing stringification	*NC*
PRE.6	Enclose header files in an inclusion guard	*FC*

continues on the next page…

Preprocessor (PRE)		Support
…continued		
PRE.7	Avoid using repeated question marks	*FC+E*
PRE.8	Guarantee that header file names are unique	*NC*
PRE.9	Do not replace secure functions with deprecated or obsolescent functions Can be checked by providing stubs for deprecated or obsolescent functions.	*NC*
PRE.10	Wrap multistatement macros in a do-while loop Similar to MISRA-C:2004 Rule 19.4	*NC*
PRE.11	Do not conclude macro definitions with a semicolon	*FC+E*
PRE.12	Do not define unsafe macros	*PC*
PRE.13	Use the Standard predefined macros to test for versions and features.	*NC*
PRE.30	Do not create a universal character name through concatenation	*FC+E*
PRE.31	Avoid side effects in arguments to unsafe macros	*PC*
PRE.32	Do not use preprocessor directives in invocations of function-like macros	*FC+E*

Declarations and Initialization (DCL)		Support
DCL.0	Const-qualify immutable objects	*PC*
DCL.1	Do not reuse variable names in subscopes MISRA C:2012 Rule 5.3	*S*
DCL.2	Use visually distinct identifiers MISRA C:2012 Dir 4.5	*NC*
DCL.3	Use a static assertion to test the value of a constant expression	*NC*
DCL.4	Do not declare more than one variable per declaration	*NC*
DCL.5	Use typedefs of non-pointer types only	*FC+E*
DCL.6	Use meaningful symbolic constants to represent literal values	*NC*
DCL.7	Include the appropriate type information in function declarators	*PC*

continues on the next page…

Declarations and Initialization (DCL)		Support
…continued		
	The type-correctness for function pointers is undecidable in general and not checked.	
DCL.8	Properly encode relationships in constant definitions	*NC*
DCL.9	Declare functions that return errno with a return type of errno_t	*NC*
DCL.10	Maintain the contract between the writer and caller of variadic functions	*NC*
	Astrée supports the implementation of stubs for variadic functions that verify the contract.	
DCL.11	Understand the type issues associated with variadic functions	*NC*
DCL.12	Implement abstract data types using opaque types	*NC*
DCL.13	Declare function parameters that are pointers to values not changed by the function as const	*FC+E*
DCL.15	Declare file-scope objects or functions that do not need external linkage as static	*FC+E*
DCL.16	Use "L", not "l", to indicate a long value	*FC+E*
DCL.17	Beware of miscompiled volatile-qualified variables	*NC*
DCL.18	Do not begin integer constants with 0 when specifying a decimal value	*FC+E*
DCL.19	Minimize the scope of variables and functions	*PC*
DCL.20	Explicitly specify void when a function accepts no arguments	*FC+E*
DCL.21	Understand the storage of compound literals	*NC*
DCL.22	Use volatile for data that cannot be cached	*NC*
DCL.23	Guarantee that mutually visible identifiers are unique	*S*
	MISRA C:2012 Rules 5.1-5	
DCL.30	Declare objects with appropriate storage durations	*PC*
DCL.31	Declare identifiers before using them	*FC+E*
DCL.36	Do not declare an identifier with conflicting linkage classifications	*PC*
DCL.37	Do not declare or define a reserved identifier	*PC*
DCL.38	Use the correct syntax when declaring a flexible array member	*NC*

continues on the next page…

Declarations and Initialization (DCL)		Support
…continued		
	Astrée reports violations of this guideline as array-out-of-bounds alarms.	
DCL.39	Avoid information leakage when passing a structure across a trust boundary	*PC*
DCL.40	Do not create incompatible declarations of the same function or object	*FC+E*
DCL.41	Do not declare variables inside a switch statement before the first case label	*FC+E*

Expressions (EXP)		Support
EXP.0	Use parentheses for precedence of operation	*NC*
	Similar to MISRA C:2012 Rule 12.1	
EXP.2	Be aware of the short-circuit behavior of the logical AND and OR operators	*FC*
EXP.3	Do not assume the size of a structure is the sum of the sizes of its members	*NC*
	Astrée reports accesses outside the bounds of allocated memory.	
EXP.5	Do not cast away a const qualification	*FC+E*
EXP.7	Do not diminish the benefits of constants by assuming their values in expressions	*NC*
EXP.8	Ensure pointer arithmetic is used correctly	*NC*
	Astrée reports potential runtime errors resulting from invalid pointer arithmetics.	
EXP.9	Use sizeof to determine the size of a type or variable	*PC*
EXP.10	Do not depend on the order of evaluation of subexpressions or the order in which side effects take place	*PC*
	File modifications are not taken into account.	
EXP.11	Do not make assumptions regarding the layout of structures with bit-fields	*NC*
	Astrée will report potential runtime errors resulting from incorrect assumptions.	

continues on the next page…

Expressions (EXP)		Support
…continued		
EXP.12	Do not ignore values returned by functions	*PC*
EXP.13	Treat relational and equality operators as if they were nonassociative	*FC+E*
EXP.14	Beware of integer promotion when performing bitwise operations on integer types smaller than int Resulting overflows or other undefined behavior is reported.	*NC*
EXP.15	Do not place a semicolon on the same line as an if, for, or while statement	*FC+E*
EXP.16	Do not compare function pointers to constant values	*PC*
EXP.19	Use braces for the body of an if, for, or while statement	*FC+E*
EXP.20	Perform explicit tests to determine success, true and false, and equality MISRA-C:2004 Rule 13.2	*NC*
EXP.30	Do not depend on the order of evaluation for side effects	*FC*
EXP.32	Do not access a volatile object through a nonvolatile reference MISRA C:2012 Rule 11.8	*S*
EXP.33	Do not read uninitialized memory	*PC*
EXP.34	Do not dereference null pointers	*NC*
EXP.35	Do not modify objects with temporary lifetime	*PC*
EXP.36	Do not cast pointers into more strictly aligned pointer types	*FC+E*
EXP.37	Call functions with the correct number and type of arguments	*PC*
EXP.39	Do not access a variable through a pointer of an incompatible type	*NC*
EXP.40	Do not modify constant objects	*PC*
EXP.42	Do not compare padding data	*PC*
EXP.43	Avoid undefined behavior when using restrict-qualified pointers Full check for MISRA C:2012 Rule 8.14 (do not use restrict)	*S*
EXP.44	Do not rely on side effects in operands to sizeof, _Alignof, or _Generic MISRA C:2012 Rule 13.6	*FC*

continues on the next page…

Expressions (EXP)		Support
…continued		
EXP.45	Do not perform assignments in selection statements	*FC+E*
EXP.46	Do not use a bitwise operator with a Boolean-like operand Subset of MISRA C:2012 Rule 10.1	*S*

Integers (INT)		Support
INT.0	Understand the data model used by your implementation(s)	*NC*
INT.1	Use rsize_t or size_t for all integer values representing the size of an object	*NC*
INT.2	Understand integer conversion rules Astrée reports all resulting overflows and keeps track of all integer values.	*NC*
INT.4	Enforce limits on integer values originating from tainted sources Supported by Astrée's taint analysis.	*NC*
INT.5	Do not use input functions to convert character data if they cannot handle all possible inputs	*NC*
INT.7	Use only explicitly signed or unsigned char type for numeric values MISRA C:2012 Rule 10.1, MISRA C:2012 Rule 10.3, MISRA C:2012 Rule 10.4	*S*
INT.8	Verify that all integer values are in range	*NC*
INT.9	Ensure enumeration constants map to unique values	*FC+E*
INT.10	Do not assume a positive remainder when using the % operator	*NC*
INT.12	Do not make assumptions about the type of a plain int bit-field when used in an expression	*FC*
INT.13	Use bitwise operators only on unsigned operands	*FC+E*
INT.14	Avoid performing bitwise and arithmetic operations on the same data	*NC*
INT.15	Use intmax_t or uintmax_t for formatted IO on programmer-defined integer types	*NC*
INT.16	Do not make assumptions about representation of signed integers	*PC*

continues on the next page…

Integers (INT)		Support
…continued		
INT.17	Define integer constants in an implementation-independent manner	*NC*
INT.18	Evaluate integer expressions in a larger size before comparing or assigning to that size Similar to M2012.10.6	*NC*
INT.30	Ensure that unsigned integer operations do not wrap The check for this rule currently does not consider sign information.	*NC*
INT.31	Ensure that integer conversions do not result in lost or misinterpreted data Related to MISRA C:2012 Rules 10.1, 10.3, 10.4, 10.6 and 10.7	*S*
INT.32	Ensure that operations on signed integers do not result in overflow The check for this rule currently does not consider sign information.	*NC*
INT.33	Ensure that division and remainder operations do not result in divide-by-zero errors	*NC*
INT.34	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	*PC*
INT.35	Use correct integer precisions Astrée reports potential overflows due to insufficient precision.	*NC*
INT.36	Converting a pointer to integer or integer to pointer	*FC+E*

Floating Point (FLP)		Support
FLP.0	Understand the limitations of floating-point numbers	*NC*
FLP.1	Take care in rearranging floating-point expressions	*NC*
FLP.2	Avoid using floating-point numbers when precise computation is needed	*PC*
FLP.3	Detect and handle floating-point errors	*NC*
FLP.4	Check floating-point inputs for exceptional values Astrée reports potential runtime errors resulting from missing checks for exceptional values.	*NC*

continues on the next page…

Floating Point (FLP)		Support
...continued		
FLP.5	Do not use denormalized numbers	*NC*
FLP.6	Convert integers to floating point for floating-point operations This rules aims to prevent truncations and overflows. All possible overflows are reported by Astrée.	*NC*
FLP.7	Cast the return value of a function that returns a floating-point type	*NC*
FLP.30	Do not use floating-point variables as loop counters	*FC+E*
FLP.32	Prevent or detect domain and range errors in math functions	*PC*
FLP.34	Ensure that floating-point conversions are within range of the new type Astrée reports potential overflows.	*NC*
FLP.36	Preserve precision when converting integral values to floating-point type Astrée keeps track of all floating point rounding errors and loss of precision and will report code defects resulting from them.	*NC*
FLP.37	Do not use object representations to compare floating-point values	*PC*

Arrays (ARR)		Support
ARR.0	Understand how arrays work	*NC*
ARR.1	Do not apply the sizeof operator to a pointer when taking the size of an array	*FC+E*
ARR.2	Explicitly specify array bounds, even if implicitly defined by an initializer	*PC*
ARR.30	Do not form or use out-of-bounds pointers or array subscripts	*PC*
ARR.32	Ensure size arguments for variable length arrays are in a valid range	*NC*
ARR.36	Do not subtract or compare two pointers that do not refer to the same array There is no warning when subtracting pointers that point to different fields of the same structure.	*PC*

continues on the next page...

Arrays (ARR)		Support
…continued		
ARR.37	Do not add or subtract an integer to a pointer to a non-array object MISRA-C:2004 Rule 17.4	*S*
ARR.38	Guarantee that library functions do not form invalid pointers	*NC*
ARR.39	Do not add or subtract a scaled integer to a pointer	*PC*

Characters and Strings (STR)		Support
STR.0	Represent characters using an appropriate type MISRA C:2012 Rule 10.1, MISRA-C:2004 Rule 6.1	*S*
STR.1	Adopt and implement a consistent plan for managing strings	*NC*
STR.2	Sanitize data passed to complex subsystems Supported by Astrée's taint analysis.	*NC*
STR.3	Do not inadvertently truncate a string	*NC*
STR.4	Use plain char for characters in the basic character set MISRA-C:2004 Rule 6.1	*S*
STR.5	Use pointers to const when referring to string literals	*FC+E*
STR.6	Do not assume that strtok() leaves the parse string unchanged	*NC*
STR.7	Use the bounds-checking interfaces for string manipulation Can be checked with appropriate analysis stubs.	*NC*
STR.8	Use managed strings for development of new string manipulation code	*NC*
STR.9	Don't assume numeric values for expressions with type plain character MISRA C:2012 Rule 10.1	*S*
STR.10	Do not concatenate different type of string literals	*FC+E*
STR.11	Do not specify the bound of a character array initialized with a string literal Astrée can detect subsequent code defects that this rule aims to prevent.	*NC*
STR.30	Do not attempt to modify string literals	*PC*

continues on the next page…

Characters and Strings (STR)		Support
...continued		
STR.31	Guarantee that storage for strings has sufficient space for character data and the null terminator This is checked by Astrée up to omitted string termination	*NC*
STR.32	Do not pass a non-null-terminated character sequence to a library function that expects a string Astrée supports the implementation of library stubs to verify this guideline.	*NC*
STR.34	Cast characters to unsigned char before converting to larger integer sizes	*FC+E*
STR.37	Arguments to character-handling functions must be representable as an unsigned char The non-standard functions isascii() and toascii() are not covered.	*PC*
STR.38	Do not confuse narrow and wide character strings and functions	*PC*

Memory Management (MEM)		Support
MEM.0	Allocate and free memory in the same module, at the same level of abstraction	*NC*
MEM.1	Store a new value in pointers immediately after free() Alarms about usage of invalid pointers	*NC*
MEM.2	Immediately cast the result of a memory allocation function call into a pointer to the allocated type	*PC*
MEM.3	Clear sensitive information stored in reusable resources	*NC*
MEM.4	Beware of zero-length allocations Stub implementations of allocation functions can verify that their size argument is greater than zero.	*NC*
MEM.5	Avoid large stack allocations	*NC*
MEM.6	Ensure that sensitive data is not written out to disk	*NC*
MEM.7	Ensure that the arguments to calloc(), when multiplied, do not wrap Stub implementation of calloc can verify this.	*NC*
MEM.10	Define and use a pointer validation function	*NC*

continues on the next page...

Memory Management (MEM)		Support
…continued		
MEM.11	Do not assume infinite heap space	*NC*
MEM.12	Consider using a goto chain when leaving a function on error when using and releasing resources	*NC*
MEM.30	Do not access freed memory Astrée reports accesses to freed dynamically allocated memory.	*NC*
MEM.31	Free dynamically allocated memory when no longer needed Can be partially checked by adapting the stubs for allocation and deallocation functions.	*NC*
MEM.33	Allocate and copy structures containing a flexible array member dynamically	*FC+E*
MEM.34	Only free memory allocated dynamically	*PC*
MEM.35	Allocate sufficient memory for an object	*PC*
MEM.36	Do not modify the alignment of objects by calling realloc()	*NC*

Input Output (FIO)		Support
FIO.1	Be careful using functions that use file names for identification	*NC*
FIO.2	Canonicalize path names originating from tainted sources	*NC*
FIO.3	Do not make assumptions about fopen() and file creation	*NC*
FIO.5	Identify files using multiple file attributes	*NC*
FIO.6	Create files with appropriate access permissions	*NC*
FIO.8	Take care when calling remove() on an open file	*NC*
FIO.9	Be careful with binary data when transferring data across systems	*NC*
FIO.10	Take care when using the rename() function	*NC*
FIO.11	Take care when specifying the mode parameter of fopen()	*NC*
FIO.13	Never push back anything other than one read character	*NC*
FIO.14	Understand the difference between text mode and binary mode with file streams	*NC*
FIO.15	Ensure that file operations are performed in a secure directory	*NC*

continues on the next page…

Input Output (FIO)		Support
…continued		
FIO.17	Do not rely on an ending null character when using fread()	*NC*
FIO.18	Never expect fwrite() to terminate the writing process at a null character	*NC*
FIO.19	Do not use fseek() and ftell() to compute the size of a regular file	*NC*
FIO.20	Avoid unintentional truncation when using fgets() or fgetws()	*NC*
FIO.21	Do not create temporary files in shared directories	*NC*
FIO.22	Close files before spawning processes	*NC*
FIO.23	Do not exit with unflushed data in stdout or stderr	*NC*
FIO.24	Do not open a file that is already open	*NC*
FIO.30	Exclude user input from format strings	*NC*
	Astrée supports the implementation of stubs and taint analysis to verify this guideline.	
FIO.32	Do not perform operations on devices that are only appropriate for files	*NC*
FIO.34	Distinguish between characters read from a file and EOF or WEOF	*NC*
FIO.37	Do not assume that fgets() or fgetws() returns a nonempty string when successful	*NC*
	Code defects stemming from returned (empty) strings will be reported.	
FIO.38	Do not copy a FILE object	*PC*
FIO.39	Do not alternately input and output from a stream without an intervening flush or positioning call	*NC*
	Can be checked by providing suitable stub implementations for the corresponding functions.	
FIO.40	Reset strings on fgets() or fgetws() failure	*NC*
FIO.41	Do not call getc(), putc(), getwc(), or putwc() with a stream argument that has side effects	*FC*
FIO.42	Close files when they are no longer needed	*NC*
	Can be checked by providing suitable stub implementations for the corresponding functions.	
FIO.44	Only use values for fsetpos() that are returned from fgetpos()	*NC*

continues on the next page…

Input Output (FIO)		Support
…continued		
FIO.45	Avoid TOCTOU race conditions while accessing files	*NC*
FIO.46	Do not access a closed file	*NC*
	Supported via stubbing and taint analysis.	
FIO.47	Use valid format strings	*NC*
	Astrée supports the implementation of stubs to verify this guideline.	

Environment (ENV)		Support
ENV.1	Do not make assumptions about the size of an environment variable	*NC*
ENV.2	Beware of multiple environment variables with the same effective name	*NC*
ENV.3	Sanitize the environment when invoking external programs	*NC*
ENV.30	Do not modify the object referenced by the return value of certain functions	*PC*
ENV.31	Do not rely on an environment pointer following an operation that may invalidate it	*NC*
ENV.32	All exit handlers must return normally	*NC*
ENV.33	Do not call system()	*FC+E*

Signals (SIG)		Support
SIG.0	Mask signals handled by noninterruptible signal handlers	*NC*
SIG.1	Understand implementation-specific details regarding signal handler persistence	*NC*
SIG.2	Avoid using signals to implement normal functionality	*NC*
SIG.30	Call only asynchronous-safe functions within signal handlers	*PC*
SIG.31	Do not access shared objects in signal handlers	*PC*
SIG.34	Do not call signal() from within interruptible signal handlers	*PC*
SIG.35	Do not return from a computational exception signal handler	*NC*

Error Handling (ERR)		Support
ERR.0	Adopt and implement a consistent and comprehensive error-handling policy	*NC*
ERR.1	Use ferror() rather than errno to check for FILE stream errors	*NC*
ERR.2	Avoid in-band error indicators	*NC*
ERR.3	Use runtime-constraint handlers when calling the bounds-checking interfaces	*NC*
ERR.4	Choose an appropriate termination strategy	*NC*
ERR.5	Application-independent code should provide error detection without dictating error handling	*NC*
ERR.6	Understand the termination behavior of assert() and abort()	*NC*
ERR.7	Prefer functions that support error checking over equivalent functions that don't	*FC+E*
ERR.30	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	*NC*
ERR.32	Do not rely on indeterminate values of errno	*NC*
ERR.33	Detect and handle standard library errors	*PC*
ERR.34	Detect errors when converting a string to a number	*NC*

Application Programming Interfaces (API)		Support
API.0	Functions should validate their parameters	*NC*
API.1	Avoid laying out strings in memory directly before sensitive data	*NC*
API.2	Functions that read or write to or from an array should take an argument to specify the source or target size	*NC*
API.3	Create consistent interfaces and capabilities across related functions	*NC*
API.4	Provide a consistent and usable error-checking mechanism	*NC*
API.5	Use conformant array parameters	*NC*
API.7	Enforce type safety	*NC*
API.8	Avoid parameter names in a function prototype	*FC+E*
API.9	Compatible values should have the same type	*NC*

continues on the next page…

Application Programming Interfaces (API)		Support
…continued		
API.10	APIs should have security options enabled by default	*NC*

Concurrency (CON)		Support
CON.1	Acquire and release synchronization primitives in the same module, at the same level of abstraction	*NC*
CON.2	Do not use volatile as a synchronization primitive	*NC*
CON.3	Ensure visibility when accessing shared variables	*NC*
CON.4	Join or detach threads even if their exit status is unimportant	*NC*
CON.5	Do not perform operations that can block while holding a lock	*NC*
CON.6	Ensure that every mutex outlives the data it protects	*NC*
CON.7	Ensure that compound operations on shared variables are atomic	*NC*
CON.8	Do not assume that a group of calls to independently atomic methods is atomic	*NC*
CON.9	Avoid the ABA problem when using lock-free algorithms	*NC*
CON.30	Clean up thread-specific storage	*NC*
CON.31	Do not destroy a mutex while it is locked	*NC*
CON.32	Prevent data races when accessing bit-fields from multiple threads Astrée reports all potential data races and deadlocks.	*NC*
CON.33	Avoid race conditions when using library functions	*NC*
CON.34	Declare objects shared between threads with appropriate storage durations	*NC*
CON.35	Avoid deadlock by locking in a predefined order Astrée reports all potential deadlocks.	*NC*
CON.36	Wrap functions that can spuriously wake up in a loop	*NC*
CON.37	Do not call signal() in a multithreaded program	*FC+E*
CON.38	Preserve thread safety and liveness when using condition variables	*NC*
CON.39	Do not join or detach a thread that was previously joined or detached	*NC*

continues on the next page…

Concurrency (CON)		Support
…continued		
CON.40	Do not refer to an atomic variable twice in an expression	*PC*
CON.41	Wrap functions that can fail spuriously in a loop	*NC*

Miscellaneous (MSC)		Support
MSC.0	Compile cleanly at high warning levels	*NC*
MSC.1	Strive for logical completeness	*PC*
MSC.4	Use comments consistently and in a readable fashion	*PC*
MSC.5	Do not manipulate time_t typed values directly	*NC*
MSC.6	Beware of compiler optimizations	*NC*
MSC.7	Detect and remove dead code	*PC*
	Astrée reports unreachable code. Rules CWE.561 and M2012.2.2 report dead code.	
MSC.9	Character encoding: Use subset of ASCII for safety	*NC*
MSC.10	Character encoding: UTF8-related issues	*NC*
MSC.11	Incorporate diagnostic tests using assertions	*NC*
MSC.12	Detect and remove code that has no effect or is never executed	*PC*
	Violations of this rule are reported for code that cannot be reached by the analyzer. Such code is definitely unreachable except if the analysis terminated prematurely because of an error. It cannot be guaranteed that all unreachable code is reported. Writes without read are not considered. Expressions are tested for side effects as a whole, thus dead subexpressions are not reported.	
MSC.13	Detect and remove unused values	*NC*
MSC.14	Do not introduce unnecessary platform dependencies	*NC*
MSC.15	Do not depend on undefined behavior	*NC*
	Astrée reports undefined behavior.	
MSC.17	Finish every set of statements associated with a case label with a break statement	*FC+E*
	Exceptions MSC17-EX1 and MSC17-EX2 are not considered and will cause alarms.	

continues on the next page…

Miscellaneous (MSC)		Support
…continued		
MSC.18	Be careful while handling sensitive data, such as passwords, in program code	*NC*
MSC.19	For functions that return an array, prefer returning an empty array over a null value	*NC*
MSC.20	Do not use a switch statement to transfer control into a complex block	*FC+E*
MSC.21	Use robust loop termination conditions Astrée reports potential infinite loops.	*NC*
MSC.22	Use the setjmp(), longjmp() facility securely	*NC*
MSC.23	Beware of vendor-specific library and language differences Astrée reports non-standard language elements.	*NC*
MSC.24	Do not use deprecated or obsolescent functions	*PC*
MSC.30	Do not use the rand() function for generating pseudorandom numbers	*FC+E*
MSC.32	Properly seed pseudorandom number generators Can be checked by suitable stubs for the corresponding library functions.	*NC*
MSC.33	Do not pass invalid data to the asctime() function Can be checked by a suitable stub for asctime().	*NC*
MSC.37	Ensure that control never reaches the end of a non-void function	*FC*
MSC.38	Do not treat a predefined identifier as an object if it might only be implemented as a macro	*S*
MSC.39	Do not call va_arg() on a va_list that has an indeterminate value	*NC*
MSC.40	Do not violate constraints The C frontend rejects in large part constraint violations.	*PC*

POSIX (POS)		Support
POS.1	Check for the existence of links when dealing with files	*NC*
POS.2	Follow the principle of least privilege	*NC*
POS.4	Avoid using PTHREAD_MUTEX_NORMAL type mutex locks	*NC*

continues on the next page…

POSIX (POS)		Support
…continued		
POS.5	Limit access to files by creating a jail	*NC*

Microsoft Windows (WIN)		Support
WIN.0	Be specific when dynamically loading libraries	*NC*
WIN.1	Do not forcibly terminate execution	*PC*
WIN.2	Restrict privileges when spawning child processes	*NC*
WIN.3	Understand HANDLE inheritance	*NC*
WIN.4	Consider encrypting function pointers	*NC*

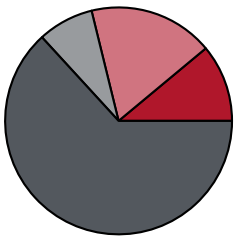
3.2 SEI CERT C++ Coding Standard

Coding guideline checks can be executed with or without runtime error analysis. Coupling **QA-MISRA** with **Astrée**'s runtime error analysis can raise the degree to which a coding guideline is supported.

The following table shows the degree of rule support for each rule assuming no coupling. Detailed information on the degree of support when coupled with **Astrée** can be found in the dedicated a[3] for C/C++ Compliance documentation.

In total, 47 rules of the rule set – i. e. 28% of all 163 rules – are checked:

	All Rules
fully checked	18 (11 %)
partially checked	29 (17 %)
implicitly checkable	0 (0 %)
not checked	116 (71 %)



Preprocessor (PRE)		Support
PRE.30C	Do not create a universal character name through concatenation	*NC*
PRE.31C	Avoid side effects in arguments to unsafe macros	*NC*
PRE.32C	Do not use preprocessor directives in invocations of function-like macros	*NC*

Declarations and Initialization (DCL)		Support
DCL.30C	Declare objects with appropriate storage durations	*PC*
DCL.39C	Avoid information leakage in structure padding	*NC*
DCL.40C	Do not create incompatible declarations of the same function or object	*PC*
DCL.50	Do not define a C-style variadic function	*FC+E*
DCL.51	Do not declare or define a reserved identifier	*PC*
DCL.52	Never qualify a reference type with const or volatile	*NC*
DCL.53	Do not write syntactically ambiguous declarations	*NC*
DCL.54	Overload allocation and deallocation functions as a pair in the same scope	*PC*
DCL.55	Avoid information leakage when passing a class object across a trust boundary	*NC*
DCL.56	Avoid cycles during initialization of static objects	*NC*
DCL.57	Do not let exceptions escape from destructors or deallocation functions	*FC+E*
DCL.58	Do not modify the standard namespaces	*NC*
DCL.59	Do not define an unnamed namespace in a header file	*FC+E*
DCL.60	Obey the one-definition rule	*PC*

Expressions (EXP)		Support
EXP.34C	Do not dereference null pointers	*PC*
EXP.35C	Do not modify objects with temporary lifetime	*NC*
EXP.36C	Do not cast pointers into more strictly aligned pointer types	*NC*
EXP.37C	Call functions with the correct number and type of arguments	*NC*
EXP.39C	Do not access a variable through a pointer of an incompatible type	*NC*
EXP.42C	Do not compare padding data	*NC*
EXP.45C	Do not perform assignments in selection statements	*NC*
EXP.46C	Do not use a bitwise operator with a Boolean-like operand	*NC*
EXP.47C	Do not call va_arg with an argument of the incorrect type	*NC*

continues on the next page…

Expressions (EXP)		Support
…continued		
EXP.50	Do not depend on the order of evaluation for side effects	*NC*
EXP.51	Do not delete an array through a pointer of the incorrect type	*NC*
EXP.52	Do not rely on side effects in unevaluated operands	*PC*
EXP.53	Do not read uninitialized memory	*PC*
EXP.54	Do not access an object outside of its lifetime	*PC*
EXP.55	Do not access a cv-qualified object through a cv-unqualified type	*PC*
EXP.56	Do not call a function with a mismatched language linkage	*NC*
EXP.57	Do not cast or delete pointers to incomplete classes	*FC+E*
EXP.58	Pass an object of the correct type to va_start	*NC*
EXP.59	Use offsetof() on valid types and members	*NC*
EXP.60	Do not pass a nonstandard-layout type object across execution boundaries	*NC*
EXP.61	A lambda object must not outlive any of its reference captured objects	*NC*
EXP.62	Do not access the bits of an object representation that are not part of the object's value representation	*NC*
EXP.63	Do not rely on the value of a moved-from object	*NC*

Integers (INT)		Support
INT.30C	Ensure that unsigned integer operations do not wrap	*NC*
INT.31C	Ensure that integer conversions do not result in lost or misinterpreted data	*NC*
INT.32C	Ensure that operations on signed integers do not result in overflow	*NC*
INT.33C	Ensure that division and remainder operations do not result in divide-by-zero errors	*PC*
INT.34C	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	*NC*
INT.35C	Use correct integer precisions	*NC*

continues on the next page…

Integers (INT)		Support
…continued		
INT.36C	Converting a pointer to integer or integer to pointer	*NC*
INT.50	Do not cast to an out-of-range enumeration value	*PC*

Floating Point (FLP)		Support
FLP.30C	Do not use floating-point variables as loop counters	*NC*
FLP.32C	Prevent or detect domain and range errors in math functions	*NC*
FLP.34C	Ensure that floating-point conversions are within range of the new type	*NC*
FLP.36C	Preserve precision when converting integral values to floating-point type	*NC*
FLP.37C	Do not use object representations to compare floating-point values	*NC*

Arrays (ARR)		Support
ARR.30C	Do not form or use out-of-bounds pointers or array subscripts	*PC*
ARR.37C	Do not add or subtract an integer to a pointer to a non-array object	*NC*
ARR.38C	Guarantee that library functions do not form invalid pointers	*NC*
ARR.39C	Do not add or subtract a scaled integer to a pointer	*NC*

Containers (CTR)		Support
CTR.50	Guarantee that container indices and iterators are within the valid range	*NC*
CTR.51	Use valid references, pointers, and iterators to reference elements of a container	*NC*
CTR.52	Guarantee that library functions do not overflow	*NC*
CTR.53	Use valid iterator ranges	*NC*
CTR.54	Do not subtract iterators that do not refer to the same container	*NC*

continues on the next page…

Containers (CTR)		Support
…continued		
CTR.55	Do not use an additive operator on an iterator if the result would overflow	*NC*
CTR.56	Do not use pointer arithmetic on polymorphic objects	*NC*
CTR.57	Provide a valid ordering predicate	*NC*
CTR.58	Predicate function objects should not be mutable	*NC*

Characters and Strings (STR)		Support
STR.30C	Do not attempt to modify string literals	*NC*
STR.31C	Guarantee that storage for strings has sufficient space for character data and the null terminator	*NC*
STR.32C	Do not pass a non-null-terminated character sequence to a library function that expects a string	*NC*
STR.34C	Cast characters to unsigned char before converting to larger integer sizes	*NC*
STR.37C	Arguments to character-handling functions must be representable as an unsigned char	*NC*
STR.38C	Do not confuse narrow and wide character strings and functions	*NC*
STR.50	Guarantee that storage for strings has sufficient space for character data and the null terminator	*PC*
STR.51	Do not attempt to create a std::string from a null pointer	*NC*
STR.52	Use valid references, pointers, and iterators to reference elements of a basic_string	*NC*
STR.53	Range check element access	*NC*

Memory Management (MEM)		Support
MEM.30C	Do not access freed memory	*PC*
MEM.31C	Free dynamically allocated memory when no longer needed	*PC*
MEM.34C	Only free memory allocated dynamically	*NC*
MEM.35C	Allocate sufficient memory for an object	*NC*

continues on the next page…

Memory Management (MEM)		Support
…continued		
MEM.36C	Do not modify the alignment of objects by calling realloc()	*NC*
MEM.50	Do not access freed memory	*PC*
MEM.51	Properly deallocate dynamically allocated resources	*PC*
MEM.52	Detect and handle memory allocation errors	*NC*
MEM.53	Explicitly construct and destruct objects when manually managing object lifetime	*NC*
MEM.54	Provide placement new with properly aligned pointers to sufficient storage capacity	*PC*
MEM.55	Honor replacement dynamic storage management requirements	*PC*
MEM.56	Do not store an already-owned pointer value in an unrelated smart pointer	*NC*
MEM.57	Avoid using default operator new for over-aligned types	*FC*

Input Output (FIO)		Support
FIO.30C	Exclude user input from format strings	*NC*
FIO.32C	Do not perform operations on devices that are only appropriate for files	*NC*
FIO.34C	Distinguish between characters read from a file and EOF or WEOF	*NC*
FIO.37C	Do not assume that fgets() or fgetws() returns a nonempty string when successful	*NC*
FIO.38C	Do not copy a FILE object	*NC*
FIO.39C	Do not alternately input and output from a stream without an intervening flush or positioning call	*NC*
FIO.40C	Reset strings on fgets() or fgetws() failure	*NC*
FIO.41C	Do not call getc(), putc(), getwc(), or putwc() with a stream argument that has side effects	*NC*
FIO.42C	Close files when they are no longer needed	*NC*
FIO.44C	Only use values for fsetpos() that are returned from fgetpos()	*NC*
FIO.45C	Avoid TOCTOU race conditions while accessing files	*NC*
FIO.46C	Do not access a closed file	*NC*

continues on the next page…

Input Output (FIO)		Support
…continued		
FIO.47C	Use valid format strings	*NC*
FIO.50	Do not alternately input and output from a file stream without an intervening positioning call	*NC*
FIO.51	Close files when they are no longer needed	*NC*

Environment (ENV)		Support
ENV.30C	Do not modify the object referenced by the return value of certain functions	*NC*
ENV.31C	Do not rely on an environment pointer following an operation that may invalidate it	*NC*
ENV.32C	All exit handlers must return normally	*NC*
ENV.33C	Do not call system()	*FC*
ENV.34C	Do not store pointers returned by certain functions	*NC*

Signals (SIG)		Support
SIG.31C	Do not access shared objects in signal handlers	*NC*
SIG.34C	Do not call signal() from within interruptible signal handlers	*NC*
SIG.35C	Do not return from a computational exception signal handler	*NC*

Exceptions and Error Handling (ERR)		Support
ERR.30C	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	*NC*
ERR.32C	Do not rely on indeterminate values of errno	*NC*
ERR.33C	Detect and handle standard library errors	*PC*
ERR.34C	Detect errors when converting a string to a number	*NC*
ERR.50	Do not abruptly terminate the program	*PC*
ERR.51	Handle all exceptions	*PC*
ERR.52	Do not use setjmp() or longjmp()	*FC+E*

continues on the next page…

Exceptions and Error Handling (ERR)		Support
…continued		
ERR.53	Do not reference base classes or class data members in a constructor or destructor function-try-block handler	*FC+E*
ERR.54	Catch handlers should order their parameter types from most derived to least derived	*FC+E*
ERR.55	Honor exception specifications	*PC*
ERR.56	Guarantee exception safety	*NC*
ERR.57	Do not leak resources when handling exceptions	*NC*
ERR.58	Handle all exceptions thrown before main() begins executing	*PC*
ERR.59	Do not throw an exception across execution boundaries	*NC*
ERR.60	Exception objects must be nothrow copy constructible	*NC*
ERR.61	Catch exceptions by lvalue reference	*FC+E*
ERR.62	Detect errors when converting a string to a number	*NC*

Object Oriented Programming (OOP)		Support
OOP.50	Do not invoke virtual functions from constructors or destructors	*FC+E*
OOP.51	Do not slice derived objects	*NC*
OOP.52	Do not delete a polymorphic object without a virtual destructor	*PC*
OOP.53	Write constructor member initializers in the canonical order	*FC+E*
OOP.54	Gracefully handle self-copy assignment	*NC*
OOP.55	Do not use pointer-to-member operators to access nonexistent members	*NC*
OOP.56	Honor replacement handler requirements	*NC*
OOP.57	Prefer special member functions and overloaded operators to C Standard Library functions	*PC*
OOP.58	Copy operations must not mutate the source object	*NC*

Concurrency (CON)		Support
CON.33C	Avoid race conditions when using library functions	*NC*

continues on the next page…

Concurrency (CON)		Support
…continued		
CON.37C	Do not call signal() in a multithreaded program	*FC+E*
CON.40C	Do not refer to an atomic variable twice in an expression	*NC*
CON.41C	Wrap functions that can fail spuriously in a loop	*NC*
CON.43C	Do not allow data races in multithreaded code	*NC*
CON.50	Do not destroy a mutex while it is locked	*NC*
CON.51	Ensure actively held locks are released on exceptional conditions	*NC*
CON.52	Prevent data races when accessing bit-fields from multiple threads	*NC*
CON.53	Avoid deadlock by locking in a predefined order	*NC*
CON.54	Wrap functions that can spuriously wake up in a loop	*NC*
CON.55	Preserve thread safety and liveness when using condition variables	*NC*
CON.56	Do not speculatively lock a non-recursive mutex that is already owned by the calling thread	*NC*

Miscellaneous (MSC)		Support
MSC.30C	Do not use the rand() function for generating pseudorandom numbers	*FC+E*
MSC.32C	Properly seed pseudorandom number generators	*PC*
MSC.33C	Do not pass invalid data to the asctime() function	*NC*
MSC.37C	Ensure that control never reaches the end of a non-void function	*FC+E*
MSC.38C	Do not treat a predefined identifier as an object if it might only be implemented as a macro	*NC*
MSC.39C	Do not call va_arg() on a va_list that has an indeterminate value	*NC*
MSC.40C	Do not violate constraints	*NC*
MSC.41C	Never hard code sensitive information	*NC*
MSC.50	Do not use std::rand() for generating pseudorandom numbers	*FC+E*
MSC.51	Ensure your random number generator is properly seeded	*PC*
MSC.52	Value-returning functions must return a value from all exit paths	*FC+E*

continues on the next page…

Miscellaneous (MSC)		Support
…continued		
MSC.53	Do not return from a function declared [[noreturn]]	*FC*
MSC.54	A signal handler must be a plain old function	*NC*

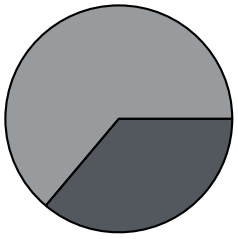
4 JSF AV C++

Coding guideline checks can be executed with or without runtime error analysis. Coupling **QA-MISRA** with **Astrée**'s runtime error analysis can raise the degree to which a coding guideline is supported.

The following table shows the degree of rule support for each rule assuming no coupling. Detailed information on the degree of support when coupled with **Astrée** can be found in the dedicated a[3] for C/C++ Compliance documentation.

In total, 145 rules of the rule set – i. e. 64% of all 227 rules – are checked:

	All Rules
fully checked	0 (0 %)
partially checked	0 (0 %)
implicitly checkable	145 (64 %)
not checked	82 (36 %)



Code Size and Complexity		**Support**
1	Any one function (or method) will contain no more than 200 logical source lines of code (L-SLOCs).	*S*
2	There shall not be any self-modifying code.	*NC*
3	All functions shall have a cyclomatic complexity number of 20 or less.	*S*

Breaking Rules		**Support**
4	To break a **should** rule, the following approval must be received by the developer: • approval from the software engineering lead (obtained by the unit approval in the developmental CM tool)	*NC*

continues on the next page…

Breaking Rules		Support
...continued		
5	To break a **will** or a **shall** rule, the following approval must be received by the developer: • approval from the software engineering lead (obtained by the unit approval in the developmental CM tool) • approval from the software product manager (obtained by the unit approval in the developmental CM tool)	*NC*
6	Each deviation from a **shall** rule shall be documented in the file that contains the deviation. Deviations from this rule shall not be allowed, AV Rule 5 notwithstanding.	*NC*
7	Approval will not be required for a deviation from a **shall** or **will** rule that complies with an exception specified by that rule.	*NC*

Language		Support
8	All code shall conform to ISO/IEC 14882:2002(E) standard C++.	*S*

Character Sets		Support
9	Only those characters specified in the C++ basic source character set will be used.	*S*
10	Values of character types will be restricted to a defined and documented subset of ISO 10646-1.	*S*
11	Trigraphs will not be used.	*S*
12	The following digraphs will not be used:	*S*
13	Multi-byte characters and wide string literals will not be used.	*S*
14	Literal suffixes shall use uppercase rather than lowercase letters.	*S*

Run-time Checks		Support
15	Provision shall be made for run-time checking (defensive programming).	*S*

Libraries		Support
16	Only DO-178B level A certifiable or SEAL 1 C/C++ libraries shall be used with safety-critical (i.e. SEAL 1) code.	*S*

Standard Libraries		Support
17	The error indicator `errno` shall not be used.	*S*
18	The macro `offsetof`, in library `<stddef.h>`, shall not be used.	*S*
19	`<locale.h>` and the `setlocale` function shall not be used.	*S*
20	The `setjmp` macro and the `longjmp` function shall not be used.	*S*
21	The signal handling facilities of `<signal.h>` shall not be used.	*S*
22	The input/output library `<stdio.h>` shall not be used.	*S*
23	The library functions `atof`, `atoi` and `atol` from library `<stdlib.h>` shall not be used.	*S*
24	The library functions `abort`, `exit`, `getenv` and `system` from library `<stdlib.h>` shall not be used.	*S*
25	The time handling functions of library `<time.h>` shall not be used.	*S*

Pre-Processing Directives		Support
26	Only the following pre-processor directives shall be used: 1. `#ifndef` 2. `#define` 3. `#endif` 4. `#include`	*S*
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.	*S*
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.	*S*

continues on the next page…

Pre-Processing Directives		Support
...continued		
29	The `#define` pre-processor directive shall not be used to create inline macros. Inline functions shall be used instead.	*S*
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.	*S*
31	The `#define` pre-processor directive will only be used as part of the technique to prevent multiple inclusions of the same header file.	*S*
32	The `#include` pre-processor directive will only be used to include header (*.h) files.	*S*

Header Files		Support
33	The `#include` directive shall use the `<filename.h>` notation to include header files.	*S*
34	Header files should contain logically related declarations only.	*NC*
35	A header file will contain a mechanism that prevents multiple inclusions of itself.	*S*
36	Compilation dependencies should be minimized when possible.	*S*
37	Header (include) files should include only those header files that are required for them to successfully compile. Files that are only used by the associated .cpp file should be placed in the .cpp file - not the .h file.	*NC*
38	Declarations of classes that are only accessed via pointers (*) or references (&) should be supplied by forward headers that contain only forward declarations.	*NC*
39	Header files (*.h) will not contain non-const variable definitions or function definitions.	*S*

Implementation Files		Support
40	Every implementation file shall include the header files that uniquely define the inline functions, types, and templates used.	*S*

Style		Support
41	Source lines will be kept to a length of 120 characters or less.	*NC*
42	Each expression-statement will be on a separate line.	*S*
43	Tabs should be avoided.	*NC*
44	All indentations will be at least two spaces and be consistent within the same source file.	*NC*

Naming Identifiers		Support
45	All words in an identifier will be separated by the '_' character.	*NC*
46	User-specified identifiers (internal and external) will not rely on significance of more than 64 characters.	*NC*
47	Identifiers will not begin with the underscore character '_'.	*NC*
48	Identifiers will not differ by: • Only a mixture of case • The presence/absence of the underscore character • The interchange of the letter 'O', with the number '0' or the letter 'D' • The interchange of the letter 'I', with the number '1' or the letter 'l' • The interchange of the letter 'S' with the number '5' • The interchange of the letter 'Z' with the number '2' • The interchange of the letter 'n' with the letter 'h'.	*S*
49	All acronyms in an identifier will be composed of uppercase letters	*NC*

Naming Classes, Structures, Enumerated types and typedefs		Support
50	The first word of the name of a class, structure, namespace, enumeration, or type created with `typedef` will begin with an uppercase letter. All others letters will be lowercase.	*NC*

Naming Functions, Variables and Parameters		Support
51	All letters contained in function and variable names will be composed entirely of lowercase letters.	*NC*

Naming Constants and Enumerators		Support
52	Identifiers for constant and enumerator values shall be lowercase.	*NC*

Naming Files		Support
53	Header files will always have a file name extension of ".h".	*S*
53.1	The following character sequences shall not appear in header file names: ’, /*, //, or ".	*S*
54	Implementation files will always have a file name extension of ".cpp".	*S*
55	The name of a header file should reflect the logical entity for which it provides declarations.	*NC*
56	The name of an implementation file should reflect the logical entity for which it provides definitions and have a ".cpp" extension (this name will normally be identical to the header file that provides the corresponding declarations.)	*NC*

Classes		Support
57	The public, protected, and private sections of a class will be declared in that order (the public section is declared before the protected section which is declared before the private section).	*NC*

Functions		Support
58	When declaring and defining functions with more than two parameters, the leading parenthesis and the first argument will be written on the same line as the function name. Each additional argument will be written on a separate line (with the closing parenthesis directly after the last argument).	*NC*

Blocks		Support
59	The statements forming the body of an `if`, `else if`, `else`, `while`, `do...while` or `for` statement shall always be enclosed in braces, even if the braces form an empty block.	*S*
60	Braces ("{}") which enclose a block will be placed in the same column, on separate lines directly before and after the block.	*NC*
61	Braces ("{}") which enclose a block will have nothing else on the line except comments (if necessary).	*NC*

Pointers and References		Support
62	The dereference operator '' and the address-of operator '&' will be directly connected with the type-specifier.	*NC*

Miscellaneous		Support
63	Spaces will not be used around '.' or '->', nor between unary operators and operands.	*NC*

Class Interfaces		Support
64	A class interface should be complete and minimal.	*NC*

Considerations Regarding Access Rights		Support
65	A structure should be used to model an entity that does not require an invariant.	*S*
66	A class should be used to model an entity that maintains an invariant.	*S*
67	Public and protected data should only be used in structs—not classes.	*S*

Member Functions		Support
68	Unneeded implicitly generated member functions shall be explicitly disallowed.	*S*

const Member Functions		Support
69	A member function that does not affect the state of an object (its instance variables) will be declared `const`.	*S*

Friends		Support
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.	*S*

Object Lifetime		Support
70.1	An object shall not be improperly used before its lifetime begins or after its lifetime ends.	*S*
71	Calls to an externally visible operation of an object, other than its constructors, shall not be allowed until the object has been fully initialized.	*S*
71.1	A class's virtual functions shall not be invoked from its destructor or any of its constructors.	*S*
72	The invariant for a class should be: • a part of the postcondition of every class constructor, • a part of the precondition of the class destructor (if any), • a part of the precondition and postcondition of every other publicly accessible operation.	*S*
73	Unnecessary default constructors shall not be defined.	*NC*
74	Initialization of nonstatic class members will be performed through the member initialization list rather than through assignment in the body of a constructor.	*S*
75	Members of the initialization list shall be listed in the order in which they are declared in the class.	*S*
76	A copy constructor and an assignment operator shall be declared for classes that contain pointers to data items or nontrivial destructors.	*NC*

continues on the next page…

Object Lifetime		Support
...continued		
77	A copy constructor shall copy all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).	*NC*
77.1	The definition of a member function shall not contain default arguments that produce a signature identical to that of the implicitly-declared copy constructor for the corresponding class/structure.	*NC*

Destructors		Support
78	All base classes with a virtual function shall define a virtual destructor.	*S*
79	All resources acquired by a class shall be released by the class's destructor.	*NC*

Assignment Operators		Support
80	The default copy and assignment operators will be used for classes when those operators offer reasonable semantics.	*S*
81	The assignment operator shall handle self-assignment correctly.	*NC*
82	An assignment operator shall return a reference to `this`.	*S*
83	An assignment operator shall assign all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).	*NC*

Operator Overloading		Support
84	Operator overloading will be used sparingly and in a conventional manner.	*S*
85	When two operators are opposites (such as == and !=), both will be defined and one will be defined in terms of the other.	*S*

Inheritance		Support
86	Concrete types should be used to represent simple independent concepts.	*NC*
87	Hierarchies should be based on abstract classes.	*NC*
88	Multiple inheritance shall only be allowed in the following restricted form: n interfaces plus m private implementations, plus at most one protected implementation.	*S*
88.1	A stateful virtual base shall be explicitly declared in each derived class that accesses it.	*NC*
89	A base class shall not be both virtual and non-virtual in the same hierarchy.	*S*
90	Heavily used interfaces should be minimal, general and abstract.	*NC*
91	Public inheritance will be used to implement "is-a" relationships.	*NC*
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: • Preconditions of derived methods must be at least as weak as the preconditions of the methods they override. • Postconditions of derived methods must be at least as strong as the postconditions of the methods they override.	*NC*
93	"has-a" or "is-implemented-in-terms-of" relationships will be modeled through membership or non-public inheritance.	*NC*
94	An inherited nonvirtual function shall not be redefined in a derived class.	*S*
95	An inherited default parameter shall never be redefined.	*S*
96	Arrays shall not be treated polymorphically.	*S*
97	Arrays shall not be used in interfaces. Instead, the `Array` class should be used.	*S*

Virtual Member Functions		Support
97.1	Neither operand of an equality operator (== or !=) shall be a pointer to a virtual member function.	*S*

Namespaces		Support
98	Every nonlocal name, except main(), should be placed in some namespace.	*S*
99	Namespaces will not be nested more than two levels deep.	*NC*
100	Elements from a namespace should be selected as follows: • using declaration or explicit qualification for few (approximately five) names, • using directive for many names.	*NC*

Templates		Support
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.	*NC*
102	Template tests shall be created to cover all actual template instantiations.	*NC*
103	Constraint checks should be applied to template arguments.	*NC*
104	A template specialization shall be declared before its use.	*S*
105	A template definition's dependence on its instantiation contexts should be minimized.	*NC*
106	Specializations for pointer types should be made where appropriate.	*NC*

Function Declaration, Definition and Arguments		Support
107	Functions shall always be declared at file scope.	*S*
108	Functions with variable numbers of arguments shall not be used.	*S*
109	A function definition should not be placed in a class specification unless the function is intended to be inlined.	*NC*
110	Functions with more than 7 arguments will not be used.	*S*

continues on the next page…

Function Declaration, Definition and Arguments		Support
...continued		
111	A function shall not return a pointer or reference to a non-static local object.	*S*
112	Function return values should not obscure resource ownership.	*S*

Return Types and Values		Support
113	Functions will have a single exit point.	*S*
114	All exit points of value-returning functions shall be through return statements.	*S*
115	If a function returns error information, then that error information will be tested.	*S*

Function Parameters (Value, Pointer or Reference)		Support
116	Small, concrete-type arguments (two or three words in size) should be passed by value if changes made to formal parameters should not be reflected in the calling function.	*S*
117	Arguments should be passed by reference if NULL values are not possible.	*NC*
117.1	An object should be passed as const T& if the function should not change the value of the object.	*S*
117.2	An object should be passed as T& if the function may change the value of the object.	*NC*
118	Arguments should be passed via pointers if NULL values are possible.	*NC*
118.1	An object should be passed as `const T` if its value should not be modified.	*NC*
118.2	An object should be passed as T if its value may be modified.	*NC*

Function Invocation		Support
119	Functions shall not call themselves, either directly or indirectly (i.e. recursion shall not be allowed).	*S*

Function Overloading		Support
120	Overloaded operations or methods should form families that use the same semantics, share the same name, have the same purpose, and that are differentiated by formal parameters.	*NC*

Inline Functions		Support
121	Only functions with 1 or 2 statements should be considered candidates for inline functions.	*NC*
122	Trivial accessor and mutator functions should be inlined.	*S*
123	The number of accessor and mutator functions should be minimized.	*NC*
124	Trivial forwarding functions should be inlined.	*NC*

Temporary Objects		Support
125	Unnecessary temporary objects should be avoided.	*NC*

Comments		Support
126	Only valid C++ style comments (//) shall be used.	*S*
127	Code that is not used (commented out) shall be deleted.	*NC*
128	Comments that document actions or sources (e.g. tables, figures, paragraphs, etc.) outside of the file being documented will not be allowed.	*S*
129	Comments in header files should describe the externally visible behavior of the functions or classes being documented.	*S*
130	The purpose of every line of executable code should be explained by a comment, although one comment may describe more than one line of code.	*S*
131	One should avoid stating in comments what is better stated in code (i.e. do not simply repeat what is in the code).	*S*
132	Each variable declaration, typedef, enumeration value, and structure member will be commented.	*S*

continues on the next page…

Comments		Support
...continued		
133	Every source file will be documented with an introductory comment that provides information on the file name, its contents, and any program-required information (e.g. legal statements, copyright information, etc).	*S*
134	Assumptions (limitations) made by functions should be documented in the function's preamble.	*S*

Declarations and Definitions		Support
135	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.	*S*
136	Declarations should be at the smallest feasible scope.	*NC*
137	All declarations at file scope should be static where possible.	*S*
138	Identifiers shall not simultaneously have both internal and external linkage in the same translation unit.	*S*
139	External objects will not be declared in more than one file.	*S*
140	The `register` storage class specifier shall not be used.	*S*
141	A class, structure, or enumeration will not be declared in the definition of its type.	*S*

Initialization		Support
142	All variables shall be initialized before use.	*S*
143	Variables will not be introduced until they can be initialized with meaningful values.	*S*
144	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.	*S*
145	In an enumerator list, the `'='` construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	*S*

Initialization		Support
146	Floating point implementations shall comply with a defined floating point standard.	*NC*
147	The underlying bit representations of floating point numbers shall not be used in any way by the programmer.	*S*
148	Enumeration types shall be used instead of integer types (and constants) to select from a limited series of choices.	*NC*

Constants		Support
149	Octal constants (other than zero) shall not be used.	*S*
150	Hexadecimal constants will be represented using all uppercase letters.	*S*
151	Numeric values in code will not be used; symbolic values will be used instead.	*S*
151.1	A string literal shall not be modified.	*S*

Variables		Support
152	Multiple variable declarations shall not be allowed on the same line.	*S*

Unions and Bit Fields		Support
153	Unions shall not be used.	*S*
154	Bit-fields shall have explicitly unsigned integral or enumeration types only.	*S*
155	Bit-fields will not be used to pack data into a word for the sole purpose of saving space.	*S*
156	All the members of a structure (or class) shall be named and shall only be accessed via their names.	*NC*

Operators		Support
157	The right hand operand of a && or || operator shall not contain side effects.	*S*

continues on the next page…

Operators		Support
...continued		
158	The operands of a logical && or || shall be parenthesized if the operands contain binary operators.	*S*
159	Operators ||, &&, and unary & shall not be overloaded.	*S*
160	An assignment expression shall be used only as the expression in an expression statement.	*S*
162	Signed and unsigned values shall not be mixed in arithmetic or comparison operations.	*S*
163	Unsigned arithmetic shall not be used.	*NC*
164	The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the left-hand operand (inclusive).	*S*
164.1	The left-hand operand of a right-shift operator shall not have a negative value.	*NC*
165	The unary minus operator shall not be applied to an unsigned expression.	*S*
166	The `sizeof` operator will not be used on expressions that contain side effects.	*S*
167	The implementation of integer division in the chosen compiler shall be determined, documented and taken into account.	*NC*
168	The comma operator shall not be used.	*S*

Pointers & References		Support
169	Pointers to pointers should be avoided when possible.	*S*
170	More than 2 levels of pointer indirection shall not be used.	*S*

continues on the next page...

Pointers & References		Support
...continued		
171	Relational operators shall not be applied to pointer types except where both operands are of the same type and point to: • the same object, • the same function, • members of the same object, or • elements of the same array (including one past the end of the same array).	*NC*
173	The address of an object with automatic storage shall not be assigned to an object which persists after the object has ceased to exist.	*NC*
174	The null pointer shall not be de-referenced.	*NC*
175	A pointer shall not be compared to NULL or be assigned NULL; use plain 0 instead.	*S*
176	A typedef will be used to simplify program syntax when declaring function pointers.	*NC*

Type Conversions		Support
177	User-defined conversion functions should be avoided.	*S*
178	Down casting (casting from base to derived class) shall only be allowed through one of the following mechanism: • Virtual functions that act like dynamic casts (most likely useful in relatively simple cases) • Use of the visitor (or similar) pattern (most likely useful in complicated cases)	*S*
179	A pointer to a virtual base class shall not be converted to a pointer to a derived class.	*S*
180	Implicit conversions that may result in a loss of information shall not be used.	*S*
181	Redundant explicit casts will not be used.	*S*

continues on the next page...

Type Conversions		Support
...continued		
182	Type casting from any type to or from pointers shall not be used.	*S*
183	Every possible measure should be taken to avoid type casting.	*NC*
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.	*S*
185	C++ style casts (`const_cast`, `reinterpret_cast`, and `static_cast`) shall be used instead of the traditional C-style casts.	*S*

Flow Control Structures		Support
186	There shall be no unreachable code.	*S*
187	All non-null statements shall potentially have a side-effect.	*S*
188	Labels will not be used, except in `switch` statements.	*S*
189	The `goto` statement shall not be used.	*S*
190	The `continue` statement shall not be used.	*S*
191	The `break` statement shall not be used (except to terminate the cases of a `switch` statement).	*NC*
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.	*S*
193	Every non-empty `case` clause in a switch statement shall be terminated with a `break` statement.	*S*
194	All `switch` statements that do not intend to test for every enumeration value shall contain a final `default` clause.	*S*
195	A switch expression will not represent a Boolean value.	*S*
196	Every switch statement will have at least two `case`s and a potential `default`.	*S*
197	Floating point variables shall not be used as loop counters.	*S*
198	The initialization expression in a for loop will perform no actions other than to initialize the value of a single for loop parameter.	*NC*

continues on the next page...

Flow Control Structures		Support
...continued		
199	The increment expression in a for loop will perform no action other than to change a single loop parameter to the next value for the loop.	*S*
200	Null initialize or increment expressions in `for` loops will not be used; a `while` loop will be used instead.	*S*
201	Numeric variables being used within a `for` loop for iteration counting shall not be modified in the body of the loop.	*S*

Expressions		Support
202	Floating point variables shall not be tested for exact equality or inequality.	*S*
203	Evaluation of expressions shall not lead to overflow/underflow (unless required algorithmically and then should be heavily documented). Signed arithmetic overflows in `constexpr` expressions are rejected by the frontend as language error.	*S*
204	A single operation with side-effects shall only be used in the following contexts: 1. by itself 2. the right-hand side of an assignment 3. a condition 4. the only argument expression with a side-effect in a function call 5. condition of a loop 6. switch condition 7. single part of a chained operation.	*NC*
204.1	The value of an expression shall be the same under any order of evaluation that the standard permits.	*S*
205	The `volatile` keyword shall not be used unless directly interfacing with hardware.	*S*

Memory Allocation		Support
206	Allocation/deallocation from/to the free store (heap) shall not occur after initialization.	*S*
207	Unencapsulated global data will be avoided.	*NC*

Fault Handling		Support
208	C++ exceptions shall not be used (i.e. `throw`, `catch` and `try` shall not be used.)	*NC*

Data Abstraction		Support
209	The basic types of `int`, `short`, `long`, `float` and `double` shall not be used, but specific-length equivalents should be `typedef`'d accordingly for each compiler, and these type names used in the code.	*S*

Data Representation		Support
210	Algorithms shall not make assumptions concerning how data is represented in memory (e.g. big endian vs. little endian, base class subobject ordering in derived classes, nonstatic data member ordering across access specifiers, etc.)	*NC*
210.1	Algorithms shall not make assumptions concerning the order of allocation of nonstatic data members separated by an access specifier.	*NC*
211	Algorithms shall not assume that `shorts`, `ints`, `longs`, `floats`, `doubles` or `long doubles` begin at particular addresses.	*NC*

Underflow/Overflow		Support
212	Underflow or overflow functioning shall not be depended on in any special way.	*NC*

Order of Execution		Support
213	No dependence shall be placed on C++'s operator precedence rules, below arithmetic operators, in expressions.	*NC*

continues on the next page…

Order of Execution		Support
…continued		
214	Assuming that non-local static objects, in separate translation units, are initialized in a special order shall not be done.	*S*

Pointer Arithmetic		Support
215	Pointer arithmetic will not be used.	*S*

5 ISO/IEC TS 17961:2013

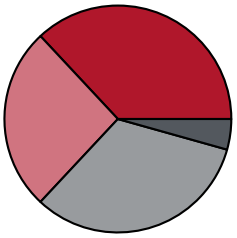
Coding guideline checks can be executed with or without runtime error analysis. Coupling **QA-MISRA** with **Astrée**'s runtime error analysis can raise the degree to which a coding guideline is supported.

The following table shows the degree of rule support for each rule assuming no coupling. Detailed information on the degree of support when coupled with **Astrée** can be found in the dedicated a[3] for C/C++ Compliance documentation.

The *ISO/IEC TS 17961:2013* rule set is based on [6].

In total, 37 rules of the rule set – i. e. 80% of all 46 rules – are checked:

	All Rules
fully checked	8 (17 %)
partially checked	17 (36 %)
implicitly checkable	12 (26 %)
not checked	9 (19 %)



ISO/IEC TS 17961:2013		Support
accfree	Accessing freed memory.	*S*
	MISRA C:2012 Rules/Directives D.4.12, 1.3, 21.3	
accsig	Accessing shared objects in signal handlers.	*PC*
addrescape	Escaping of the address of an automatic object.	*PC*
alignconv	Converting pointer values to more strictly aligned pointer types.	*FC+E*
argcomp	Calling functions with incorrect arguments.	*PC*
asyncsig	Calling functions in the C Standard Library other than abort, _Exit, and signal from within a signal handler.	*PC*
boolasgn	No assignment in conditional expressions.	*FC+E*
chreof	Using character values that are indistinguishable from EOF.	*NC*
chrsgnext	Passing arguments to character-handling functions that are not representable as unsigned char.	*PC*
	The non-standard functions isascii() and toascii() are not covered.	
dblfree	Freeing memory multiple times.	*PC*

continues on the next page…

ISO/IEC TS 17961:2013		Support
…continued		
diverr	Integer division errors.	*NC*
fileclose	Failing to close files or free dynamic memory when they are no longer needed. MISRA C:2012 Rules/Directives 22.1	*NC*
filecpy	Copying a FILE object.	*PC*
funcdecl	Declaring the same function or object in incompatible ways.	*FC+E*
insufmem	Allocating insufficient memory.	*PC*
intoflow	Overflowing signed integers.	*NC*
intptrconv	Converting a pointer to integer or integer to pointer. MISRA-C:2004 Rule 11.3, MISRA C:2012 Rules 11.4, 1.3	*S*
inverrno	Incorrectly setting and using errno.	*NC*
invfmtstr	Using invalid format strings. MISRA C:2012 Rules/Directives D.4.1, D.4.11, 1.3, 21.6	*S*
invptr	Forming or using out-of-bounds pointers or array subscripts.	*PC*
ioileave	Interleaving stream inputs and outputs without a flush or positioning call. MISRA C:2012 Rules/Directives D.4.1, 1.3, 21.6	*S*
liberr	Failing to detect and handle standard library errors.	*PC*
libmod	Modifying the string returned by getenv, localeconv, setlocale, and strerror.	*PC*
libptr	Forming invalid pointers by library function. MISRA C:2012 Rules/Directives D.4.1, D.4.11, 1.3	*S*
libuse	Using an object overwritten by getenv, localeconv, setlocale, and strerror. MISRA C:2012 Rules/Directives 21.8	*S*
nonnullcs	Passing a non-null-terminated character sequence to a library function that expects a string. MISRA C:2012 Rules/Directives D.4.1, D.4.11, 1.3	*NC*
nullref	Dereferencing an out-of-domain pointer.	*NC*

continues on the next page…

ISO/IEC TS 17961:2013		Support
…continued		
padcomp	Comparison of padding data.	*PC*
ptrcomp	Accessing an object through a pointer to an incompatible type. MISRA C:2012 Rules/Directives 1.3, 10.8, 11.2, 11.3	*S*
ptrobj	Subtracting or comparing two pointers that do not refer to the same array.	*PC*
resident	Using identifiers that are reserved for the implementation.	*FC+E*
restrict	Passing pointers into the same object as arguments to different restrict-qualified parameters. MISRA C:2012 Rules/Directives 1.3, 8.14	*S*
sigcall	Calling signal from interruptible signal handlers.	*PC*
signconv	Conversion of signed characters to wider integer types before a check for EOF. The check for this rule reports any such conversion, not checking whether the resulting value will be compared to EOF or not.	*FC*
sizeofptr	Taking the size of a pointer to determine the size of the pointed-to type.	*FC+E*
strmod	Modifying string literals.	*PC*
swtchdflt	Use of an implied default in a switch statement.	*FC+E*
syscall	Calling system.	*FC+E*
taintformatio	Using a tainted value to write to an object using a formatted input or output function. MISRA C:2012 Rules/Directives D.4.1, D.4.11, 21.6	*S*
taintnoproto	Using a tainted value as an argument to an unprototyped function pointer. Function prototypes can be enforced by MISRA-C:2004 Rule 8.1 or MISRA C:2012 Rule 8.2.	*S*
taintsink	Tainted, potentially mutilated, or out-of-domain integer values are used in a restricted sink. Astrée warns if the resulting pointer value is dereferenced.	*NC*
taintstrcpy	Tainted strings are passed to a string copying function. MISRA C:2012 Rules/Directives D.4.1, D.4.11	*NC*

continues on the next page…

ISO/IEC TS 17961:2013		Support
…continued		
uninitref	Referencing uninitialized memory.	*PC*
usrfmt	Including tainted or out-of-domain input in a format string. MISRA C:2012 Rules/Directives D.4.1, D.4.11, 1.3, 21.6	*S*
xfilepos	Using a value for fsetpos other than a value returned from fgetpos. MISRA C:2012 Rules/Directives D.4.1, D.4.11, 1.3, 21.6	*S*
xfree	Reallocating or freeing memory that was not dynamically allocated.	*PC*

6 HIS Metrics

1. Metrics with thresholds (Metriken mit Grenzwerten)		Support
COMF	Kommentardichte "COMF" (comment density)	*yes*
PATH	Anzahl der Pfade (number of paths)	*yes*
GOTO	Anzahl Sprunganweisungen (number of jump statements)	*yes*
v(G)	Zyklomatische Komplexität (cyclomatic complexity)	*yes*
CALLING	Anzahl der aufrufenden Funktionen (number of callers)	*yes*
CALLS	Anzahl der aufgerufenen Funktionen (number of callees)	*yes*
PARAM	Anzahl Funktionsparameter (number of arguments)	*yes*
STMT	Anzahl der Befehle pro Funktion (statements per function)	*yes*
LEVEL	Anzahl der Aufruflevel (call levels)	*yes*
RETURN	Anzahl der Aussprungpunkte (number of returns)	*yes*
S_i	Stabilitätsindex	*no*
	See **2. Metrics without thresholds**.	
VOCF	Sprachumfang	*yes*
NOMV	This metric counts the number of violations of the HIS Subset MISRA C 1.0.2.	*partial*
	The number of rule violations of supported and activated MISRA rules is shown by **QA-MISRA**.	
NOMVPR	This metric counts the number of violations of the HIS Subset MISRA C 1.0.2. per rule.	*partial*
	The number of rule violations of per supported and activated MISRA rules is shown by **QA-MISRA**.	
ap_cg_cycle	Anzahl der Rekursionen	*yes*

2. Metrics without thresholds (Metriken ohne Grenzwerte)		Support
S_{change}, S_{del}, and S_{new}	***QA-MISRA*** *provides a revisioning system: users can create revisions of the analysis project at different points in time which include the entire analysis state including option settings, alarms, and the source code.* ***QA-MISRA*** *can also compare different revisions and visualize the differences (e.g. in number of alarms by category, in option setting, in coverage, etc.). Since revisions also contain the full source code, also differences in the source code can be easily obtained. This information can then be used as a basis for computing these metrics.*	

Bibliography

[1] MISRA Limited. *MISRA-C:2004 Guidelines for the use of the C language in critical systems*. ISBN 978-0-9524156-4-0, October 2004.

[2] MISRA Limited. *MISRA-C:2012 Guidelines for the use of the C language in critical systems*. ISBN 978-1-906400-11-8, March 2013.

[3] QA Systems GmbH. ***QA-MISRA*** – *User Documentation*. Version 23.04, Build 13183398, April 18, 2023.

[4] AUTOSAR. Guidelines for the use of the C++14 language in critical and safety-related systems (release 19-03). Release 19-03, March 2019.

[5] Carnegie Mellon University. SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems. 2016.

[6] ISO. ISO/IEC TS 17961:2013(E): Information Technology–Programming Languages, Their Environments and System Software Interfaces–C Secure Coding Rules. November 2013.