QA-MISRA

Compliance Matrices for

MISRA C:2012
(including Amendments)

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.

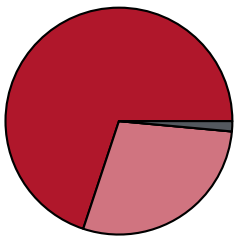
Coding guideline checks can be executed with or without runtime error analysis. Coupling **QA-MISRA** with **Astrée**'s runtime error analysis can raise the degree to which a coding guideline is supported.

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.

1.2 Coverage

In total, 203 rules of the rule set – i. e. 99% of all 206 rules – are checked:

	All Rules
fully checked	144 (70 %)
partially checked	59 (29 %)
implicitly checkable	0 (0 %)
not checked	3 (1 %)



2 MISRA C:2012

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 *MISRA C:2012* rule set is based on [2].

Directives		**Support**
D.1.1	Any implementation-defined behaviour on which the output of the program depends shall be documented and understood. The tool allows to scan the code base for specific implementation-defined behavior and allows commenting the respective findings with a justification.	*PC*
D.2.1	All source files shall compile without any compilation errors. The C frontend rejects in large part constraint violations and reports them with a diagnostic.	*PC*
D.3.1	All code shall be traceable to documented requirements.	*PC*
D.4.1	Run-time failures shall be minimized. Documentation requirements are not checked. This rule check is supported by Astrée's semantic analysis: Astrée reports runtime errors that constitute potential violations of this rule.	*PC*
D.4.2	All usage of assembly language should be documented.	*PC*
D.4.3	Assembly language shall be encapsulated and isolated.	*FC+E*
D.4.4	Sections of code should not be "commented out".	*NC*
D.4.5	Identifiers in the same name space with overlapping visibility should be typographically unambiguous.	*PC*
D.4.6	Typedefs that indicate size and signedness should be used in place of the basic numerical types.	*PC*

continues on the next page…

Directives		Support
…continued		
	Only use of typedefs is checked, not whether their names indicate size and signedness. This rule check is supported by Astrée's semantic analysis: Astrée reports overflows resulting from incorrect assumptions about the size/signedness of numerical types.	
D.4.7	If a function returns error information, then that error information shall be tested.	*PC*
	It is checked that the return value of calls to these functions is used, i.e. not immediately withdrawn. Whether there is further processing of this value, and whether this is exhaustive, is not checked.	
D.4.8	If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden.	*FC+E*
D.4.9	A function should be used in preference to a function-like macro where they are interchangeable.	*FC*
D.4.10	Precautions shall be taken in order to prevent the contents of a header file being included more than once.	*PC*
D.4.11	The validity of values passed to library functions shall be checked.	*PC*
	This rule check is supported by Astrée's semantic analysis: Astrée provides analysis stubs for library functions that raise alarms for arguments that may cause runtime errors. Without Astrée, the tool checks this rule with reduced coverage and precision.	
D.4.12	Dynamic memory allocation shall not be used.	*PC*
D.4.13	Functions which are designed to provide operations on a resource should be called in an appropriate sequence.	*PC*
	Astrée warns about illegal use of memory management functions such as double-free. File operations can be covered using appropriate stub implementations.	

A standard C environment		Support
1.1	The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits.	*PC*
	The frontend rejects in large part violations of ISO/IEC 9899:1999.	
1.2	Language extensions should not be used.	*PC*

continues on the next page…

A standard C environment		Support
...continued		
	Language extensions are rejected by the frontend as far as not supported by the tool.	
1.3	There shall be no occurrence of undefined or critical unspecified behaviour. This rule check is supported by Astrée's semantic analysis: Astrée notifies about undefined and unspecified behavior according to ISO/IEC 9899:1999.	*PC*

Unused code		Support
2.1	A project shall not contain unreachable code. 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. Without Astrée, this rule is checked with reduced coverage.	*PC*
2.2	There shall be no dead code.	*PC*
2.3	A project should not contain unused type declarations.	*FC+E*
2.4	A project should not contain unused tag declarations.	*FC+E*
2.5	A project should not contain unused macro definitions.	*FC+E*
2.6	A function should not contain unused label declarations.	*FC+E*
2.7	There should be no unused parameters in functions.	*FC+E*

Comments		Support
3.1	The character sequences /* and // shall not be used within a comment.	*FC+E*
3.2	Line-splicing shall not be used in // comments.	*FC+E*

Character sets and lexical conventions		Support
4.1	Octal and hexadecimal escape sequences shall be terminated.	*FC+E*
4.2	Trigraphs should not be used.	*FC+E*

Identifiers		Support
5.1	External identifiers shall be distinct.	*FC+E*
5.2	Identifiers declared in the same scope and name space shall be distinct.	*FC+E*
5.3	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.	*FC+E*
5.4	Macro identifiers shall be distinct.	*FC+E*
5.5	Identifiers shall be distinct from macro names.	*FC+E*
5.6	A typedef name shall be a unique identifier.	*FC+E*
5.7	A tag name shall be a unique identifier.	*FC+E*
5.8	Identifiers that define objects or functions with external linkage shall be unique.	*FC+E*
5.9	Identifiers that define objects or functions with internal linkage should be unique.	*FC+E*

Types		Support
6.1	Bit-fields shall only be declared with an appropriate type. The rule takes as argument a semicolon-separated list of basic integer types that are accepted as bitfield types. The default is 'int'.	*FC+E*
6.2	Single-bit named bit fields shall not be of a signed type.	*FC+E*

Literals and constants		Support
7.1	Octal constants shall not be used.	*FC+E*
7.2	A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type.	*FC+E*
7.3	The lowercase character "l" shall not be used in a literal suffix.	*FC+E*
7.4	A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char".	*FC+E*

Declarations and definitions		Support
8.1	Types shall be explicitly specified.	*FC+E*

continues on the next page...

Declarations and definitions		Support
...continued		
8.2	Function types shall be in prototype form with named parameters.	*FC+E*
8.3	All declarations of an object or function shall use the same names and type qualifiers.	*FC+E*
8.4	A compatible declaration shall be visible when an object or function with external linkage is defined.	*FC+E*
8.5	An external object or function shall be declared once in one and only one file.	*FC+E*
8.6	An identifier with external linkage shall have exactly one external definition.	*FC+E*
8.7	Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.	*FC+E*
8.8	The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage.	*FC+E*
8.9	An object should be defined at block scope if its identifier only appears in a single function.	*FC+E*
8.10	An inline function shall be declared with the static storage class.	*FC+E*
8.11	When an array with external linkage is declared, its size should be explicitly specified.	*FC+E*
8.12	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique.	*FC+E*
8.13	A pointer should point to a const-qualified type whenever possible.	*PC*
8.14	The restrict type qualifier shall not be used.	*FC+E*

Initialization		Support
9.1	The value of an object with automatic storage duration shall not be read before it has been set. Without Astrée, this rule is checked only partially and with reduced precision.	*PC*
9.2	The initializer for an aggregate or union shall be enclosed in braces.	*FC+E*

continues on the next page...

Initialization		Support
...continued		
9.3	Arrays shall not be partially initialized.	*FC+E*
9.4	An element of an object shall not be initialized more than once.	*FC+E*
9.5	Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.	*FC+E*

The essential type model		Support
10.1	Operands shall not be of an inappropriate essential type. Some violations of that rule (especially use of floating points with integer operators) are rejected by the frontend as language error.	*FC*
10.2	Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations.	*FC+E*
10.3	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.	*FC+E*
10.4	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.	*FC+E*
10.5	The value of an expression should not be cast to an inappropriate essential type.	*FC+E*
10.6	The value of a composite expression shall not be assigned to an object with wider essential type.	*FC+E*
10.7	If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type.	*FC+E*
10.8	The value of a composite expression shall not be cast to a different essential type category or a wider essential type.	*FC+E*

Pointer type conversions		Support
11.1	Conversions shall not be performed between a pointer to a function and any other type.	*FC+E*
11.2	Conversions shall not be performed between a pointer to an incomplete type and any other type.	*FC+E*

continues on the next page...

Pointer type conversions		Support
…continued		
11.3	A cast shall not be performed between a pointer to object type and a pointer to a different object type.	*FC+E*
11.4	A conversion should not be performed between a pointer to object and an integer type.	*FC+E*
11.5	A conversion should not be performed from pointer to void into pointer to object.	*FC+E*
11.6	A cast shall not be performed between pointer to void and an arithmetic type.	*FC+E*
11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type.	*FC+E*
11.8	A cast shall not remove any const or volatile qualification from the type pointed to by a pointer.	*FC+E*
11.9	The macro NULL shall be the only permitted form of integer null pointer constant.	*FC+E*

Expressions		Support
12.1	The precedence of operators within expressions should be made explicit.	*FC+E*
12.2	The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand.	*PC*
12.3	The comma operator should not be used.	*FC+E*
12.4	Evaluation of constant expressions should not lead to unsigned integer wrap-around.	*FC+E*

Side effects		Support
13.1	Initializer lists shall not contain persistent side effects. File modifications are not taken into account.	*PC*
13.2	The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders. File modifications are not taken into account.	*PC*

continues on the next page…

Side effects		Support
...continued		
13.3	A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator.	*FC+E*
13.4	The result of an assignment operator should not be used.	*FC+E*
13.5	The right hand operand of a logical && or || operator shall not contain persistent side effects. File modifications are not taken into account.	*PC*
13.6	The operand of the sizeof operator shall not contain any expression which has potential side effects.	*FC+E*

Control statement expressions		Support
14.1	A loop counter shall not have essentially floating type.	*PC*
14.2	A for loop shall be well-formed. Only the side effect criterion for loop condition expressions is checked. File modifications are not taken into account.	*PC*
14.3	Controlling expressions shall not be invariant. Violations of this rule are reported for expressions that can be proven to be invariant by the analyzer. It cannot be guaranteed that all invariant expressions are reported. Without Astrée, this rule is checked with reduced coverage.	*PC*
14.4	The controlling expression of an if statement and the controlling expression of an iteration- statement shall have essentially Boolean type.	*FC+E*

Control flow		Support
15.1	The goto statement should not be used.	*FC+E*
15.2	The goto statement shall jump to a label declared later in the same function.	*FC+E*
15.3	Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement.	*FC+E*
15.4	There should be no more than one break or goto statement used to terminate any iteration statement.	*FC+E*

continues on the next page...

Control flow		Support
…continued		
15.5	A function should have a single point of exit at the end.	*FC+E*
15.6	The body of an iteration-statement or a selection-statement shall be a compound-statement.	*FC+E*
15.7	All if … else if constructs shall be terminated with an else statement.	*FC+E*

Switch statements		Support
16.1	All switch statements shall be well-formed.	*FC+E*
16.2	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement. The required content of the respective default clause (comment or statement, see amplification of this rule) is currently not checked because of the absence of comments in preprocessed code.	*FC+E*
16.3	An unconditional break statement shall terminate every switch-clause.	*FC+E*
16.4	Every switch statement shall have a default label. The required content of the respective default clause (comment or statement, see amplification of this rule) is currently not checked because of the absence of comments in preprocessed code.	*PC*
16.5	A default label shall appear as either the first or the last switch label of a switch statement.	*FC+E*
16.6	Every switch statement shall have at least two switch-clauses.	*FC+E*
16.7	A switch-expression shall not have essentially Boolean type.	*FC+E*

Functions		Support
17.1	The features of <stdarg.h> shall not be used.	*FC+E*
17.2	Functions shall not call themselves, either directly or indirectly.	*PC*
17.3	A function shall not be declared implicitly.	*FC+E*
17.4	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	*FC+E*

continues on the next page…

Functions		Support
...continued		
17.5	The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements.	*PC*
17.6	The declaration of an array parameter shall not contain the static keyword between the [].	*FC+E*
17.7	The value returned by a function having non-void return type shall be used.	*FC+E*
17.8	A function parameter shall not be modified. Violations of this rule are only reported for explicit assignments to identifiers declared as parameter. Modifications of parameters via pointers (e.g. by called functions) are not reported.	*PC*

Pointers and arrays		Support
18.1	A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand. This rule check is supported by Astrée's semantic analysis: Astrée reports all invalid pointer dereferences (including out-of-bound accesses) resulting from violations of this rule.	*PC*
18.2	Subtraction between pointers shall only be applied to pointers that address elements of the same array. There is no warning when subtracting pointers that point to different fields of the same structure. This rule check is supported by Astrée's semantic analysis: Astrée reports all invalid pointer dereferences (including out-of-bound accesses) resulting from violations of this rule.	*PC*
18.3	The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object.	*PC*
18.4	The +, -, += and -= operators should not be applied to an expression of pointer type.	*FC+E*
18.5	Declarations should contain no more than two levels of pointer nesting.	*FC+E*
18.6	The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist.	*PC*
18.7	Flexible array members shall not be declared.	*FC+E*

continues on the next page...

Pointers and arrays		Support
...continued		
18.8	Variable-length array types shall not be used.	*FC+E*

Overlapping storage		Support
19.1	An object shall not be assigned or copied to an overlapping object. This rule check is supported by Astrée's semantic analysis: Astrée allows for detecting additional classes of violations of this rule on top of the syntax-based checks. Without Astrée, this rule is checked with reduced coverage.	*PC*
19.2	The union keyword should not be used.	*FC+E*

Preprocessing directives		Support
20.1	#include directives should only be preceded by preprocessor directives or comments.	*FC+E*
20.2	The ', " or \ characters and the /* or // character sequences shall not occur in a header file name.	*FC+E*
20.3	The #include directive shall be followed by either a <filename> or "filename" sequence.	*FC+E*
20.4	A macro shall not be defined with the same name as a keyword.	*FC+E*
20.5	#undef should not be used.	*FC+E*
20.6	Tokens that look like a preprocessing directive shall not occur within a macro argument.	*FC+E*
20.7	Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses.	*FC+E*
20.8	The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1.	*FC+E*
20.9	All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #define'd before evaluation.	*FC+E*
20.10	The # and ## preprocessor operators should not be used.	*FC+E*
20.11	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator.	*FC+E*

continues on the next page...

Preprocessing directives		Support
...continued		
20.12	A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators.	*FC+E*
20.13	A line whose first token is # shall be a valid preprocessing directive.	*FC+E*
20.14	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related.	*FC+E*

Standard libraries		Support
21.1	#define and #undef shall not be used on a reserved identifier or reserved macro name.	*FC+E*
21.2	A reserved identifier or macro name shall not be declared.	*FC+E*
21.3	The memory allocation and deallocation functions of <stdlib.h> shall not be used.	*FC+E*
21.4	The standard header file <setjmp.h> shall not be used.	*FC+E*
21.5	The standard header file <signal.h> shall not be used.	*FC+E*
21.6	The Standard Library input/output functions shall not be used.	*FC+E*
21.7	The atof, atoi, atol and atoll functions of <stdlib.h> shall not be used.	*FC+E*
21.8	The standard library functions abort, exit, getenv and system of <stdlib.h> shall not be used.	*FC+E*
21.9	The standard library functions bsearch and qsort of <stdlib.h> shall not be used.	*FC+E*
21.10	The Standard Library time and date functions shall not be used.	*FC+E*
21.11	The standard header file <tgmath.h> shall not be used.	*FC+E*
21.12	The exception handling features of <fenv.h> should not be used.	*FC+E*

Resources		Support
22.1	All resources obtained dynamically by means of Standard Library functions shall be explicitly released.	*PC*

continues on the next page...

Resources		Support
…continued		
22.2	A block of memory shall only be freed if it was allocated by means of a Standard Library function.	*PC*
22.3	The same file shall not be open for read and write access at the same time on different streams. Can be validated by the run-time error analysis using appropriate stub implementations.	*PC*
22.4	There shall be no attempt to write to a stream which has been opened as read-only.	*PC*
22.5	A pointer to a FILE object shall not be dereferenced.	*PC*
22.6	The value of a pointer to a FILE shall not be used after the associated stream has been closed.	*PC*

3 MISRA C:2012 Amendment 1

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.

Directives		**Support**
D.4.14	The validity of values received from external sources shall be checked.	*PC*

Expressions		**Support**
12.5	The sizeof operator shall not have an operand which is a function parameter declared as "array of type".	*FC+E*

Standard libraries		**Support**
21.8	The standard library functions abort, exit and system of <stdlib.h> shall not be used.	*FC+E*
21.13	Any value passed to a function in <ctype.h> shall be representable as an unsigned char or be the value EOF.	*PC*
21.14	The Standard Library function memcmp shall not be used to compare null terminated strings.	*PC*
21.15	The pointer arguments to the Standard Library functions memcpy, memmove and memcmp shall be pointers to qualified or unqualified versions of compatible types.	*PC*
21.16	The pointer arguments to the Standard Library function memcmp shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type.	*PC*

continues on the next page…

Standard libraries		Support
...continued		
21.17	Use of the string handling functions from <string.h> shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters. Violations of this rule which cause a runtime error are reported by Astrée.	*PC*
21.18	The size_t argument passed to any function in <string.h> shall have an appropriate value. Violations of this rule which cause a runtime error are reported by Astrée.	*PC*
21.19	The pointers returned by the Standard Library functions localeconv, getenv, setlocale or, strerror shall only be used as if they have pointer to const-qualified type.	*PC*
21.20	The pointer returned by the Standard Library functions asctime, ctime, gmtime, localtime, localeconv, getenv, setlocale or strerror shall not be used following a subsequent call to the same function.	*PC*

Resources		Support
22.7	The macro EOF shall only be compared with the unmodified return value from any Standard Library function capable of returning EOF.	*PC*
22.8	The value of errno shall be set to zero prior to a call to an errno-setting-function.	*PC*
22.9	The value of errno shall be tested against zero after calling an errno-setting-function.	*PC*
22.10	The value of errno shall only be tested when the last function to be called was an errno-setting-function.	*PC*

4 MISRA C:2012 Amendment 2

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.

A standard C environment		**Support**
1.4	Emergent language features shall not be used.	*PC*

Standard libraries		**Support**
21.3	The memory allocation and deallocation functions of <stdlib.h> shall not be used.	*FC+E*
21.8	The Standard Library termination functions of <stdlib.h> shall not be used.	*FC+E*
21.21	The Standard Library function system of <stdlib.h> shall not be used.	*FC+E*

5 MISRA C:2012 Amendment 3

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.

Directives		**Support**
D.4.15	Evaluation of floating-point expressions shall not lead to the undetected generation of infinities and NaNs.	*NC*

A standard C environment		**Support**
1.4	Emergent language features shall not be used.	*PC*
1.5	Obsolescent language features shall not be used. The checks for this rule comprise all statically detectable obsolescent language features.	*PC*

Types		**Support**
6.3	A bit field shall not be declared as a member of a union.	*FC+E*

Literals and constants		**Support**
7.5	The argument of an integer constant macro shall have an appropriate form.	*FC+E*

Declarations and definitions		**Support**
8.15	All declarations of an object with an explicit alignment specification shall specify the same alignment.	*FC*

continues on the next page…

Declarations and definitions		Support
...continued		
8.16	The alignment specification of zero should not appear in an object declaration.	*FC*
8.17	At most one explicit alignment specifier should appear in an object declaration.	*FC*

The essential type model		Support
10.1	Operands shall not be of an inappropriate essential type. Some violations of that rule (especially use of floating points with integer operators) are rejected by the frontend as language error.	*FC*

Functions		Support
17.9	A function declared with a _Noreturn function specifier shall not return to its caller.	*FC*
17.10	A function declared with a _Noreturn function specifier shall have void return type.	*FC*
17.11	A function that never returns should be declared with a _Noreturn function specifier.	*PC*
17.12	A function identifier should only be used with either a preceding &, or with a parenthesized parameter list.	*FC*
17.13	A function type shall not be type qualified.	*FC*

Pointers and arrays		Support
18.9	An object with temporary lifetime shall not undergo array-to-pointer conversion.	*PC*

Standard libraries		Support
21.11	The standard header file <tgmath.h> shall not be used.	*FC*
21.12	The standard header file <fenv.h> shall not be used.	*FC*
21.22	All operand arguments to any type-generic macros declared in <tgmath.h> shall have an appropriate essential type.	*FC*

continues on the next page...

Standard libraries		Support
…continued		
21.23	All operand arguments to any multi-argument type-generic macros declared in <tgmath.h> shall have the same standard type.	*FC*
21.24	The random number generator functions of <stdlib.h> shall not be used.	*FC*

Generic selections		Support
23.1	A generic selection should only be expanded from a macro.	*FC*
23.2	A generic selection that is not expanded from a macro shall not contain potential side effects in the controlling expression.	*FC+E*
23.3	A generic selection should contain at least one non-default association.	*FC+E*
23.4	A generic association shall list an appropriate type.	*FC+E*
23.5	A generic selection should not depend on implicit pointer type conversion.	*FC+E*
23.6	The controlling expression of a generic selection shall have an essential type that matches its standard type.	*FC+E*
23.7	A generic selection that is expanded from a macro should evaluate its argument only once.	*NC*
23.8	A default association shall appear as either the first or the last association of a generic selection.	*FC+E*

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.