# MISRA C:2012 Amendment 3

Updates for ISO/IEC 9899:2011/2018
Phase 2 — New C11/C18 features

October 2022

First published October 2022 by The MISRA Consortium Limited
1 St James Court
Whitefriars
Norwich
Norfolk
NR3 1RU
UK

www.misra.org.uk

Copyright © 2022 The MISRA Consortium Limited

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical or photocopying, recording or otherwise without the prior written permission of the Publisher.

"MISRA", "MISRA C" and the triangle logo are registered trademarks owned by The MISRA Consortium Limited. Other product or brand names are trademarks or registered trademarks of their respective holders and no endorsement or recommendation of these products by MISRA is implied.

ISBN 978-1-911700-02-9 PDF

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

# MISRA C:2012 Amendment 3

Updates for ISO/IEC 9899:2011/2018
Phase 2 — New C11/C18 features

October 2022

# MISRA Mission Statement

We provide world-leading, best practice guidelines for the safe and secure application of both embedded control systems and standalone software.

MISRA is a collaboration between manufacturers, component suppliers and engineering consultancies which seeks to promote best practice in developing safety- and security-related electronic systems and other software-intensive applications. To this end, MISRA publishes documents that provide accessible information for engineers and management, and holds events to permit the exchange of experiences between practitioners.

Disclaimer

*Adherence to the requirements of this document does not in itself ensure error-free robust software or guarantee portability and re-use.*

*Compliance with the requirements of this document, or any other standard, does not of itself confer immunity from legal obligations.*

# Foreword

An updated edition of the C Standard, ISO/IEC 9899:2011 and commonly referred to as C11, was released just as MISRA C:2012 was being prepared for publication, meaning it arrived too late for the MISRA C Working Group to take it into consideration.

Subsequently, a minor revision, ISO/IEC 9899:2018 and commonly referred to as C18, followed. While C18 did not introduce any new features, it made some refinements to the existing C Standard.

As the adoption of C11 became more widespread, the MISRA C Working Group decided that it was time to address the new edition of the C Standard, support for which is being implemented by means of a series of amendments to MISRA C:2012 – the first of which, MISRA C:2012 Amendment 2 *C11 Core* was published in 2020.

This document further amends MISRA C:2012 as required to introduce support for more of the features introduced by ISO/IEC 9899:2011, while also improving the guidance for the earlier C Standard – it adds one new directive and twenty-three new rules, as well as revising a number of existing guidelines and supporting material.

Subsequent amendments will be used to introduce specific guidance for the remaining C11 features.

We trust that this amendment will be welcomed by the community at large, and will offer confidence to projects and organizations who have held off migrating to C11.

Andrew Banks FBCS CITP
Chairman, MISRA C Working Group

# Acknowledgements

The MISRA consortium would like to thank the following individuals for their significant contribution to the writing of this document:

| | |
|---|---|
| Andrew Banks | LDRA Ltd. and Intuitive Consulting |
| Roberto Bagnara | BUGSENG and University of Parma |
| Alex Gilding | Perforce Software Inc. |
| Chris Tapp | LDRA Ltd. and Keylevel Consultants Ltd. |

The MISRA consortium also wishes to acknowledge contributions from the following members of the MISRA C Working Group during the development and review process:

| | |
|---|---|
| Dave Banham | BlackBerry Ltd. |
| Jill Britton | Perforce Software Inc. |
| Daniel Kästner | AbsInt Angewandte Informatik GmbH |
| Gerlinde Kettl | Vitesco Technologies GmbH |
| Gavin McCall | Independent |
| Michal Rozenau | Parasoft Corp |
| Liz Whiting | LDRA Ltd. |

The MISRA consortium also wishes to acknowledge contributions from the following individuals during the development and review process:

| | |
|---|---|
| Gaétan Noël | BlackBerry Ltd. |
| David Ward | HORIBA MIRA Ltd. |

DokuWiki was used extensively during the drafting of this document. Our thanks go to all those involved in its development.

This document was typeset using Open Sans. Open Sans is a trademark of Google and may be registered in certain jurisdictions. Digitized data copyright © 2010–2011, Google Corporation. Licensed under the Apache License, Version 2.0.

# Contents

1 Overview 1

1.1 Applicability 1

1.2 C language updates 1

1.3 Future directions 2

2 Amendments 3

2.1 Section 6 — Introduction to the guidelines 3

2.2 Section 7 — Directives 3

2.5 Appendix C — Type safety issues with C 38

3 Consequential amendments 43

3.1 Consequential amendments to other existing guidelines 43

3.2 Appendix A — Summary of guidelines 44

3.3 Appendix B — Guidelines attributes 45

3.4 Appendix H — Undefined and critical unspecified behaviour 46

3.6 Addendum 3 — Coverage against CERT C 47

4 References 48

4.1 MISRA C 48

4.2 The C Standard 48

4.3 Other Standards 49

# 1 Overview

## 1.1 Applicability

This amendment is intended to be used with MISRA C:2012 (Third Edition, First Revision) [2] as revised and amended by

- MISRA C:2012 Technical Corrigendum 2 [4], and
- MISRA C:2012 Amendment 2 [6]

This amendment is also compatible with MISRA C:2012 (Third Edition) [1] as revised and amended by:

- MISRA C:2012 Technical Corrigendum 1 [3],
- MISRA C:2012 Technical Corrigendum 2 [4],
- MISRA C:2012 Amendment 1 [5], and
- MISRA C:2012 Amendment 2 [6]

## 1.2 C language updates

This document further amends MISRA C:2012 [2] as follows:

1. To permit the use, with restrictions, of the following ISO/IEC 9899:2011 [11] features:
   - No-return functions (`<stdnoreturn.h>`)
   - Alignment of objects (`<stdalign.h>`)
   - Type-generic expressions (`_Generic`)
2. To provide further guidance on the use of the following C language features:
   - Type-generic math macros (`<tgmath.h>`)
   - Floating-point (including comparisons, NaNs and infinities)
   - Complex numbers
3. To restrict usage of features declared as obsolescent by the C Standard

When using ISO/IEC 9899:2011 [11], use of the following features remains prohibited without the support of a deviation against Rule 1.4:

- Support for multiple threads of execution (`<stdatomic.h>`, `<threads.h>`)
- Bounds-checking interfaces (Annex K)

*Notes:*

1. ISO/IEC 9899:2018 [12] incorporates corrigenda applicable to ISO/IEC 9899:2011 [11]. As such, it is functionally equivalent to ISO/IEC 9899:2011 and is therefore also supported through this amendment.

## 1.3 Future directions

It is a long-term objective for MISRA C to define a predictable subset of the C language, and to provide explicit guidance for the avoidance of all instances of undefined and critically unspecified behaviour. For this reason, features that are initially permitted without the support of specific guidelines may be subject to restrictions in the future.

Further subsequent amendments will introduce specific guidelines to cover the remaining features that are prohibited by Rule 1.4, supporting their use through the introduction of guidance that must be followed to avoid any undefined or unspecified behaviour.

# 2 Amendments

## 2.1 Section 6 — Introduction to the guidelines

### 2.1.1 Amend Section 6.9 — Presentation of the guidelines

#### Amendment

Expand the examples to include *typedef*s and objects of complex type.

AMD3.1 : In the assumed list of *typedef*s, replace:

```
float32_t         /* 32-bit floating-point   */
float64_t         /* 64-bit floating-point   */
```

with:

```
float32_t         /*  32-bit floating-point - real    */
float64_t         /*  64-bit floating-point - real    */
float128_t        /* 128-bit floating-point - real    */
cfloat32_t        /*  32-bit floating-point - complex */
cfloat64_t        /*  64-bit floating-point - complex */
cfloat128_t       /* 128-bit floating-point - complex */
```

AMD3.2 : In the example list of objects, replace:

```
float32_t f32a;    /* 32-bit floating-point   */
float64_t f64b;    /* 64-bit floating-point   */
```

with:

```
float32_t   f32a;   /*  32-bit floating-point - real    */
float64_t   f64b;   /*  64-bit floating-point - real    */
float128_t  f128c;  /* 128-bit floating-point - real    */
cfloat32_t  cf32a;  /*  32-bit floating-point - complex */
cfloat64_t  cf64b;  /*  64-bit floating-point - complex */
cfloat128_t cf128c; /* 128-bit floating-point - complex */
```

## 2.2 Section 7 — Directives

### 2.2.1 Directive 4.6 — Typedefs

#### Amendment

Add support for complex types

AMD3.3 : In the "Amplification" section, replace:

For example, on a 32-bit C90 implementation the following definitions might be suitable:

with:

For example, on a C90 implementation (where the standard header `<stdint.h>` is not available) the following definitions may be suitable, but need to be adjusted to suit the project's implementation:

AMD3.4 : In the "Amplification" section, add a new paragraph immediately before the definition of `float32_t`:

For real and complex floating-point objects, the following definitions may be suitable (depending on the implementation), where the specified size represents the precision:

AMD3.5 : In the "Amplification" section, add the following after the definition of `float128_t`:

```
typedef      float  _Complex cfloat32_t;
typedef      double _Complex cfloat64_t;
typedef long double _Complex cfloat128_t;
```

### 2.2.2 Directive 4.9 — Function-like macros

#### Amendment

Add an exception permitting the use of *function-like macros* for `_Generic` selections.

AMD3.6 : Add a new "Exception" section, as follows:

#### Exception

The use of *function-like macros* for `_Generic` selections is advised by Rule 23.1, and is therefore permitted by this Directive.

AMD3.7 : Add to further example:

The following is compliant by exception – in most cases `_Generic` selections cannot be replaced with a function.

```
#define GFUNC(X) ( _Generic( ... ) )     /* Compliant by exception */
```

AMD3.8 : Add to "See also" list:

, Rule 23.1

### 2.2.3 Directive 4.11 — Precision

#### Amendment

Add guidance on the use of precision

AMD3.9 : In the "Rationale" section, add a new paragraph immediately after the existing paragraph beginning *Although most...*.

The trigonometric periodic functions in `<math.h>` incur significant precision loss when called with arguments with relatively large absolute value. If $x$ is an IEC 60559 single-precision number and $x \geq 2^{23}$, then the smallest single-precision range containing $[x, x + 2\pi)$ contains no more than three floating-point numbers (the same holds for IEC 60559 double-precision numbers substituting $2^{52}$ to $2^{23}$). This implies that computing trigonometric periodic functions on large values, because of the gross input inaccuracy, gives non-significant results independently from the quality of the function implementation. For this reason, trigonometric periodic functions should not be called on arguments whose absolute value is larger than $k \cdot \pi$, where $k$ is a property of the floating-point representation. Ideally, this principle should be applied with $k = 1$, but depending on the application and its precision goals, a larger value for $k$ can be used.

### 2.2.4 Directive 4.15 — Infinities and NaNs

#### Amendment

Add guidance on the undetected generation of infinities and NaNs

AMD3.10 : Add the following new directive after Directive 4.14:

## Dir 4.15 Evaluation of floating-point expressions shall not lead to the undetected generation of infinities and NaNs

C90 [Implementation 20, 22]
C99 [Unspecified 30, 46, 47, 48, 49; Implementation J.3.6(2), J.3.6(5)]
C11 [Unspecified 30, 52, 53, 56, 57; Implementation J.3.6(2), J.3.6(5)]

| | |
|---|---|
| Category | Required |
| Applies to | C90, C99, C11 |

### Amplification

If the evaluation of an arithmetic floating-point expression or a call to a mathematical function can possibly generate an infinity or a NaN (not-a-number), then that result shall be adequately tested. For efficiency reasons, it is possible to exploit the implementation-defined propagation rules for infinities and NaNs to delay the test, provided no infinity or NaN reaches sections of code that have not been designed to handle them.

### Rationale

Operations on floating-point numbers can overflow, that is, generate results of magnitude too large to be represented in the considered format. For such cases, the widely-used IEC 60559 Standard has special representations for positive and negative infinity. While the possibility to denote infinities is useful in some contexts because it allows operations to continue after overflow has occurred, much care has to be exercised, because:

- it is likely that an error has occurred that resulted in an approximation error of unknown magnitude;
- infinities can easily generate unwanted underflows;
- careless computation with infinities can cause the generation of NaNs, whose correct handling requires even more care.

Note: an infinity does not necessarily correspond to a *large* value: for instance, on an implementation that conforms to IEC 60559, a float computation whose exact result is slightly larger than `3.4e+38` will be rounded to infinity and `logf( infinity )` returns infinity even though `logf( 3.4e+38 )` is less than 89.

Some operations on floating-point numbers are invalid or undetermined, such as taking the square root of a negative number or dividing a zero by a zero. The IEC 60559 Standard has special representations for such exceptional, non-numerical results: they are called NaNs. While of course there are legitimate uses of NaNs, programming with NaNs requires exceptional care, for example:

- in some implementations there are both “signalling” and “quiet” NaNs;
- NaNs can be signed;
- `x == x` is false if `x` is a NaN;
- some library functions (such as `hypot()`) can return a non-NaN even if one of the arguments is NaN.

For these reasons, projects that do use infinites or NaNs need to take measures ensuring such special representations are detected and handled before they propagate to areas of the systems that are not prepared to receive them.

## Example

```
#include <math.h>

extern float64_t get_result();                /* Return value may be infinite
                                                 ... or valid data              */
extern void      use_result( float64_t c );  /* Not protected against NANs or
                                                 ... or infinities              */

void f( void )
{
  float64_t a = get_result();

  use_result( a );      /* Non-compliant - c not tested      */

  if ( !isnan( a ) )
  {
    use_result( a );    /* Non-compliant - c may be infinite */
  }

  if ( isfinite( a ) )
  {
    use_result( a );    /* Compliant */
  }
}
```

The following example shows that it is not necessary to check for NaNs or infinities at every step:

```
void g( void )
{
  float64_t a = get_result();

  float64_t b = exp( -2.0 * a );   /* Compliant     - exp() propagates infinities
                                                      ... and NaNs as expected    */

  float64_t c = (a * (1.0 + b) - (1.0 - b)) / (a * a);

  /* Division can result in NaN, even if operands are not infinity or NaN         */

  use_result( c );                 /* Non-compliant - not protected against NaNs
                                                      ... or infinities           */

  if ( isfinite( c ) )
  {
    use_result( c );               /* Compliant     - protected against NaNs      */
                                                      ... and infinities          */
  }
}
```

## See also

Dir 1.1, Dir 4.1, Dir 4.7

## 2.3 Section 8 — Rules

### 2.3.1 Amend Rule 1.4 — General restrictions

#### Amendment

Remove the general restriction on features covered by this amendment.

AMD3.11 : Delete the first bullet point, relating to the `_Generic` keyword.

AMD3.12 : Delete the second bullet point, relating to the `<stdnoreturn.h>` header file.

AMD3.13 : Delete the fifth bullet point, relating to the `<stdalign.h>` header file.

### 2.3.2 New Rule 1.5 — Obsolescent features

#### Amendment

Add guidance on the use of obsolescent language features.

AMD3.14 : Add the following new rule after Rule 1.4:

#### Rule 1.5 Obsolescent language features shall not be used

| | |
|---|---|
| Category | Required |
| Analysis | Undecidable, System |
| Applies to | C99, C11 |

#### Amplification

Obsolescent features are those identified in the *Future language directions* and *Future library directions* sections of the C Standard, and are listed in Appendix F.

#### Rationale

Features are declared as obsolescent by the C Standard when they are superseded by safer or better alternatives, or are considered to exhibit undesirable behaviour. Features declared as obsolescent by a particular version of the C Standard may be withdrawn in a later version.

#### See also

Rule 1.1

### 2.3.3 New Rule 6.3 — Bit fields

#### Amendment

Add further guidance on the use of bit fields.

AMD3.15 : Add the following new rule after Rule 6.2:

## Rule 6.3 A bit field shall not be declared as a member of a union

| | |
|---|---|
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99, C11 |

C90 [Implementation 30]
C99 [Implementation J.3.9(4)
C11 [Implementation J.3.9(5)]

### Amplification

A member of a union shall not be declared as a bit field.

This rule does not apply to sub-objects within union members that do not themselves have a union type.

### Rationale

The exact bitwise position of a bit field within a storage unit is implementation defined. Therefore, if two bit fields are declared such that they fit within the same storage unit of a union, the compiler is not required to overlay them over one another beginning from the starting bit of the storage unit.

If the union is used for *type-punning*, it is therefore unclear which bits of the previously-stored value will be accessed by the bit field.

If the union is not intended to be used for *type-punning*, there is no point in declaring the members as bit fields, because no space will be saved (a complete storage unit will need to be allocated within the union anyway).

### Example

```
/* Compliant     - if the user wants to type-pun, the bits of 'big' which will
                   ... be overlaid are clearly identified                     */
union U1 {
  uint8_t small;
  uint32_t big;
};

/* Non-compliant - it is unclear which, if any, bits of 'big' are overlaid by
                   ... 'small' in this type */
union U2 {
  uint32_t small:8;
  uint32_t big;
};

/* Non-compliant - it is unclear if any bits of 'big' are overlaid by
                   ... 'small' in this type */
union U3 {
  uint32_t small: 8;
  uint32_t big  :24;
};

/* Compliant     - a sub-object can be a bit-field */
union U4 {
  struct {
    uint8_t a:4;
    uint8_t b:4;
    uint8_t c:4;
    uint8_t d:4;
  } q;
  uint16_t r;
};
```

### 2.3.4 New Rule 7.5 — Integer constant macros

#### Amendment

Add guidance on the use of integer constant macros.

AMD3.16 : Add the following new rule after Rule 7.4:

| Rule 7.5 | The argument of an integer constant macro shall have an appropriate form |
|---|---|

C99 [Undefined 137], C11 [Undefined 145]

| | |
|---|---|
| Category | Mandatory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C99, C11 |

#### Amplification

The argument of an integer constant macro shall satisfy the following:

- The argument must be an unsuffixed integer (decimal, octal or hexadecimal) literal;
- The value of the argument must not exceed the limits for the equivalent *exact-width* type indicated by the name of the macro used. For example, the argument to `UINT16_C` must be representable as an unsigned 16-bit value.

### Rationale

The behaviour is undefined if the argument of an integer constant macro does not have an appropriate form.

### Example

```
#include <stdint.h>

uint32_t u1 = UINT32_C( 10   );    /* Compliant                               */
uint32_t u2 = UINT32_C( 10UL );    /* Non-compliant - constant is suffixed    */
uint32_t u3 = UINT32_C( 10.0 );    /* Non-compliant - floating-point constant */
uint16_t u4 = UINT16_C(  -2  );    /* Non-compliant - constant expression     */

 int32_t s1 =  INT32_C(  -2  );    /* Non-compliant - constant expression     */
 int32_t s2 = -INT32_C(   2  );    /* Compliant                               */
```

In the following example, the constant has a value that cannot be represented in 16 bits.

```
uint_least16_t u5 = UINT16_C ( 0x10000 ); /* Non-compliant, even if uint_least16_t
                                             ... is implemented as a 32-bit type */
```

## 2.3.5 New Rules 8.15-8.17 — Alignment

### Amendment

Add guidance on the use of alignment.

AMD3.17 : Add the following new rules after Rule 8.14:

**Rule 8.15** All declarations of an object with an explicit *alignment specification* shall specify the same *alignment*

C11 [Undefined 73]

| | |
|---|---|
| Category | Required |
| Analysis | Decidable, System |
| Applies to | C11 |

### Amplification

If any declaration (including the definition) of an object includes an explicit *alignment specification*, all other declarations of the same object shall also include the same explicit *alignment specification*.

Two *alignment specifications* are the same if they are lexically identical after macro expansion.

This rule applies both to alignments specified directly by an expression, and by naming a type.

### Rationale

If multiple declarations of an object with external linkage have conflicting *alignment specifications*, the behaviour is undefined. Therefore, if the alignment of the object is important, it should be listed explicitly in order to avoid the risk of creating conflicting declarations in multiple translation units. If the alignment of the object does not need to be set explicitly, the *alignment specification* should be omitted entirely to avoid confusion.

Since an object with internal linkage is only accessible by name from within a single translation unit, there is no direct risk of incompatible alignment specifications causing undefined behaviour.

However, this rule reduces the risk of accidentally introducing undefined behaviour during maintenance if the code is later modified to give an object external linkage instead.

If the alignment operand is a type name and the *alignment specifications* do not consistently use that same type name, there is a risk of introducing inconsistency if the configuration changes, such as if code is recompiled on a different platform with different fundamental type alignments.

## Example

```
/* header.h - #included by both file1.c and file2.c */

extern alignas (16)        int32_t a;
extern alignas (0)         int32_t b;
extern                     int32_t c;
extern                     int32_t d;
extern alignas (16)        int32_t e;
extern alignas (16)        int32_t f;
extern                     int32_t g;

extern alignas (float)     int32_t i;
extern alignas (float)     int32_t j;
extern alignas (float)     int32_t k;
extern alignas (float)     int32_t l;
extern alignas (float32_t) int32_t m;

/* file1.c */

alignas (16)        int32_t a; /* Compliant     - same explicit alignment         */
alignas (16)        int32_t b; /* Non-compliant - not consistently explicit       */
alignas (16)        int32_t c; /* Non-compliant - not consistently explicit       */
                    int32_t d; /* Compliant     - not manually aligned            */
                    int32_t e; /* Non-compliant - constraint violation            */
alignas (16)        int32_t f; /* Non-compliant because of file2.c                */
alignas (16)        int32_t g; /* Non-compliant, and undefined because of file2.c */

extern alignas (16) int32_t h; /* Non-compliant with file2.c                      */

alignas (float)     int32_t i; /* Compliant     - same type used                  */
alignas (double)    int32_t j; /* Non-compliant - different type, and therefore
                                                  may be a constraint violation   */
alignas (4)         int32_t k; /* Non-compliant - regardless of the size of float */
alignas (float32_t) int32_t l; /* Non-compliant - potentially a different type on
                                                  a different platform            */
alignas (float32_t) int32_t m; /* Compliant     - same type used by name          */

/* file2.c */
extern              int32_t f; /* Non-compliant - not consistent with
                                                  either file1.c or header.h      */

extern alignas (8)  int32_t g; /* Non-compliant - undefined behaviour because of
                                                  inconsistency with file1.c      */

extern alignas (8)  int32_t h; /* Non-compliant - not consistent with file1.c
                                                  and undefined behaviour         */
```

## See also

Rule 8.6, Rule 8.16, Rule 8.17

## Rule 8.16 The *alignment specification* of zero should not appear in an object declaration

| | |
|---|---|
| Category | Advisory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C11 |

### Amplification

This rule applies to any *integer constant expression* operand to `_Alignas` that evaluates to zero.

### Rationale

If the alignment of an object is important, it should be specified explicitly.

If configuration settings or platform implementation details are intended to change the alignment of an object to conditionally disable explicit alignment, this should be abstracted by the preprocessor.

### Example

```
             int32_t a; /* Compliant: no alignment specification                */
alignas (16) int32_t b; /* Compliant: explicit non-zero alignment specification */
alignas (0)  int32_t c; /* Non-compliant: zero-alignment specification          */

/* Non-compliant on platforms where sizeof (int) == sizeof (long) */
alignas (sizeof (long) - sizeof (int)) int32_t d;
```

When the alignment is not important, the configuration can remove it entirely:

```
#if REQUIRED_ALIGNMENT > 0
#define ALIGNED alignas (REQUIRED_ALIGNMENT)
#else
#define ALIGNED /**/
#endif
```

### See also

Rule 8.15, Rule 8.17

## Rule 8.17 At most one explicit *alignment specifier* should appear in an object declaration

| | |
|---|---|
| Category | Advisory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C11 |

### Rationale

If the alignment of an object is important, it should be specified explicitly.

Because an *alignment specifier* only places a minimum requirement on the actual alignment of an object, C permits a declaration to contain multiple *alignment specifiers*, with the strictest imposing the final requirement.

If separate conditions require different minimum permitted alignments for an object, they should be combined explicitly by the expression controlling the specifier. Otherwise, the conflicting *alignment specifiers* risk obscuring the intent of the declaration from a human reviewer.

### Example

```
int32_t a;                         /* Compliant     -  no alignment specifier  */
alignas(16)             int32_t b; /* Compliant     - one alignment specifier  */
alignas(16) alignas(8)  int32_t c; /* Non-compliant - two alignment specifiers */
alignas(16) alignas(0)  int32_t d; /* Non-compliant - also violates Rule 8.16  */
```

The following example shows a way of generating conditional alignment:

```
#define SML_ALIGN 16
#define BIG_ALIGN 32
alignas(MAX(SML_ALIGN, BIG_ALIGN)) int32_t e; /* Compliant */
```

### See also

Rule 8.15, Rule 8.16

## 2.3.6 Amend Section 8.10.2 — Essential types

### Amendment

Expand the *essential type model* to include complex types.

AMD3.18 : In the list of *essential type categories*, replace:

- *Essentially floating*.

With:

- *Essentially floating*, which may be either:
  - *Essentially real floating*, or
  - *Essentially complex floating*.

Note: For the purposes of this *essential type model*, *essentially real floating* and *essentially complex floating* are considered to be distinct *essential type categories*.

AMD3.19 : Replace the table showing the essential types with:

| *Essential type category* | | | | |
|---|---|---|---|---|
| Boolean | character | signed | unsigned | enum<i> |
| _Bool | char | signed char<br>signed short<br>signed int<br>signed long<br>signed long long | unsigned char<br>unsigned short<br>unsigned int<br>unsigned long<br>unsigned long long | named enum |

| *Essential type category* | |
|---|---|
| floating | |
| real floating | complex floating |
| float<br>double<br>long double | float _Complex<br>double _Complex<br>long double _Complex |

### 2.3.7 Amend Rule 10.1

#### Amendment

Add a restriction on the use of:

- expressions of *essentially complex floating* type
- relational operators to compare essentially real floating objects.

AMD3.20 : In the "Amplification" section, replace the table, as follows:

| Operator | Operand | Essential type category of arithmetic operand | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | Boolean | character | enum | signed | unsigned | floating | |
| | | | | | | | real | complex |
| `[ ]` | integer | 3 | 4 | | | | 1 | 9 |
| + (unary) | | 3 | 4 | 5 | | | | |
| – (unary) | | 3 | 4 | 5 | | 8 | | |
| `+ -` | either | 3 | | 5 | | | | |
| `++ --` | | 3 | | 5 | | | | 9 |
| `* /` | either | 3 | 4 | 5 | | | | |
| `%` | either | 3 | 4 | 5 | | | 1 | 9 |
| `< > <= >=` | either | 3 | | | | | | 9 |
| `== !=` | either | | | | | | 10 | 10 |
| `! && \|\|` | any | | 2 | 2 | 2 | 2 | 2 | 2 |
| `<< >>` | left | 3 | 4 | 5, 6 | 6 | | 1 | 9 |
| `<< >>` | right | 3 | 4 | 7 | 7 | | 1 | 9 |
| `~ & \| ^` | any | 3 | 4 | 5, 6 | 6 | | 1 | 9 |
| `?:` | 1st | | 2 | 2 | 2 | 2 | 2 | 2 |
| `?:` | 2nd and 3rd | | | | | | | |

AMD3.21 : In the "Amplification" section, delete the first note:

Under this rule ... operators.

AMD3.22 : In the "Rationale" section, add a new rationales 9 and 10:

9. The use of an expression of *essentially complex floating* type for these operands is a constraint violation.

10. The inherent nature of floating-point types is such that comparisons of equality will often not evaluate to true even when they are expected to. In addition the behaviour of such a comparison cannot be predicted before execution, and may well vary from one implementation to another. Deterministic floating-point comparisons should take into account the floating-point granularity (`FLT_EPSILON`) and the magnitude of the numbers being compared.

AMD3.23 : In the "Exception" section, number the existing exception as Exception 1, and add a new Exception 2:

2. Comparison (for equality or inequality) of *essentially real floating* or *essentially complex floating* objects with the constant literal value of zero or the macro `INFINITY` or `-INFINITY` is permitted.

AMD3.24 : In the "Examples" section, replace the comment:

Compliant by exception

with

Compliant by exception 1

AMD3.25 : In the "Examples" section, append the following new examples:

The following examples demonstrate the scope of Rationale 10:

```
float32_t f32w = -1.0f;
float32_t f32x =  0.2f;
float32_t f32y =  0.9f;
float32_t f32z =  0.0f;

if ( ( f32w + f32x ) == f32y ) /* Non-compliant                          */
if ( f32w != f32y )            /* Non-compliant                          */
if ( f32w <= f32y )            /* Compliant                              */

if ( f32w != 0.0f )            /* Compliant     - Exception 2            */
if ( f32w != f32z )            /* Non-compliant - not a literal constant */
```

AMD3.26 : Add a "See also" list:

### See also

Dir 4.15

## 2.3.8 Amend Rule 10.3

### Amendment

Add an Exception to allow the assignment of an *essentially real floating* expression to an object of *essentially complex floating type* when appropriate.

AMD3.27 : In the "Exception" section, add a new exception 4:

4. An *essentially real floating* expression may be assigned to an object of *essentially complex floating type* provided that its *corresponding real type* is not narrower than the type of the expression.

AMD3.28 : In the "Example" section, in the group of compliant examples, add:

```
cf32a = f32a;    /* By exception 4 */
cf64a = f64a;    /* By exception 4 */
```

AMD3.29 : In the "Example" section, in the group of non-compliant incompatible examples, add:

```
f32a = cf32a;    /* real floating and complex floating */
f64a = cf64a;    /* real floating and complex floating */
```

AMD3.30 : In the "Example" section, in the group of non-compliant narrowing examples, add:

```
cf32a = f64a;    /* complex floating and real floating */
```

### 2.3.9 Amend Rule 10.4

#### Amendment

Add an Exception to allow operations between *essentially real floating* and *essentially complex floating* objects.

AMD3.31 : In the "Exception" section, after the existing exception 2, add a new paragraph and exception 3:

Operations involving mixed real and complex operands have well-defined semantics and are therefore permitted:

3. The operators covered by this rule may have one operand with *essentially real floating* type and the other operand with *essentially complex floating* type. In the case of the conditional operator, this exception applies to the second and third operands.

AMD3.32 : In the "Example" section, add the following::

The following is compliant by exception 3.

```
cf32a += f32a;    /* complex floating and real floating */
```

### 2.3.10 Amend Rule 10.5

#### Amendment

Add restriction on the use of casting to and from *essentially complex floating* objects.

AMD3.33 : In the "Amplification" section, replace the table, as follows:

| Essential type category | from | | | | | | |
|---|---|---|---|---|---|---|---|
| to | *Boolean* | *character* | *enum* | *signed* | *unsigned* | *real floating* | *complex floating* |
| *Boolean* | - | Avoid | Avoid | Avoid | Avoid | Avoid | Avoid |
| *character* | Avoid | - | | | | Avoid | Avoid |
| *enum* | Avoid | Avoid | Avoid$^{*}$ | Avoid | Avoid | Avoid | Avoid |
| *signed* | Avoid | | | - | | | |
| *unsigned* | Avoid | | | | - | | |
| *real floating* | Avoid | Avoid | | | | - | |
| *complex floating* | Avoid | Avoid | | | | | - |

### 2.3.11 Amend Rule 10.7

#### Amendment

Add clarification, permitting coexistence of *essentially real floating* and *essentially complex floating* operations.

AMD3.34 : In the "Rationale" section, replace the second paragraph, as follows:

Restricting implicit conversions on *composite expressions* means that sequences of arithmetic operations within an expression must be conducted in the same *essential type*, with the exception that *essentially real floating* and *essentially complex floating* operations may coexist. This reduces possible developer confusion.

### 2.3.12 Amend Rule 10.8

#### Amendment

Add Exceptions to allow casting between *essentially real floating* and *essentially complex floating* types.

AMD3.35 : Add a new "Exception" section:

Exception
1. An *essentially real floating* expression may be cast to an *essentially complex floating* type providing that the corresponding real type is not wider than the type of the expression.
2. An *essentially real complex* expression may be cast to *essentially real floating* type providing that type is not wider than the corresponding real type of the expression.

### 2.3.13 New Rules 17.9-17.11 — Use of _Noreturn

#### Amendment

Permit the use of `_Noreturn` in a controlled manner.

AMD3.36 : Add the following three new rules after Rule 17.8:

**Rule 17.9 A function declared with a _*Noreturn* function specifier shall not return to its caller**

C11 [Undefined 71]

| | |
|---|---|
| Category | Mandatory |
| Analysis | Undecidable, System |
| Applies to | C11 |

#### Rationale

By definition, use of the _*Noreturn* function specifier indicates unambiguously that a function is not expected to return to its caller, by any path.

Returning from such a function is indicative of an error in the program's control flow, resulting in undefined behaviour.

#### Example

```
_Noreturn void a ( void )
{
  return; /* Non-compliant - breaks _Noreturn contract */
}
```

```
_Noreturn void b ( void )
{
  while ( true ) /* Permitted by Rule 14.3 Exception 1 */
  {
    ...
  }

  /* Compliant - function can never return */
}

_Noreturn void c ( void )
{
  abort();               /* Compliant - no return from abort(). */
}


_Noreturn void d ( int32_t i )
{
  if ( i > 0 )
  {
    abort();
  }

  /* Non-compliant - returns if i <= 0. */
}
```

### See also

Rule 17.11

## Rule 17.10 A function declared with a _*Noreturn* function specifier shall have *void* return type

| | |
|---|---|
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C11 |

### Rationale

By definition, use of the _*Noreturn* function specifier indicates unambiguously that a function will not return to its caller.

A function that cannot return cannot provide a return value and shall therefore be defined with a *void* return type.

### Example

```
_Noreturn int32_t f ( void );   /* Non-compliant */
_Noreturn void    g ( void );   /* Compliant     */
```

### See also

Rule 17.9, Rule 17.11

### Rule 17.11 A function that never returns should be declared with a *_Noreturn* function specifier

| | |
|---|---|
| Category | Advisory |
| Analysis | Undecidable, System |
| Applies to | C11 |

#### Rationale

Declaring a function that cannot return as *_Noreturn* highlights that this is "by design".

#### Exception

This rule does not apply to *main()*, as the C Standard states that it is a constraint violation for *main()* to be declared with the *_Noreturn* function specifier.

#### Example

```
void f ( void )  /* Non-compliant - exit() call means there is no return */
{
  exit ( 0 );
}
```

#### See also

Rule 17.9

## 2.3.14 New Rule 17.12 — Function identifiers

### Amendment

Add guidance on the use of function identifiers.

AMD3.37 : Add the following new rule after the new Rule 17.11:

### Rule 17.12 A function identifier should only be used with either a preceding `&`, or with a parenthesized parameter list

| | |
|---|---|
| Category | Advisory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99, C11 |

#### Rationale

A function identifier not preceded by `&` and not followed by a parenthesized parameter list (which may be empty) can be confusing: it may not be clear whether the intent is to call the function (but the parenthesized parameter list has accidentally been omitted) or the intent is to obtain the function address.

### Example

```
typedef int32_t (*pfn_i)(void);

extern  int32_t   func1 ( void );  /* Note: A function         */
extern  int32_t (*func2)( void );  /* Note: A function pointer */


void func ( void )
{
  pfn_i pfn1 = &func1;          /* Compliant                                */
  pfn_i pfn2 =  func1;          /* Non-compliant                            */

  int32_t i32a = (*pfn1)();     /* Compliant     - explicit call via a fn-pointer */
  pfn1();                       /* Compliant     - implicit call via a fn-pointer */


  if ( func1 == func2 )         /* Non-compliant                            */
  {
    /* ... */
  }

  if ( func1 ( ) == func2 ( ) ) /* Compliant     - comparing return values      */
  {
      /* ... */
  }

  if ( &func1 == func2 )        /* Compliant     - comparing a function's address */
  {
    /* ... */
  }
}
```

## 2.3.15 New Rule 17.13 — Function type qualifiers

### Amendment

Add guidance on the use of function type qualifiers.

AMD3.38 : Add the following new rule after the new Rule 17.12:

### Rule 17.13 A *function type* shall not be type qualified

C90 [Undefined *], C99 [Undefined 63], C11 [Undefined 66]

| | |
|---|---|
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99, C11 |

### Amplification

The type qualifiers are *const*, *volatile*, *restrict* or *_Atomic*.

### Rationale

The behaviour is undefined if the specification of a *function type* includes any type qualifiers.

Note: this rule applies to a *function type* and not to the return type of a function.

### Example

```
const uint16_t    cf (void);          /* Compliant     - returns const uint16_t    */
const uint16_t * pcf (void);          /* Compliant     - returns a pointer to
                                                              const uint16_t */
```

In the following examples, `ftype` is the type of a function returning `uint16_t`:

```
typedef uint16_t ftype (void);

typedef const ftype          cftype; /* Non-compliant - cftype is const-qualified */

typedef       ftype          dftype; /* Compliant     - dftype is not qualified   */
typedef       ftype * const pcftype; /* Compliant     - const pointer to ftype    */

typedef const uint16_t * cfptype (void); /* Compliant - cfptype is the type of a
                                                        function returning
                                                                 const uint16_t * */
```

## 2.3.16 New Rule 18.9 — Object lifetime

### Amendment

Add guidance on the lifetime of objects.

AMD3.39 : Add the following new rule after Rule 18.8:

**Rule 18.9** An object with *temporary lifetime* shall not undergo array-to-pointer conversion

C99 [Undefined 8, 35], C11 [Undefined 9]

| | |
|---|---|
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99, C11 |

### Amplification

*Temporary lifetime* is a storage duration which describes the lifetime of the elements of non-*lvalue* arrays.

An array which is a member of a non-*lvalue* expression with structure or union type shall not be used as a value, other than as the immediate *postfix-expression* operand to a subscript operator.

The subscript operator shall not be used to produce a modifiable *lvalue*.

### Rationale

An array object can be a member of a structure or union, and therefore form part of the result value of any value expression. Because arrays used in an expression are always value-converted to a pointer to their elements, it is possible to form a pointer to such array sub-objects even when they are not part of a declared object with normal lifetime.

Modifying elements of such an array implicitly results in undefined behaviour in C90, and explicitly results in undefined behaviour in C99 and later. Accessing the elements of such an array after the end of its lifetime results in undefined behaviour, which is implicitly limited to the next sequence point in C90 and C99 (meaning the pointer cannot be stored or passed to any function), and to the duration of the complete containing expression in C11 and later.

The pointer will not be to `const`-qualified elements unless the array element type is `const`-qualified; therefore the array's type may appear to allow modification, and the generation of apparently modifiable element *lvalues*, even though the array itself has *temporary lifetime*.

Since the containing object can always be assigned to a named intermediate object by value, there is little reason to ever attempt to take the address of the value with temporary lifetime.

#### Example

```
/* Value object containing an array as an element */

struct S1 {
  int32_t array[10];
};

struct S1 s1;
struct S1 getS1 (void);

void foo(int32_t const * p);


/* Compliant - not temporary storage duration */

int32_t * p = s1.array;
s1.array[0] = 1;
foo( s1.array );


/* Non-compliant - temporary storage duration */

p = getS1().array;              /* also creates dangling pointer        */
foo( getS1().array );
foo( (s1 = s1).array );         /* other forms of non-lvalue expression */


/* Compliant - immediate element access is always safe */

int32_t j = getS1().array[3];   /* element copied: const access         */
j = (s1 = s1).array[3];


/* Non-compliant - element used as a modifiable lvalue */

getS1().array[3] = j;
(1 ? s1 : s1).array[3] = j;
```

### 2.3.17 Amend Rule 21.11 — Type-generic macros

#### Amendment

Add cross-references to the additional guidance on the use of the features defined within `<tgmath.h>`, and recategorize now that additional guidance is available.

AMD3.40 : Replace the rule headline as follows:

The standard *header file* `<tgmath.h>` should not be used

AMD3.41 : Recategorize the Rule as *Advisory* (was *Required*).

AMD3.42 : Replace the rule amplification as follows:

The standard *header file* `<tgmath.h>` should not be #included.

Note: Due to the duplication of macro names between `<tgmath.h>`, `<math.h>` and `<complex.h>` this rule does not have the additional requirement that *none of the features that are specified as being provided by `<tgmath.h>` should be used*, as use by means of #including either of these other standard *header files* is not constrained. Any other definition of a macro specified as being provided by `<tgmath.h>` will be a violation of Rule 21.1 and/or Rule 21.2.

AMD3.43 : Add a "See also" list:

### See also

Rule 21.1, Rule 21.2, Rule 21.22, Rule 21.23

## 2.3.18 Amend Rule 21.12 — Floating-point environment

### Amendment

Extend Rule 21.12 to include all features of `<fenv.h>`.

AMD3.44 : Replace the rule headline as follows:

The standard *header file* `<fenv.h>` shall not be used.

AMD3.45 : Revise the source references to add C99 [Undefined 112] and C11 [Undefined 118].

AMD3.46 : Recategorize the Rule as *Required* (was *Advisory*).

AMD3.47 : Replace the rule amplification as follows:

The standard *header file* `<fenv.h>` shall not be #included, and none of the features that are specified as being provided by `<fenv.h>` shall be used.

AMD3.48 : Extend the rationale, by adding the following to the end of the current rationale:

Calling the *fesetenv* or *feupdateenv* functions with invalid arguments results in *undefined behaviour*.

Calling the *fesetround* function should be done with care because:

1. Setting the rounding mode may have unexpected consequences, e.g. setting the current rounding mode to upwards does not guarantee that the result of evaluating an expression is an upward approximation to the value of the expression over the reals.

2. Several implementations of functions declared in `<math.h>` have been designed to support round-to-nearest only: if such functions are called when a different rounding mode is set, the results can be unpredictable.

Note: In *conforming implementations*, the rounding direction mode is set to rounding to nearest at program start-up.

## 2.3.19 New Rules 21.22 and 21.23 — Type-generic macros

### Amendment

Add guidance on the use of the *type-generic macros* declared in `<tgmath.h>`.

AMD3.49 : Add the following two new rules after Rule 21.21 (added by MISRA C:2012 Amendment 2 [6])

## Rule 21.22 All operand arguments to any *type-generic macros* declared in `<tgmath.h>` shall have an appropriate *essential type*

C99 [Undefined 184], C11 [Undefined 195]

| | |
|---|---|
| Category | Mandatory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C99, C11 |

### Amplification

The operand arguments passed to the *type-generic* macros defined in `<tgmath.h>` shall have *essentially signed*, *essentially unsigned* or *essentially floating* (either *essentially real floating* or *essentially complex floating*) type.

Arguments to the following macros shall not have *essentially complex floating* type:

*atan2*, *cbrt*, *ceil*, *copysign*, *erf*, *erfc*, *exp2*, *expm1*, *fdim*, *floor*, *fma*, *fmax*, *fmin*, *fmod*, *frexp*, *hypot*, *ilogb*, *ldexp*, *lgamma*, *llrint*, *llround*, *log10*, *log1p*, *log2*, *logb*, *lrint*, *lround*, *nearbyint*, *nextafter*, *nexttoward*, *remainder*, *remquo*, *rint*, *round*, *scalbn*, *scalbln*, *tgamma*, *trunc*.

Note: The final parameter to the *frexp* and *remquo* macros is for output, and is not considered an operand.

### Rationale

Arguments of non-arithmetic types are not convertible to any of the *corresponding real types* defined for the macros defined in `<tgmath.h>`. Attempting to use them therefore results in undefined behaviour.

Casting an argument with *essentially signed* or *essentially unsigned* type to an *essentially real floating* type is not required because the purpose of these macros is to be type-generic. The essential type of the parameter derives from the argument.

Passing an *essentially complex floating* argument to one of the macros listed in the amplification results in undefined behaviour.

### Example

Real-valued arguments must have *essentially signed* or *essentially unsigned* or *essentially floating* type:

```
float   f1,  f2;
int     i1,  i2;

char    c1,  c2;
void   *p1, *p2;

void fn1 (void)
{
    f2 = sqrt (f1);    /* Compliant     - essentially floating real type  */
    i2 = sqrt (i1);    /* Compliant     - essentially integer real type   */

    c2 = sqrt (c1);    /* Non-compliant - essentially character real type */

    p2 = sqrt (p1);    /* Non-compliant - undefined behaviour             */
}
```

Arguments to the macros listed in the amplification must have a real type:

```
         float  f1,  f2;
_Complex float cf1, cf2;

void fn2 (void)
{
     f2 = sqrt ( f1);  /* Compliant     - real argument               */
    cf2 = sqrt (cf1);  /* Compliant     - sqrt has a complex equivalent */

     f2 = ceil ( f1);  /* Compliant     - real argument               */
    cf2 = ceil (cf1);  /* Non-compliant - undefined behaviour         */
}
```

## See also

Rule 21.11, Rule 21.23

## Rule 21.23 All operand arguments to any multi-argument *type-generic macros* declared in `<tgmath.h>` shall have the same standard type

| | |
|---|---|
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C99, C11 |

### Amplification

This rule applies to the following multi-argument *type-generic macros*:

*atan2*, *copysign*, *fdim*, *fma*, *fmax*, *fmin*, *fmod*, *frexp*, *hypot*, *ldexp*, *nextafter*, *nexttoward*, *pow*, *remainder*, *remquo*, *scalbn*, *scalbln*

All operand arguments passed to any of the multi-argument macros defined in `<tgmath.h>` shall have the same standard type, after integer promotion has been applied to any integer arguments.

Note: The final parameter to the *frexp* and *remquo* macros is for output, and is not considered an operand.

### Rationale

Ensuring that the types used to determine the *corresponding real type* for the whole call expression are consistent makes the relationship between the input values and the result clearer.

On platforms with extended real types, the *essentially real floating* types may not be able to be strictly ordered by precision. In this case there is a risk that if two arguments have different types, the deduced common real type may be an unexpected size or silently lose precision.

### Example

```
void f (void) {
    float32_t f1;
    float32_t f2;
    float64_t d1;
    float64_t d2;

    f2 = pow(f1, f2);            /* Compliant                                        */
    d2 = pow(d1, d2);            /* Compliant                                        */

    f2 = pow(f1, d2);            /* Non-compliant - unclear which argument was
                                                    intended to control precision */
    f2 = pow(f1, (float32_t)d2); /* Compliant                                        */
}

void g (void) {
    short s16;
    int   i32;
    long  l32;

    i32 = pow( s16, i32 );       /* Compliant     - both arguments are int after
                                                    integer promotion              */
    i32 - pow( i32, l32 );       /* Non-compliant - arguments are not the same
                                                    type after promotion           */

    i32 = pow( s16, 10 );        /* Compliant     - 10 has literal type int        */
    i32 = pow( 10u, 10ul );      /* Non-compliant - literal types unsigned int
                                                    and unsigned long              */
}
```

### See also

Rule 21.11, Rule 21.22

## 2.3.20 New Rule 21.24 — C random numbers

### Amendment

Add restrictions on the use of the random number functions of `<stdlib.h>`.

AMD3.50 : Add the following new rule after new Rule 21.23

**Rule 21.24 The random number generator functions of `<stdlib.h>` shall not be used**

| | |
|---|---|
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99, C11 |

### Amplification

The functions `rand` and `srand` shall not be used and no macro with one of these names shall be expanded.

### Rationale

The C Standard Library function `rand()` makes no guarantees as to the quality of the random sequence produced.

The C Standard warns that the numbers generated by some implementations of the `rand()` function have a short cycle and the results can be predictable. Applications with particular requirements should use a generator that is known to be sufficient for their needs.

The `srand()` function is also included within this rule, as without any associated use of the `rand()` function, its use is superfluous.

#### Example

```
#include <stdlib.h>

int r = rand();     /* Non-compliant */
```

### 2.3.21 Create new section 8.23 — Generic selections

#### Amendment

Add new section 8.23 and associated rules.

AMD3.51 : Add new Section 8.23 for generic selections

### 8.23 Generic selections

#### Rule 23.1 A generic selection should only be expanded from a macro

| | |
|---|---|
| Category | Advisory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C11 |

#### Amplification

A generic selection expression should only be expanded from a function-like macro, with one of the macro's arguments expanded into the controlling expression of the generic selection.

#### Rationale

Generic selections are useful to implement type queries or checks, or generic functions. If the generic selection is applied directly rather than expanded from a macro body, the type of the operand is already known locally and querying it is not necessary.

#### Example

The following example is non-compliant, as it is not a macro.

```
int32_t x = 0;

/* Non-compliant - not a macro */
int32_t y = _Generic( x
          , int32_t   : 1
          , float32_t : 2 );
```

The following example implements a regular generic function, and can be expanded into contexts where the operand type may vary.

```
/* Compliant - used to implement a generic function */
#define arith(X) ( _Generic( (X)                      \
                 , int32_t   : handle_int32          \
                 , float32_t : handle_float          \
                 , default   : handle_any   ) (X) )
```

The following example is not compliant because it expands to a generic selection which does not depend on the type of an argument.

```
/* Non-compliant */
#define maybe_inc(Y) ( _Generic( x                   \
                     , int32_t : 1                   \
                     , default : 0 ) + (Y) )
```

### See also

Dir 4.9, Rule 13.6, Rule 23.2

## Rule 23.2 A generic selection that is not expanded from a macro shall not contain potential *side effects* in the controlling expression

| | |
|---|---|
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C11 |

### Amplification

If the controlling expression of a generic selection is not expanded from a macro argument, it shall not appear to contain any *side effects*.

A function call is considered to be a *side effect* for the purposes of this rule.

### Rationale

The controlling expression of a generic selection is never evaluated. It is only used for its type, and usually has to be repeated in order to be combined with the result value in some way.

If an expression is specified which syntactically contains a side effect, that effect will not be applied, even if the expression has a type that would cause it to be evaluated by `sizeof`, such as a VLA.

### Example

```
#ifdef BIG
typedef int64_t STATE;
#else
typedef int16_t STATE;
#endif

STATE shared_state;

/* Compliant */
_Static_assert ( _Generic ( shared_state
               , int32_t: 0
               , default: 1 )
               , "error on wrong type");
```

This example is non-compliant because the apparent side effect modifying `shared_state` is not evaluated by the controlling expression.

```
/* Non-compliant */
_Static_assert ( _Generic ( ++shared_state
               , int32_t: 0
               , default: 1)
               , "error on wrong type");
```

### See also

Rule 13.6, Rule 23.1, Rule 23.7

## Rule 23.3 A generic selection should contain at least one non-default association

| | |
|---|---|
| Category | Advisory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C11 |

### Amplification

A generic selection should contain at least one association which explicitly specifies an object type. The presence of a `default` association is always optional.

### Rationale

A generic selection that consists only of a usable `default` association does not do anything useful; the default association's result value is evaluated unconditionally.

Omitting a default association indicates intent to check the operand type, by introducing a constraint violation when the type does not match.

### Example

```
/* Non-compliant - consists only of a default association */
#define no_op(X) _Generic( (X), default: (X) )

/* Compliant - has a non-default and a default association */
#define filter_ints(X)  ( _Generic( (X)                          \
                        , int32_t: handle_int                    \
                        , default: handle_numeric_value ) (X) )

/* Compliant - has non-default associations */
#define only_ints(X)  ( _Generic( (X)                \
                      ,  int32_t: handle_int         \
                      , uint32_t: handle_int ) (X) )
```

The following example demonstrates that, by omitting a default association, there is a clear intent to check the operand type, by introducing a constraint violation when the type does not match.

```
/* Compliant - it has a single permitted type and is
               intended to prevent implicit conversion */
#define require_char(X) ( _Generic ( (X), char8_t: (X) ) )
```

### See also

Rule 23.4, Rule 23.8

## Rule 23.4 A generic association shall list an appropriate type

| | |
|---|---|
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C11 |

### Amplification

The controlling expression of a generic selection undergoes *lvalue conversion* before having its type compared to the entries in the association list. The association list shall not contain any associations for:

- a `const`-qualified object type
- a `volatile`-qualified object type
- an atomic object type
- an array type
- a function type
- an unnamed structure or union definition

### Rationale

Since the Standard does not impose a constraint limiting the types in the association list to the possible types of values after conversion, it is possible to list associations for types that would never be eligible for selection.

Value conversion removes top-level qualification and "decays" array and function values into pointers. Therefore, such types can never match the type of the converted value of the controlling expression. Listing them is not a constraint violation, but serves no useful purpose, and is almost certainly an error. This only affects object types, not qualification on pointed-to types.

Listing an unnamed structure or union in the association list is a violation of this rule because every occurrence of a *struct-declaration-list* creates a distinct type, therefore it can only be *matched* by a default association.

### Example

In this set of examples, the first pair are non-compliant because they explicitly specify generic associations for types that can never be the type of the controlling expression, while the second pair are compliant as they only attempt to specify generic associations for types that can match the controlling expression.

```
typedef int32_t Func (int);
typedef int32_t Array [10];

typedef Func  * FuncP;
typedef Array * ArrayP;


/* Non-compliant (because Func is listed) */
#define handle_function_nc(X) _Generic((X)                   \
                              , Func : handle_funcp (&(X))    \
                              , FuncP: handle_funcp   (X)   )
```

```
/* Non-compliant (because Array is listed) */
#define handle_array_nc(X) _Generic((X)                     \
                           , Array    : handle_intp ((X) + 0) \
                           , ArrayP   : handle_intp (*(X))    \
                           , int32_t *: handle_intp (X) )


/* Compliant */
#define handle_function(X) _Generic( (X), FuncP: handle_funcp (X) )


/* Compliant */
#define handle_array(X) _Generic(&(X)[0]                     \
                        , int32_t *: handle_intp (X)          \
                        , ArrayP   : handle_intp (*(X)) )

Array arr1;
Array arr2[10];      /* Two-dimensional - array of arrays                        */

handle_array (arr1); /* type of &(arr1[0]) is int32_t * - can pass to handle_intp */
handle_array (arr2); /* type of &(arr2[0]) is ArrayP    - must dereference again  */
```

In this set of examples, the first example is non-compliant because it explicitly specifies a type that includes qualifiers at the object level; qualification is removed by value conversion, so such types are not selectable. The second example is able to take qualification into account because the qualifiers apply to the pointed-to type, not the object type, and are not removed from the type of the value of the controlling expression.

```
typedef int32_t Int;
typedef int32_t const CInt;


/* Non-compliant */
#define filter_const_nc(X) ( _Generic((X)                          \
                           , CInt    : handle_const_intp            \
                           , default: handle_other_value ) (&(X)) )


/* Compliant */
#define filter_const(X) ( _Generic((X)  \
                        , CInt * : handle_const_intp                \
                        , Int *  : handle_const_intp                \
                        , default: handle_other_value ) (X) )
```

In this set of examples, the first example is non-compliant because it tries to explicitly specify array types, which are not selectable, in an attempt to involve dependent type information in the match. The second example below is compliant because it explicitly specifies a pointer type, and forces the type of the controlling expression to be a pointer with &. Creating a pointer to an array preserves the complete array type, including its dimension.

```
/* Avoid multiple associations if size == 1 */
#define SizeofNonEmpty(A) (sizeof (A) > 1 ? sizeof (A) : 2)


/* Non-compliant - tries to match argument array type */
#define only_strings_nc(X) _Generic((X)                              \
  , char[SizeofNonEmpty (X)]: handle_sized_string (sizeof (X), (X))  \
  , char[1]                 : handle_null_terminator (1, (X)))
```

```
/* Compliant - matches pointer to argument array type */
#define only_strings(X) _Generic(&(X)                                  \
  , char (*) [SizeofNonEmpty (X)]: handle_sized_string (sizeof (X), (X))  \
  , char (*) [1]: handle_null_terminator (1, (X)))

only_strings_nc ("hello"); /* Constraint violation     */
only_strings    ("world"); /* Matches as char (*) [6]  */
```

## See also

Rule 23.5

## Rule 23.5 A generic selection should not depend on implicit pointer type conversion

| | |
|---|---|
| Category | Advisory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C11 |

### Amplification

This rule only applies to selection based on pointer types. Selection based on an arithmetic type is never a violation of this rule.

This rule applies when a default association is selected for a controlling expression whose type could, in other contexts (such as when passed as an argument to a function), be implicitly converted to another type explicitly listed in the association list.

### Rationale

The controlling expression of a generic selection undergoes value conversion before having its type compared to any of the entries in the association list, but this is the only conversion applied to its type. It can then only match an association exactly, or select the default association.

No attempt is made to implicitly convert the type to match an association.

For instance, listing an association for `void *` will never match a pointer to a complete object type, and listing an association for `T const *` will never match `T *`. This is different from function arguments, and may therefore cause surprise for users of a generic function implemented as a generic association, if a type that would be implicitly convertible to an explicitly-listed type is instead allowed to fall through to the default association.

### Example

Given:

```
void handle_pi  (      int32_t *);   /* No associations have this signature */
void handle_cpi (const int32_t *);
void handle_any (void          *);
```

Using the following generic function with a pointer to `const int32_t` is compliant, but using a pointer to (mutable) `int32_t` is not, because there is no implicit conversion to qualified-pointer in generic matching:

```
#define handle_pointer1(X) ( _Generic ((X)                        \
                           , const int32_t *: handle_cpi          \
                           , default        : handle_any) (X) )

const int32_t ci;
handle_pointer1 (&ci);     /* Compliant     - const int32_t * is selected */

int32_t mi;
handle_pointer1 (&mi);     /* Non-compliant - default is selected
                                              NOT const int32_t *         */
```

Using the following generic function with a (mutable) pointer to `int32_t` is non-compliant, as there is no conversion to qualified pointer or to pointer to `void` in generic matching.

```
#define handle_pointer2(X) ( _Generic ((X)                        \
                           , void          *: handle_any          \
                           , const int32_t *: handle_cpi          \
                           , default        : handle_any) (X))

handle_pointer2 (&mi);     /* Non-compliant - default is selected
                                              NOT void * or const int32_t * */
```

A generic function that wishes to handle pointers to any object type and distinguish them by qualification, can wrap the controlling expression such that the pointed-to type is explicitly converted to a pointer to `void`:

```
void handle_unq (void                *);
void handle_cq  (void const          *);
void handle_vq  (void       volatile *);
void handle_cvq (void const volatile *);

#define handle_pointer(X)  ( _Generic (1 ? (X) : (void *)(X)          \
                           , void                * : handle_unq       \
                           , void const          * : handle_cq        \
                           , void       volatile * : handle_vq        \
                           , void const volatile * : handle_cvq) (X) )
/* Uses are always compliant */
```

Per the specification of the conditional operator's composite result type, any argument to the macro with a pointer to object type will be converted to a similarly-qualified pointer to `void` before it is used as the controlling expression.

### See also

Rule 23.3, Rule 23.4, Rule 23.6

## Rule 23.6 The controlling expression of a generic selection shall have an *essential type* that matches its standard type

| | |
|---|---|
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C11 |

### Amplification

Using an expression that has a distinct essential type from its standard type as the controlling expression of a generic selection is always a violation of this rule, except for integer constant expressions that are neither character constants nor essentially boolean.

Notes:

1. An enumeration constant unconditionally has the type `signed int` and will not match either the implementation defined underlying type, if different from `int`, or the (indistinguishable) enumerated type, whichever is listed.
2. The same consideration applies to `true` and `false` defined in `<stdbool.h>`, which have type `int` and will not match an association of type `_Bool`.

### Rationale

The association selected by a generic selection must be in line with the type system under which the code was designed, in order to be useful and consistent. Code written under MISRA guidelines is subject to the essential type system, and therefore if the selected type is not consistent with the essential type of the controlling expression, the generic selection will have violated the type system used for the design.

Because generic selections choose a standard type by name, they can expose this difference if used with an argument that has inconsistent essential and standard types.

### Exception

This rule does not apply to integer constant expressions which would have an essentially signed or unsigned type of lower rank than `int`, because of the *Type of Lowest Rank* rule, but are neither character constants nor essentially boolean.

### Example

The non-compliant examples below accept an operand with essential type of `signed short` and `char` respectively, but both have a standard type of `signed int`, and only the standard type determines which association will match.

An exception is made for integer constant expressions that are not character constants, because this is in line with most user expectations.

```
#define filter_ints(X) ( _Generic((X)                    \
                       ,   signed short: handle_sshort     \
                       , unsigned short: handle_ushort     \
                       ,   signed int  : handle_sint       \
                       , unsigned int  : handle_uint       \
                       ,   signed long : handle_slong      \
                       , unsigned long : handle_ulong      \
                       , default       : handle_any)  (X)  )


short s16 = 0;
int   i32 = 0;
long  l32 = 0;

/* Non-compliant */
filter_ints (s16 + s16);  /* Selects int, essentially short */
filter_ints ('c');        /* Selects int, essentially char  */

/* Compliant */
filter_ints (s16);
filter_ints (i32);
filter_ints (l32);

/* Compliant by exception */
filter_ints (10u);        /* UTLR is unsigned char */
filter_ints (250 + 350);  /* STLR is signed short */
```

Because an enumerated type is compatible with its implementation-defined underlying type, it is not distinguishable from the underlying type by `_Generic` (although distinguishing different enumerated types is possible because this compatibility is not transitive).

Attempting to list both would be a constraint violation. Therefore, using an enumerated type as the controlling expression to a generic selection that lists the underlying type, or vice versa, is always a violation.

```
enum E { A = 0, B = 1, C = 2 };
enum E e = A;

/* Non-compliant, assuming compiler chooses 'unsigned int' as the underlying type */
filter_ints (e); /* selects 'unsigned int', essentially enum */
filter_ints (A); /* selects   'signed int', essentially enum */
```

### See also

Rule 23.5

## Rule 23.7 A generic selection that is expanded from a macro should evaluate its argument only once

| | |
|---|---|
| Category | Advisory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C11 |

### Amplification

If the controlling expression of a generic selection is expanded from a macro argument, it should also be expanded elsewhere in the macro body so that it is evaluated exactly once in the entire expanded macro body. The number of times that the expression is evaluated should not depend on which association is selected, and should be consistent for all associations in the generic selection.

This rule does not depend on the presence of side effects in the operand expression.

### Rationale

The controlling expression of a generic selection is never evaluated. It is only used for its type, and usually has to be repeated in order to be combined with the result value in some way.

If an expression is specified which syntactically contains a side effect, that effect will not be applied. If the generic selection is (as is usually the case) the result of a macro expansion which uses one of the macro's arguments to select the type, this non-evaluation is concealed from the invoking code.

Combining the input value with the result expression “outside” the generic selection (such as choosing a function to call as the result, rather than the complete function call) is more likely to guarantee consistent evaluation of the expanded operand.

### Exception

This rule is not violated if all result expressions for the generic selection are constant expressions, and the macro never expands the argument outside of the controlling expression. This allows for the implementation of type queries like `is_pointer_const`.

### Example

The following examples are consistent in their expansion of side effects for all associations. In the first example, the argument used for the controlling expression are expanded exactly once outside of the generic selection, so the generic selection does not prevent it from being evaluated. In the second example, the argument expands once each into the result expression of each association. It will be evaluated exactly once regardless of which generic association is *selected*.

```
/* Compliant */
#define gfun1(X)  ( _Generic((X)             \
                  , float32_t: fun1f         \
                  , float64_t: fun1          \
                  , default  : fun1l)  (X) )

#define gfun2(X)    _Generic((X)             \
                  , float32_t: fun2f  (X)    \
                  , float64_t: fun2   (X)    \
                  , default  : fun2l  (X) )
```

The following example is non-compliant because whether or not the macro argument is evaluated depends on whether the default association is *selected* or not. The default association does not evaluate it because it has no use for the value, but this dangerously assumes that there were no side effects in the expression.

```
/* Non-compliant */
#define gfun3(X)    _Generic((X)                 \
                  , float32_t : fun3f  (X)       \
                  , float64_t : fun3   (X)       \
                  , float128_t: fun3l  (X)       \
                  , default   : default_result )
```

The following example only expands a macro argument into the controlling expression, and all possible result expressions are integer constant expressions. This implements a type query that can be used for type checking.

```
/* Compliant by exception */
#define is_pointer_const(P) _Generic(1 ? (P) : (void *)(P)  \
                           , void const          *: 1       \
                           , void const volatile *: 1       \
                           , default              : 0 )

_Static_assert (is_pointer_const (pi), "must not be an out-parameter");
```

## See also

Rule 23.2

## Rule 23.8 A default association shall appear as either the first or the last association of a generic selection

| | |
|---|---|
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C11 |

### Amplification

A default association shall only appear as either the first generic association in the association list, or the last – this rule only applies when a generic selection contains a default association.

### Rationale

Having a default association appear as either the first or last in the association list makes it easy to locate, clarifying the intent of the selection structure.

## Example

```
/* Non-compliant - default is not first or last association */
#define sqrt(X)  ( _Generic((X)              \
                , float32_t : sqrtf         \
                , default   : sqrt          \
                , float128_t: sqrtl)  (X) )

/* Compliant - default is first association */
#define cbrt(X)  ( _Generic((X)              \
                , default   : cbrt          \
                , float32_t : cbrtf         \
                , float128_t: cbrtl)  (X) )

/* Compliant - no default */
#define assert_untyped_nonatomic(X)  ( _Generic((X)                              \
                                    , void                * : handle_ptr        \
                                    , void const          * : handle_ptr        \
                                    , void volatile       * : handle_ptr        \
                                    , void const volatile * : handle_ptr)  (X) )
```

## See also

Rule 16.5, Rule 23.3

## 2.4 Section 9 — References

### 2.4.1 Amend Reference 35 — Floating-point

#### Amendment

Replace the reference to IEEE 754 with IEC 60559.

AMD3.52 : Replace Reference 35 (IEEE 754) with:

IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems*, International Organization for Standardization, 1989

## 2.5 Appendix C — Type safety issues with C

### 2.5.1 Appendix C.1 — Type conversions

#### 2.5.1.1 Appendix C.1.3 — Concerns with conversions

#### Amendment

Add concern over loss of imaginary part.

AMD3.53 : After the first bullet point ("Loss of value") insert:

- Loss of imaginary part: e.g. conversion of a complex floating type to a real floating type, where the imaginary part of the complex value is discarded

AMD3.54 : After the final bullet point ("Conversion of a floating type ...") insert:

- Conversion of a real floating type to a complex floating type, having wider or equal *corresponding real type*.

## 2.6 Appendix D — The essential type model

### Amendment

Address complex floating-point expressions.

AMD3.55 : Append a new bullet point, as follows:

- **Complex floating-point expressions** There is no promotion from real floating to complex floating for the operands of a complex expression.

### 2.6.1 Appendix D.1 — The essential type category of expressions

### Amendment

Expand the *essential type model* to include complex types.

AMD3.56 : In the list of *essential type categories*, replace:

- *Essentially floating*.

With:

- *Essentially floating*, which may be either:
  - *Essentially real floating*, or
  - *Essentially complex floating*.

Note: For the purposes of this *essential type model*, *essentially real floating* and *essentially complex floating* are considered to be distinct *essential type categories*.

AMD3.57 : Replace the table showing the essential types with:

| *Essential type category* | | | | |
|---|---|---|---|---|
| Boolean | character | signed | unsigned | enum<i> |
| _Bool | char | signed char<br>signed short<br>signed int<br>signed long<br>signed long long | unsigned char<br>unsigned short<br>unsigned int<br>unsigned long<br>unsigned long long | named enum |

| *Essential type category* | |
|---|---|
| floating | |
| real floating | complex floating |
| float<br>double<br>long double | float _Complex<br>double _Complex<br>long double _Complex |

### 2.6.2 Appendix D.7.12 — Generic selection ( _Generic )

Notes:

1. Appendix D.7.12 was introduced by MISRA C:2012 Amendment 2 [6].

2. MISRA C:2012 Technical Corrigendum 2 [4] combined the previously existing sections D.7.9, D.7.10 and D.7.11 into a single D.7.9, but did not renumber the subsequent D.7.12.

AMD3.58 : Renumber appendix D.7.12 as D.7.10

AMD3.59 : Replace:

The *essential type* is the same as the *essential type* of the result expression.

with

The *essential type* of a `_Generic` selection is the *essential type* of the selected expression.

## 2.7 Appendix F — Obsolescent language features

Note: Appendix F was removed by MISRA C:2012 Amendment 2 [6] and replaced with a placeholder.

AMD3.60 : Replace the existing Appendix F placeholder (introduced by MISRA C:2012 Amendment 2 [6]) as follows:

### Appendix F — Obsolescent language features

Obsolescent language features are those identified in the *Future language directions* and *Future library directions* sections of the applicable C Standard — this appendix should be used in conjunction with Rule 1.5.

In the following table:

- **ID** is a unique MISRA sequential identifier
- **Obsolescent Feature** identifies the language feature that has been declared as obsolescent.
- An X in the **C99**, **C11** or **C18** denotes that the specified feature has been declared as obsolescent in the applicable version of the C Standard.
- **Comments** identifies other MISRA C guidelines for which the use of the specified feature should also be considered a violation, or specifies a note.

| ID | Obsolescent Feature | C99 | C11 | C18 | Comments |
|---|---|---|---|---|---|
| 1 | Declaring an identifier with internal linkage at file scope without the static storage class specifier is an obsolescent feature. | X | X | X | Rule 8.8 |
| 2 | Restriction of the significance of an external name to fewer than 255 characters [...] is an obsolescent feature. | X | X | X | Note 1 |
| 3 | The placement of a storage-class specifier other than at the beginning of the declaration specifiers in a declaration is an obsolescent feature. | X | X | X | - |
| 4 | The use of function declarators with empty parentheses (not prototype-format parameter type declarators) is an obsolescent feature. | X | X | X | Rule 8.2 |
| 5 | The use of function definitions with separate parameter identifier and declaration lists (not prototype-format parameter type and identifier declarators) is an obsolescent feature. | X | X | X | Rule 8.2 |
| 6 | The macro ATOMIC_VAR_INIT is an obsolescent feature. | - | - | X | Rule 1.4 |
| 7 | The ability to undefine and perhaps then redefine the macros bool, true, and false is an obsolescent feature. | X | X | X | Rule 21.1 |
| 8 | The gets function is obsolescent, and is deprecated. | X | - | - | Rule 21.6, Note 2 |
| 9 | The use of ungetc on a binary stream where the file position indicator is zero prior to the call is an obsolescent feature. | X | X | X | Rule 21.6 |
| 10 | Invoking realloc with a size argument equal to zero is an obsolescent feature. | - | - | X | Rule 21.3 |

Notes:

1. This is a feature of the implementation and is not (usually) determinable by a static analysis tool.
2. The `gets` function was removed from the C Standard in C11.

## 2.8 Appendix J — Glossary

### Amendment

Insert the following new definitions, in the appropriate (alphabetical) order:

AMD3.61 : Insert new *generic*-related definitions:

#### 2.8.1.1 Generic function

The term *generic function* is defined by the C Standard to mean a callable entity that can handle operands of different types in appropriately different ways (i.e. “ad-hoc polymorphism”).

The exact details of its implementation are left intentionally unspecified in order to provide maximum latitude to implementations. It is not required that any *generic functions* in the C Standard library are actually implemented using generic selections, although generic selections provide an in-language mechanism for doing so; compiler intrinsic features are another possible mechanism.

Other properties of the entity representing a *generic function* are also unspecified. It is not likely that a *generic function* has a single address that can be taken, for instance.

#### 2.8.1.2 Generic match

A type *matches* if it is compatible, without undergoing any implicit conversion, with the type of the value of the controlling expression. At most one type listed in the selection may *match*; the generic association specifying that type will be *selected*.

If no type *matches*, the default association will be *selected* if present.

#### 2.8.1.3 Generic selection

A generic association is *selected* if the type specified is compatible with the type of the value of the controlling expression. If no explicitly-specified type is compatible with the type of the value of the controlling expression, the default association is *selected* if present. Otherwise, if no association is *selected*, the generic selection violates the constraint that exactly one association must be *selected* by the generic selection expression.

The result expression of the *selected* association becomes the result of the complete generic selection expression.

AMD3.62 : Insert a new definition for *type-punning*:

#### 2.8.1.4 Type-punning

*Type-punning* is when an object is referred to using different types.

For example, writing data to one member of a union and reading it from another

# 3 Consequential amendments

## 3.1 Consequential amendments to other existing guidelines

### 3.1.1 Obsolescent features

#### Amendment

Add cross-references to new Rule 1.5 for guidelines already addressing obsolescent features.

AMD3.63 : For Rule 8.2, insert into "See also" list:

Rule 1.5,

AMD3.64 : For Rule 8.8, add a "See also" list:

#### See also

Rule 1.5

### 3.1.2 Consistency of terminology regarding header file inclusion

#### Amendment

Adopt revised terminology regarding header file inclusion for the other rules not already affected by this amendment.

AMD3.65 : For Rule 17.1, replace the Headline with:

The standard *header file* `<stdarg.h>` shall not be used

AMD3.66 : For Rule 17.1, replace the Amplification with:

The standard *header file* `<stdarg.h>` shall not be #included, and none of the features that are specified as being provided by `<stdarg.h>` shall be used.

AMD3.67 : For Rule 21.4, replace the Amplification with:

The standard *header file* `<setjmp.h>` shall not be #included, and none of the features that are specified as being provided by `<setjmp.h>` shall be used.

AMD3.68 : For Rule 21.5, replace the Amplification with:

The standard *header file* `<signal.h>` shall not be #included, and none of the features that are specified as being provided by `<signal.h>` shall be used.

AMD3.69 : For Rule 21.10, replace the second line of the Amplification with:

The standard *header file* `<time.h>` shall not be #included, and none of the features that are specified as being provided by `<time.h>` shall be used.

AMD3.70 : For Rule 21.10, in the third line of the Amplification replace:

In C99,

with:

For C99 and later,

## 3.2 Appendix A — Summary of guidelines

### 3.2.1 Existing guidelines

AMD3.71 : Modify existing entries, as follows:

- Rule 21.11: Recategorize as *Advisory* and update headline
- Rule 21.12: Recategorize as *Required* and update headline

### 3.2.2 New guidelines

AMD3.72 : Insert new entries, in the appropriate places, as follows:

| Guideline | Category | Headline |
|---|---|---|
| Dir 4.15 | Required | Evaluation of floating-point expressions shall not lead to the undetected generation of infinities and NaNs |
| Rule 1.5 | Required | Obsolescent language features shall not be used |
| Rule 6.3 | Required | A bit field shall not be declared as a member of a union |
| Rule 7.5 | Mandatory | The argument of an integer-constant macro shall have an appropriate form |
| Rule 8.15 | Required | All declarations of an object with an explicit alignment specification shall specify the same alignment |
| Rule 8.16 | Advisory | The alignment specification of zero should not appear in an object declaration |
| Rule 8.17 | Advisory | At most one explicit alignment specifier should appear in an object declaration |
| Rule 17.9 | Mandatory | A function declared with a `_Noreturn` function specifier shall not return to its caller |
| Rule 17.10 | Required | A function declared with a `_Noreturn` function specifier shall have void return type |
| Rule 17.11 | Advisory | A function that never returns should be declared with a `_Noreturn` function specifier |
| Rule 17.12 | Advisory | A function identifier should only be used with either a preceding `&`, or with a parenthesized parameter list |
| Rule 17.13 | Required | A function type shall not be type qualified |
| Rule 18.9 | Required | An object with temporary lifetime shall not undergo array-to-pointer conversion |
| Rule 21.22 | Mandatory | All arguments to any type-generic macros declared in <tgmath.h> shall have appropriate type |
| Rule 21.23 | Required | All arguments to any multi-argument type-generic macros declared in <tgmath.h> should have the same standard type |
| Rule 21.24 | Required | The random number generator functions of <stdlib.h> shall not be used |
| Rule 23.1 | Advisory | A generic selection should only be expanded from a macro |
| Rule 23.2 | Required | A generic selection that is not expanded from a macro shall not contain potential side effects in the controlling expression |
| Rule 23.3 | Advisory | A generic selection should contain at least one non-default association |
| Rule 23.4 | Required | A generic association shall list an appropriate type |
| Rule 23.5 | Advisory | A generic selection should not depend on implicit pointer type conversion |

| Guideline | Category | Headline |
|---|---|---|
| Rule 23.6 | Required | The controlling expression of a generic selection shall have an *essential type* that matches its standard type |
| Rule 23.7 | Advisory | A generic selection that is expanded from a macro should evaluate its argument only once |
| Rule 23.8 | Required | A default association shall appear as either the first or the last association of a generic selection |

## 3.3 Appendix B — Guidelines attributes

### 3.3.1 Existing guidelines

AMD3.73 : Modify existing entries, as follows:

- Rule 21.11: Recategorize as *Advisory*
- Rule 21.12: Recategorize as *Required*

### 3.3.2 New guidelines

AMD3.74 : Insert new entries, in the appropriate places, as follows:

| Guideline | Category | Applies to | Analysis |
|---|---|---|---|
| Dir 4.15 | Required | C90, C99, C11 | |
| Rule 1.5 | Required | C99, C11 | Undecidable, System |
| Rule 6.3 | Required | C90, C99, C11 | Decidable, Single Translation Unit |
| Rule 7.5 | Mandatory | C99, C11 | Decidable, Single Translation Unit |
| Rule 8.15 | Required | C11 | Decidable, System |
| Rule 8.16 | Advisory | C11 | Decidable, Single Translation Unit |
| Rule 8.17 | Advisory | C11 | Decidable, Single Translation Unit |
| Rule 17.9 | Mandatory | C11 | Undecidable, System |
| Rule 17.10 | Required | C11 | Decidable, Single Translation Unit |
| Rule 17.11 | Advisory | C11 | Undecidable, System |
| Rule 17.12 | Advisory | C90, C99, C11 | Decidable, Single Translation Unit |
| Rule 17.13 | Required | C90, C99, C11 | Decidable, Single Translation Unit |
| Rule 18.9 | Required | C90, C99, C11 | Decidable, Single Translation Unit |
| Rule 21.22 | Mandatory | C99, C11 | Decidable, Single Translation Unit |
| Rule 21.23 | Required | C99, C11 | Decidable, Single Translation Unit |
| Rule 21.24 | Required | C90, C99, C11 | Decidable, Single Translation Unit |
| Rule 23.1 | Advisory | C11 | Decidable, Single Translation Unit |
| Rule 23.2 | Required | C11 | Decidable, Single Translation Unit |
| Rule 23.3 | Advisory | C11 | Decidable, Single Translation Unit |
| Rule 23.4 | Required | C11 | Decidable, Single Translation Unit |
| Rule 23.5 | Advisory | C11 | Decidable, Single Translation Unit |
| Rule 23.6 | Required | C11 | Decidable, Single Translation Unit |
| Rule 23.7 | Advisory | C11 | Decidable, Single Translation Unit |
| Rule 23.8 | Required | C11 | Decidable, Single Translation Unit |

## 3.4 Appendix H — Undefined and critical unspecified behaviour

### 3.4.1 Appendix H.1 — Undefined behaviour

AMD3.75 : Replace the following rows in the table:

| Id | | | Decidable | Guidelines | Notes |
|---|---|---|---|---|---|
| C90 | C99 | C11 | | | |
| | 8 | 9 | Yes | Rule 18.9 | |
| | 35 | | Yes | Rule 18.9 | |
| * | 63 | 66 | Yes | Rule 17.13 | |
| | | 71 | No | Rule 21.10 | |
| | | 73 | Yes | Rule 8.15 | |
| | 112 | 118 | No | Dir 4.11, Rule 21.12 | |
| | 137 | 145 | Yes | Rule 7.5 | |
| | 184 | 195 | Yes | Rule 21.11,<br>Rule 21.22 | |

### 3.4.2 Appendix H.2 — Critical unspecified behaviour

AMD3.76 : Replace the following rows in the table:

| Id | | | Decidable | Guidelines | Notes |
|---|---|---|---|---|---|
| C90 | C99 | C11 | | | |
| | 30 | 30 | Yes | Dir 4.11, Dir 4.15 | |
| | 46 | 52 | Yes | Dir 4.15 | |
| | 47 | 53 | Yes | Dir 4.15 | |
| | TC3 | 54 | Yes | Dir 4.11, Dir 4.15 | Added to C99 by TC3 |
| | TC3 | 55 | Yes | Dir 4.11, Dir 4.15 | Added to C99 by TC3 |
| | 48 | 56 | Yes | Dir 4.11, Dir 4.15 | |
| | 49 | 57 | Yes | Dir 4.11, Dir 4.15 | |

## 3.5 Addendum 1 — Rule mappings

Update MISRA C:2012 Addendum 1 [7] as follows:

| MISRA C:2004 | MISRA C:2012 | Significant changes from MISRA C:2004 for C90 code |
|---|---|---|
| Rule 13.3 (required) | Dir 1.1 (required)<br>Dir 4.11 (required)<br>Dir 4.15 (required)<br>Rule 10.1 (required) | Caution must be exercised when using floating-point arithmetic |
| Rule 16.9 (required) | Rule 17.12 (Advisory) | Reinstated by Amendment 3. The issues raised by MISRA C:2004 Rule 16.9 are also partly covered by the set of type checking rules, MISRA C:2012 Rules 10.1–10.4. |

## 3.6 Addendum 3 — Coverage against CERT C

Update MISRA C:2012 Addendum 3 [9] to reflect the changes in this Amendment

### 3.6.1 Guideline by guideline

AMD3.77 : Replace the appropriate rows as follows:

| CERT C Rule | MISRA C:2012 Guidelines | | | Comments |
|---|---|---|---|---|
| | Guidelines | Coverage | | |
| EXP35-C | R.18.9 | Explicit | Strong | |
| EXP36-C | R.11.3, R11.4, R11.5, R11.6 | Explicit | Strong | |
| EXP46-C | R.10.1 | Explicit | Strong | |
| MSC30-C | R.21.24 | Restrictive | Strong | `rand()` shall not be used |
| MSC32-C | R.21.24 | Restrictive | Strong | `srand()` shall not be used |

### 3.6.2 Coverage summary

AMD3.78 : Replace the summary table as follows:

| Classification | Strength | Number |
|---|---|---|
| Explicit | Strong | 41 |
| | Weak | 5 |
| Implicit | Strong | 1 |
| | Weak | 13 |
| Restrictive | Strong | 24 |
| | Weak | 0 |
| Partial | Strong/Weak/None | 0 |
| Out of Scope | None | 13 |
| None | None | 2 |
| | Total | 99 |

# 4 References

The following documents are referenced from within this amendment:

## 4.1 MISRA C

[1] MISRA C:2012 *Guidelines for the use of the C language in critical systems* (3rd Edition)
ISBN 978-1-906400-10-1 (paperback), ISBN 978-1-906400-11-8 (PDF),
MIRA Limited, Nuneaton, March 2013

[2] MISRA C:2012 *Guidelines for the use of the C language in critical systems* (3rd Edition, 1st Revision),
ISBN 978-1-906400-21-7 (paperback), ISBN 978-1-906400-22-4 (PDF),
HORIBA MIRA Limited, Nuneaton, February 2019

[3] MISRA C:2012 Technical Corrigendum 1, *Technical clarification of MISRA C:2012*,
ISBN 978-1-906400-17-0 (PDF),
HORIBA MIRA Limited, Nuneaton, June 2017

[4] MISRA C:2012 Technical Corrigendum 2, *Technical clarification of MISRA C:2012*,
ISBN 978-1-911700-00-5 (PDF),
The MISRA Consortium Limited, Norwich, February 2022

[5] MISRA C:2012 Amendment 1, *Additional security guidelines for MISRA C:2012*,
ISBN 978-1-906400-16-3 (PDF),
HORIBA MIRA Limited, Nuneaton, April 2016

[6] MISRA C:2012 Amendment 2, *Updates for ISO/IEC 9899:2011 Core Functionality*,
ISBN 978-1-906400-25-5 (PDF),
HORIBA MIRA Limited, Nuneaton, February 2020

[7] MISRA C:2012 Addendum 1, *Rule mappings*,
ISBN 978-1-906400-12-5 (PDF),
MIRA Limited, Nuneaton, March 2013

[8] MISRA C:2012 Addendum 2 (2nd Edition), *Coverage of MISRA C:2012 against ISO/IEC TS 17961:2013 "C Secure"*,
ISBN 978-1-906400-18-7 (PDF),
HORIBA MIRA Limited, Nuneaton, January 2018

[9] MISRA C:2012 Addendum 3, *Coverage of MISRA C:2012 against against CERT C 2016 Edition*,
ISBN 978-1-906400-19-4 (PDF),
HORIBA MIRA Limited, Nuneaton, January 2018

## 4.2 The C Standard

[10] ISO/IEC 9899:1999, *Programming languages — C*,
International Organization for Standardization, 1999

[11] ISO/IEC 9899:2011, *Programming languages — C*,
International Organization for Standardization, 2011

[12] ISO/IEC 9899:2018, *Programming languages — C*,
International Organization for Standardization, 2018

## 4.3 Other Standards

[13] IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems*, International Organization for Standardization, 1989

Note: Subsequent to the publication of ISO/IEC 9899:2018 [12], *IEC 60559:2020* has been published.