Coding Standards

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

**Mário
de Sousa**

msousa@fe.up.pt

Coding Standards for Embedded Systems

- **MISRA-C**
  - Guidelines for the use of the C language in critical systems

> The MISRA-C Guidelines define a subset of the C language in which the opportunity to make mistakes is either removed or reduced.
>
> [MISRA-C standard]

- **CERT-C**
  - Produced by Software Engineering Institute (SEI) at Carnegie Mellon University (CMU)
    (sponsored by U.S. Defence Department)

- **ISO/IEC TS 17961 – C Secure Coding Rulea**
  - Produced by ISO/IEC JTC 1/SC 22/WG 14
    (same people responsible for C standard)

JTC – Joint technical committee
WG – Working Group

MISRA-C

- Several Versions:
  - 1998, 2004, 2012 (pub. 2013), 2016 (amendment)
  - MISRA C:1998
    - first edition, derived from PRQA standards developed for Ford and Rover (UK, for automotive applications)
    - 127 rules: 93 required + 34 advisory
  - MISRA C:2004
    - Amended, extended: complete renumbering of the rules.
    - Includes examples
    - 142 rules: 122 required + 20 advisory

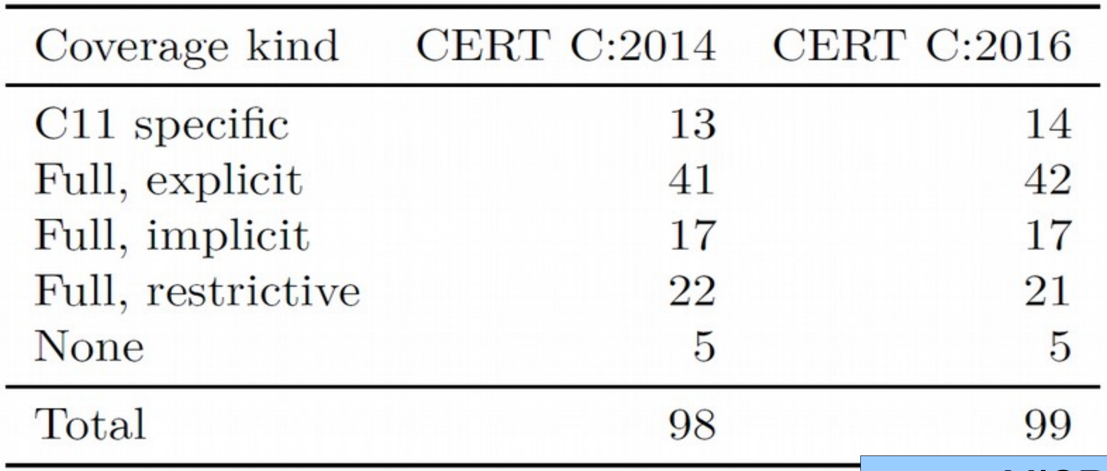
MISRA-C

- Several Versions:
  - 1998, 2004, 2012 (pub. 2013), 2016 (amendment)
  - MISRA C:2012 (from 2013)
    - extends support to the C99 (reduce the use of dangerous C99 features)
    - some restrictions were removed
    - new 'Mandatory' rule class (non-negotiable rules where deviations are never allowed)
    - enhancements to ensure rules may be checked automatically (where possible)
    - 143 rules
  - Several Amendments
    - Amendment 1 (2016) : 14 new guidelines (13 Rules, 1 Directive)
    - Amendment 2 (2020)
    - Technical Corrigendum 1 (2017) : fix typographical errors and add clarifications
    - Free to download

      https://www.misra.org.uk/Publications/tabid/57/Default.aspx#label-c3-add2

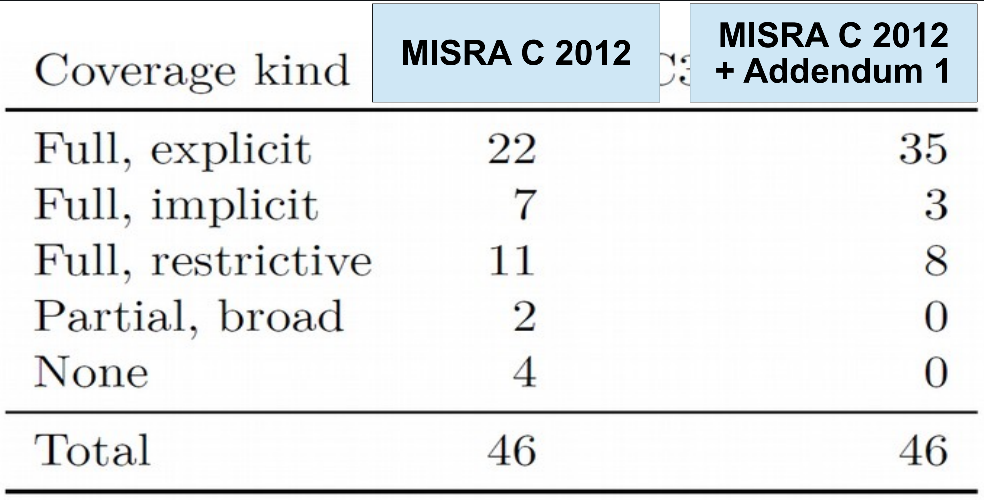
MISRA-C vs [CERT C & ISO/IEC TS 17961]

[MISRA C 2012 + Addendum 1] vs CERT C



Coverage kind	CERT C:2014	CERT C:2016
C11 specific	13	14
Full, explicit	41	42
Full, implicit	17	17
Full, restrictive	22	21
None	5	5
Total	98	99

MISRA C 2012 vs ISO/IEC TS 17961



Coverage kind	MISRA C 2012	MISRA C 2012 + Addendum 1
Full, explicit	22	35
Full, implicit	7	3
Full, restrictive	11	8
Partial, broad	2	0
None	4	0
Total	46	46

"MISRA C, for Security's Sake!", Roberto Bagnara
member‡ of MISRA C Working Group,
member‡ of ISO/IEC JTC1/SC22/WG14 - C Standardization Working Group

MISRA-C: Guideline Classification

- Guideline Types:
  - Directives
    - May be loosely defined (allowing alternative interpretations)
    - May address “process” or “documentation” requirements

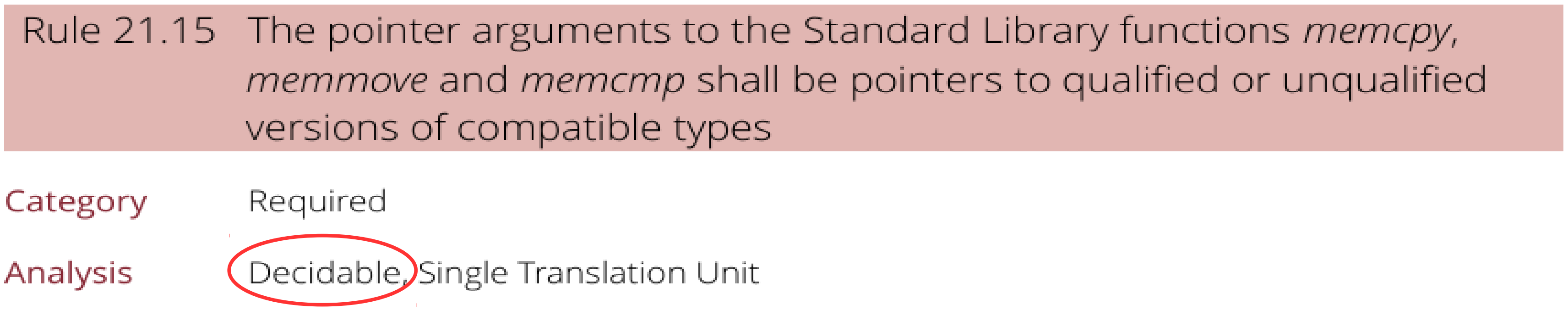
Dir 4.14	The validity of values received from external sources shall be checked

  - Rules
    - Have well defined requirements
    - Are statically enforceable (subject to certain limitations)

Rule 21.14	The Standard Library function *memcmp* shall not be used to compare null terminated strings

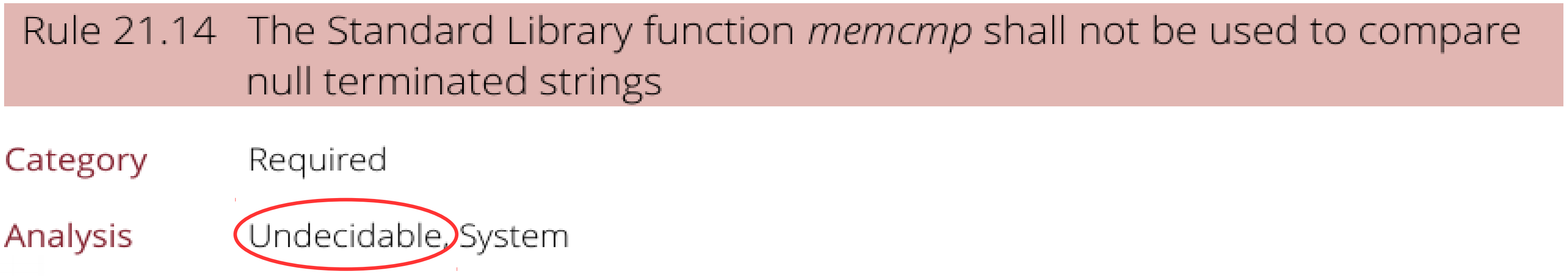
MISRA-C: Guideline Classification

- Rule Analysis
  - Decidable
    - If a tool can, in all situations, automatically determine whether the rule is being followed



Rule 21.15	The pointer arguments to the Standard Library functions *memcpy*, *memmove* and *memcmp* shall be pointers to qualified or unqualified versions of compatible types
Category	Required
Analysis	Decidable, Single Translation Unit

  - Undecidable
    - If it is not decidable



Rule 21.14	The Standard Library function *memcmp* shall not be used to compare null terminated strings
Category	Required
Analysis	Undecidable, System

MISRA-C: Guideline Classification

- Guideline Categories:
  - Mandatory (from 2012 onwards)
    - mandatory requirements
    - MISRA-C compliant code must abide by all these rules
    - **Deviation** (i.e. exceptions) **NEVER** allowed
  - Required
    - mandatory requirements
    - MISRA-C compliant code must abide by all these rules
    - **Deviation** (i.e. exceptions)are allowed:
      must follow formal deviation procedure for each rule exception
  - Advisory
    - Should normally be followed as far is reasonably practicable
    - **Violations** are identified, but deviation procedure not required

Deviation, Violations: Defined in MISRA-C Compliance procedures

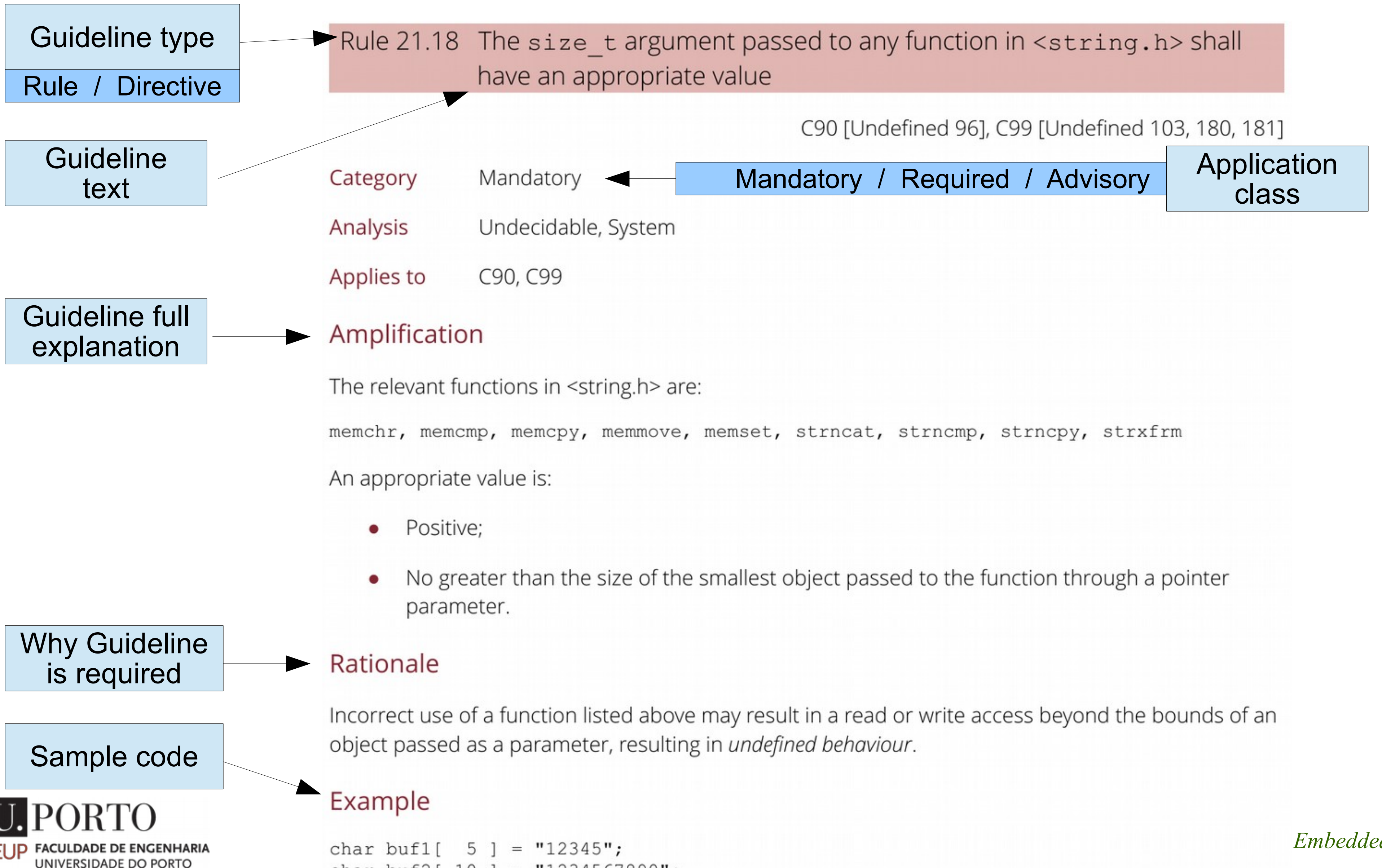
MISRA-C: Guideline Classification

- Guideline Categories:

May be Reclassified
- usually signed off by client before project starts
- documented in a “Guideline Recategorization Plan”

  - Mandatory
    - Cannot be reclassified
  - Required
    - May be reclassified to Mandatory
  - Advisory
    - May be reclassified to Mandatory, Required, or Dis-applied
  - Dis-applied
    - Violations may be ignored (do not check)

MISRA-C: Guideline Presentation


Guideline type
Rule / Directive
Guideline text
Mandatory / Required / Advisory
Application class
Guideline full explanation
Why Guideline is required
Sample code

Rule 21.18 The `size_t` argument passed to any function in `<string.h>` shall have an appropriate value

C90 [Undefined 96], C99 [Undefined 103, 180, 181]

Category	Mandatory
Analysis	Undecidable, System
Applies to	C90, C99

Amplification

The relevant functions in <string.h> are:

`memchr, memcmp, memcpy, memmove, memset, strncat, strncmp, strncpy, strxfrm`

An appropriate value is:

- Positive;
- No greater than the size of the smallest object passed to the function through a pointer parameter.

Rationale

Incorrect use of a function listed above may result in a read or write access beyond the bounds of an object passed as a parameter, resulting in *undefined behaviour*.

Example

```
char buf1[  5 ] = "12345";
```

MISRA-C: Guideline Classification

- Organization
  - Organized under topics:
    - » 1 Environment,
    - » 2 Language extensions,
    - » 3 Documentation,
    - » 4 Character sets,
    - » 5 Identifiers,
    - » 6 Types,
    - » 7 Constants,
    - » 8 Declarations and definitions,
    - » 9 Initialisation,
    - » 10 Arithmetic type conversions,
    - » 11 Pointer type conversions,
    - » 12 Expressions,
    - » 13 Control statement expressions,
    - » 14 Control flow,
    - » 15 Switch statements,
    - » 16 Functions,
    - » 17 Pointers and arrays,
    - » 18 Structures and unions,
    - » 19 Preprocessing directives,
    - » 20 Standard libraries,
    - » 21 Run-time failures

MISRA-C: 1 - Environment

- Rule 1.1 (required)
  - All code shall conform to ISO/IEC 9899:1990 "Programming languages — C"
    - It is recognised that deviations will be necessary to permit certain language extensions, for example to support hardware specific features.
- Rule 1.2 (required)
  - No reliance shall be placed on undefined or unspecified behaviour.
- Rule 1.3 (required)
  - Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the languages/compilers/assemblers conform.
    - Examples of issues that need to be understood are:
      - stack usage,
      - parameter passing and the way in which data values are stored (lengths, alignments, aliasing, overlays, etc.)

MISRA-C: 1 - Environment

- Rule 1.4 (required)
  - The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.
    - If the compiler/linker is not capable of meeting this limit, then use the limit of the compiler.
- Rule 1.5 (advisory)
  - Floating-point implementations should comply with a defined floating-point standard.

MISRA-C: 2 – Language Extensions

- Rule 2.1 (required)
  - Assembly language shall be encapsulated and isolated.
    - assembly language instructions are recommended to be encapsulated and isolated in either (a) assembler functions, (b) C functions or (c) macros.
    - e.g.: `#define NOP asm("NOP")`
- Rule 2.2, 2.3 (required)
  - Source code shall only use /* ... */ style comments.
  - The character sequence /* shall not be used within a comment.
- Rule 2.4 (advisory)
  - Sections of code should not be “commented out”.
    - use instead conditional compilation (e.g. `#if` or `#ifdef` constructs with a comment).

MISRA-C: 3 – Documentation

- Rule 3.1 (required)
  - All usage of implementation-defined behaviour shall be documented.
    - Assumes a deviation of rule 1.2 has been approved.
    - NOTE: some rules are redundant, and only apply if another rule is not being followed
- Rule 3.2, 3.4 (required)
  - The character set and its encoding shall be documented.
  - All uses of the #pragma directive shall be documented and explained.
- Rule 3.3 (advisory)
  - The implementation of integer division in the chosen compiler should be determined, documented and taken into account.
    - (e.g. -5/3 = -1 remainder -2)   or   (e.g. -5/3 = -2 remainder +1).
- Rule 3.6 (required)
  - All libraries used in production code shall comply with MISRA-C

MISRA-C: 3 – Documentation

- Rule 3.5 (required)
  - If it is being relied upon, the implementation defined behaviour and packing of bitfields shall be documented.
    - Only acceptable for packing together of short-length data to economise on storage
    - If the elements of the structure are only ever accessed by their name, the programmer needs to make no assumptions about the way that the bit fields are stored within the structure.
    - structures recommended be declared specifically to hold the sets of bit fields, and do not include any other data within the same structure.

For example the following is acceptable:

```
struct message
 /* Struct is for bit-fields only */
{
    Signed int little: 4;
            /* Note: use of basic types is required */
    unsigned int x_set: 1;
    unsigned int y_set: 1;
} message_chunk;
```

MISRA-C: 5 – Identifiers

- Rule 5.1, 5.2 (required)
  - Identifiers (internal and external) shall not rely on the significance of more than 31 characters.
  - Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.

For example the following is **NOT** acceptable:

```
int16_t i;
{
    int16_t i; /* This is a different variable */
               /* This is not compliant */
    i = 3;     /* It could be confusing as to which
                  i this refers */
}
```

MISRA-C: 5 – Identifiers

- Rule 5.3 (required)
  - A typedef name shall be a unique identifier.

For example the following is **NOT** acceptable:

```
{
    typedef unsigned char uint8_t;
}
{
    typedef unsigned char uint8_t; /* NOT compliant - redefinition */
}
{
    unsigned char uint8_t;     /* NOT compliant - reuse of uint8_t */
}
```

MISRA-C: 5 – Identifiers

- Rule 5.3 (required)
  - A tag name name shall be a unique identifier.

```
struct stag { uint16_t a; uint16_t b; };

struct stag a1 = { 0, 0 };   /* Compliant - compatible with above */
union  stag a2 = { 0, 0 };   /* NOT compliant - not compatible with
                                previous declarations */
void foo(void) {
struct stag { uint16_t a; }; /* NOT compliant - tag stag redefined */
}
```

MISRA-C: 5 – Identifiers

- Rule 5.5, 5.6 (advisory)
  - No object or function identifier with static storage duration should be reused.
  - No identifier in one name space should have the same spelling as an identifier in another name space,
    with the exception of structure member and union member names.

```
struct { int16_t key; int16_t value; } record;

int16_t value; /* NOT compliant - second use of value */


struct device_q { struct device_q *next;  /* ... */ }
struct   task_q { struct   task_q *next;  /* ... */ }
              /* Allowed - second use of next inside struct */
```

MISRA-C: 6 - Types

- Rule 6.3 (advisory)
  - typedefs that indicate size and signedness should be used in place of the basic numerical types.

Recommend the use of ISO (POSIX) typedefs

```
typedef              char   char_t;
typedef signed      char   int8_t;
typedef signed     short   int16_t;
typedef signed       int   int32_t;
typedef signed      long   int64_t;
typedef unsigned    char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned     int uint32_t;
typedef unsigned    long uint64_t;
typedef            float float32_t;
typedef           double float64_t;
typedef      long double float128_t;
```

MISRA-C: 7 – Constants

- Rule 7.1, (required)
  - Octal constants (other than zero) and octal escape sequences shall not be used.

```
code[1] = 109; /* equivalent to decimal 109 */
code[2] = 100; /* equivalent to decimal 100 */
code[3] = 052; /* equivalent to decimal 42 */
code[4] = 071; /* equivalent to decimal 57 */
```

MISRA-C: 8 – Declarations & Definitions

- Rule 8.1, 8.3, 8.5, 8.2 (required)
  - Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.
  - For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.
  - There shall be no definitions of objects or functions in a header file.
    - i.e.: header files cannot contain *executable* code.
  - Whenever an object or function is declared or defined, its type shall be explicitly stated.

```
extern x;                    /* NOT compliant - implicit int type */
extern int16_t x;            /* Compliant     - explicit type */
static foo(void);            /* NOT compliant - implicit type */
static int16_t foo(void);    /* Compliant     - explicit type */
```

MISRA-C: 8 – Declarations & Definitions

- Rule 8.6, 8.7, 8.8, 8.10 (required)
  - Functions shall be declared at file scope.
    - Declaring functions inside a { } block can be confusing!
  - Objects shall be defined at block scope if they are only accessed from within a single function.
    - Variables only used inside a function must not be declared as global variables.
  - An external object or function shall be declared in one and only one file.
    - i.e. Have a single **`extern int16_t var_x;`** in a header file, and include that file when needing to access the variable.
  - All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.
    - i.e. a variable or function only used inside a single file, should be declared as **`static`**.

MISA-C: 9 – Initialization

- Rule 9.1 (required)
  - All automatic variables shall have been assigned a value before being used.
    - All variables shall have been written to before they are read.
    - Does not necessarily require initialisation at declaration.

MISRA-C: 10 – Arithmetic Type Conversions

- Rule 10.1, 10.2 (required)
  - The value of an expression of integer type shall not be implicitly converted to a different underlying type if: <...snip...>
  - The value of an expression of floating type shall not be implicitly converted to a different type if: <...snip...>
    - » These two rules broadly encapsulate the following principles:
      - No implicit conversions between signed and unsigned types
      - No implicit conversions between integer and floating types
      - No implicit conversions from wider to narrower types
      - No implicit conversions of function arguments
      - No implicit conversions of function return expressions
      - No implicit conversions of complex expressions
    - » restricting implicit conversion of complex expressions => in a sequence of arithmetic operations within an expression, all operations are conducted in exactly the same type.

```
u32a + u16b + u16c  → OK. Both '+' operations performed as u32_t
u16a + u16b + u32c  → NOT OK. 1st operation u16_t,
                              2nd operation u32_t
```

MISRA-C: 12 – Expressions

- Rule 12.1 (advisory)
  - Limited dependence should be placed on C’s operator precedence rules in expressions.
    - Use parentheses to override default operator precedence, and also to emphasise it.
- Rule 12.2 (required)
  - The value of an expression shall be the same under any order of evaluation that the standard permits.

```
x = b[i] + i++;           → NOT OK. i++ before or after b[i]?
foo( i, i++);             → NOT OK.
p->task_start_fn (p++);   → NOT OK.
x = f(a) + g(a);          → NOT OK. f(a) or g(a) may have side effects!
x = y = y = z / 3 ;       → NOT OK.
x = y = y++;              → NOT OK.

/* also: beware accessing volatile variables! */
```

MISRA-C: 12 – Expressions

- Rule 12.10 (required)
  - The comma operator shall not be used.
    - » `for (i=0, j=10; i < j; i++, j++);`
- Rule 12.12 (required)
  - The underlying bit representations of floating-point values shall not be used.

MISRA-C: 13 – Control Statements

- Rule 13.3, 13.4 (required)
  - Floating-point expressions shall not be tested for equality or inequality.
  - The controlling expression of a for statement shall not contain any objects of floating type.
- Rule 13.5, 13.6 (required)
  - The three expressions of a for statement shall be concerned only with loop control.
  - Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop.
- Rule 13.7 (required)
  - Boolean operations whose results are invariant shall not be permitted.
    - » Most likely a programming error

MISRA-C: 14 – Control Flow

- Rule 14.1, 14.2 (required)
  - There shall be no unreachable code.
  - All non-null statements shall either:
    (a) have at least one side-effect however executed, or
    (b) cause control flow to change.
- Rule 14.4, 14.5, 14.6 (required)
  - The goto statement shall not be used.
  - The continue statement shall not be used.
  - For any iteration statement there shall be at most one break statement used for loop termination.
  - A function shall have a single point of exit at the end of the function.

MISRA-C: 16 – Functions

- Rule 16.1, 16.3 - 16.7 (required)
  - Functions shall not be defined with a variable number of arguments.
  - Identifiers shall be given for all of the parameters in a function prototype declaration.
  - The identifiers used in the declaration and definition of a function shall be identical.
  - Functions with no parameters shall be declared and defined with the parameter list void.
  - The number of arguments passed to a function shall match the number of parameters.
  - A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object. (advisory)

MISRA-C: 16 – Functions

- Rule 16.2 (required)
  - Functions shall not call themselves, either directly or indirectly.
    - Recursion is **NOT** allowed!
- Rule 16.8, 16.10 (required)
  - All exit paths from a function with non-void return type shall have an explicit return statement with an expression.
  - If a function returns error information, then that error information shall be tested.

MISRA-C: 17 – Pointers & Arrays

- Rule 17.1 – 17.3 (required)
  - Pointer arithmetic shall only be applied to pointers that address an array or array element.
  - Pointer subtraction shall only be applied to pointers that address elements of the same array.
  - >, >=, <, <= shall not be applied to pointer types except where they point to the same array.
- Rule 17.5 (advisory)
  - The declaration of objects should contain no more than 2 levels of pointer indirection.

MISRA-C: 17 – Pointers & Arrays

- Rule 17.4 (required)
  - Array indexing shall be the only allowed form of pointer arithmetic.

```
void my_fn(uint8_t * p1, uint8_t p2[]) {
  uint8_t index = 0U;
  uint8_t * p3;
  uint8_t * p4;
  *p1 = 0U;
  p1 ++;        /* NOT compliant - pointer increment */
  p1 = p1 + 5; /* NOT compliant - pointer increment */
  p1[5] = 0U;  /* NOT compliant - p1 was not declared as an array */
  p3 = &p1[5]; /* NOT compliant - p1 was not declared as an array */
  p2[0] = 0U;
  index ++;
  index = index + 5U;
  p2[index] = 0U; /* compliant */
  p4 = &p2[5];    /* compliant */
}
```

MISRA-C: 18 – Structures & Unions

- Rule 18.2, 18.3 (required)
  - An object shall not be assigned to an overlapping object.
  - An area of memory shall not be reused for unrelated purposes.
  - Unions shall not be used.
    - » Usually less efficient
    - » When high execution speed or low program memory usage is more important than portability, the implementation using unions might be preferred.

```
typedef union {
  uint32_t word;
  uint8_t bytes[4];
} word_msg_t;

uint32_t read_word_big_endian (void)
{
  word_msg_t tmp;
  tmp.bytes[0] = read_byte();
  tmp.bytes[1] = read_byte();
  tmp.bytes[2] = read_byte();
  tmp.bytes[3] = read_byte();
  return (tmp.word);
}
```

```
uint32_t read_word_big_endian (void)
{
  uint32_t word;
  word = ((uint32_t)read_byte()) << 24;
  word = word | (((uint32_t)read_byte()) << 16);
  word = word | (((uint32_t)read_byte()) << 8);
  word = word | ((uint32_t)read_byte());
  return (word);
}
```

MISRA-C: 20 – Standard Libraries

- Rule 20.4 (required)
  - Dynamic heap memory allocation shall not be used.
    - » precludes the use of calloc, malloc, realloc and free.
- Rule 20.7, 20.8, 20.10 (required)
  - The setjmp macro and the longjmp function shall not be used.
  - The signal handling facilities of <signal.h> shall not be used.
  - The input/output library <stdio.h> shall not be used in production code.
    - » getpos, fopen, ftell, gets, perror, remove, rename and ungetc.
  - The library functions atof, atoi and atol from library <stdlib.h> shall not be used.
    - » undefined behaviour when the string cannot be converted.
  - The time handling functions of library <time.h> shall not be used
    - » time, strftime.
    - » Various aspects are implementation dependent or unspecified, such as the formats of times.