RULES FOR SECURE C LANGUAGE SOFTWARE DEVELOPMENT

ANSSI GUIDELINES

ANSSI-PA-073
24/03/2022

TARGETED AUDIENCE:

Developers Administrators IT security managers IT managers Users

Information

Warning

This document, written by ANSSI, the French National Information Security Agency, presents the **"Rules for secure C language software development"**. It is freely available at `www.ssi.gouv.fr/en/`.

It is an original creation from ANSSI and it is placed under the "Open Licence v2.0" published by the Etalab mission [ETALAB].

According to the Open Licence v2.0, this guide can be freely reused, subject to mentionning its paternity (source and date of last update). Reuse means the right to communicate, distribute, redistribute, publish, transmit, reproduce, copy, adapt, modify, extract, transform and use, including for commercial purposes

These recommendations are provided as is and are related to threats known at the publication time. Considering the information systems diversity, ANSSI cannot guarantee direct application of these recommendations on targeted information systems. Applying the following recommendations shall be, at first, validated by IT administrators and/or IT security managers.

This document is a courtesy translation of the initial French document **"Règles de programmation pour le développement sécurisé de logiciels en langage C"**, available at `www.ssi.gouv.fr`. In case of conflicts between these two documents, the latter is considered as the only reference.

Document changelog:

VERSION	DATE	CHANGELOG
1.4	24/03/2022	First English version

Contents

1
Introduction

The C language offers great freedom to developers. However, it contains ambiguous or risky constructions that may introduce errors during development. The C language standard does not specify all the desired behaviours, and therefore some remain undefined or unspecified[1]. *Developers of compilers, libraries or operating systems are therefore free to make their own choices.*

Restrictions must therefore be set on the use of C language in order to identify the various risky or non-portable constructions and to limit or even prohibit their use. The restrictions defined in this guide are intended to encourage the production of more secure, safer, more robust software, and also to encourage their portability from one system to another, whether PC type or embedded.

This guide defines a set of rules, recommendations and good practices dedicated to secure C language development. *In this document, we currently restrict ourselves to the 2 standards C90*[2] *and C99, which are the most widely used.*

When a rule is directly associated with one specific standard, this is clearly indicated to avoid any confusion. If not specified, both standards are concerned.

Rule

A *rule* must *always* be respected; no exceptions are tolerated.

Recommendation

A *recommendation* should be respected except in certain exceptional cases, which implies a clear and precise justification from the developer. Recommendations are abbreviated to “RECO”.

This guide also contains good practices. These are often somewhat more subjective points like coding conventions, such as the indentation of the code, for example.

[1] These concepts are defined on page9.

[2] N.B. the C90 standard is also referred to as C89 by the C community.

Good practice

The *good practices* defined in this guide are highly recommended, but they can be replaced by those already in place in the developer's organisation or development team if equivalent rules exist.

This guide has various objectives:

- increasing the security, quality and reliability of the source code produced, by identifying bad or dangerous programming practices;
- facilitating the analysis of the source code during peer review or using static analysis tools;
- establishing a level of confidence in the security, reliability and robustness of a development;
- making software maintenance easier, as well as the addition of features.

The idea of this guide is not to reinvent the wheel, but rather to use existing documents (methodological guides, language standard references, *etc.*) to extract, modify and specify a set of recommendations for the secure development of the C language. The reference documents used are as follows:

- MISRA-C: 2012 Guidelines for the use of the C language in critical systems [Misra2012],
- C ANSI 90 [AnsiC90],
- C ANSI 99 [AnsiC99],
- GCC: Reference Documentation [GccRef],
- CLANG'S Documentation [ClangRef],
- SEI CERT C Coding Standard [Cert],
- ISO 17961 C Secure Coding Rules [IsoSecu],
- CWE MITRE Common Weakness Enumeration [Cwe].

This guide is not aimed at any particular application field and is not intended to replace the development constraints imposed by any normative context (automotive, aeronautics, critical systems, *etc.*). Its aim is to address precisely those secure C developments not covered by these normative constraints.

2
Coding convention

Above all, any development project of any kind must follow a clear, precise and documented development convention. This development convention must be known to all developers and applied systematically.

Each developer has his or hers own programming habits, code layout and variable naming. However, when producing software, these different programming habits between developers result in a heterogeneous set of source files, which are more difficult to audit and maintain.

RULE
1

RULE — Application of clear and explicit coding conventions

Coding conventions must be defined and documented at the software project level. These conventions must define at least the following points: encoding of source files, code layout and indentation, standard types to be used, naming (libraries, files, functions, types, variables, *etc.*), documentation format.
These conventions must be followed by each developer.

This rule regarding development conventions is certainly obvious and the aim here is not to impose, for example, a choice of variable naming (such as snake-case versus camel-case), but to ensure that a choice has indeed been made at the beginning of the development project and that it is explicit.

Appendix E provides examples of coding conventions that can be used or adapted as required.

[Misra2012] Dir. 4.5 Identifiers in the same name space with overlapping visibility should be typographically unambiguous.
[Cert] Rec. DCL02-C Use visually distinct identifiers.
[IsoSecu] Using Identifiers that are reserved for the implementation [resident].

3

Undefined and unspecified behaviours

As the remainder of this guide will frequently make use of the concepts of *undefined behaviour* and *unspecified behaviour*, we recall them hereinbelow.

Undefined behaviour

An *undefined behaviour* is a behaviour for which nothing is imposed by the C standard, and which follows an error in the construction of the program, a non-portable construction or an incorrect use of data. An example is a *signed integer overflow*.

Unspecified behaviour

An *unspecified behaviour* is a behaviour for which the C standard provides at least two alternative behaviours that are accepted, but none of which are imposed. An example is the order of evaluation of the operations # and ## during the substitution of a macro.

Information

The exhaustive list of all the unspecified behaviours and all the undefined behaviours is available in appendices G and J of standards C90 [AnsiC90] and C99 [AnsiC99].

This guide only considers a C coding environment compliant with the C90 or C99 standards.

Information

Concurrent programming is not covered in this version of the guide, but will be covered in a later version.

RULE
2

RULE – Only C coding in accordance with the standard is authorised

No violation of C constraints and syntax as defined in the C90 or C99 standards is authorised.

References

[Misra2012] Rule 1.1 The program shall contain no violations of the standard C syntax and *constraints* and shall not exceed the implementation translation limits.
[Misra2012] Rule 1.2 Language extensions should not be used.
[Misra2012] Rule 1.3 There shall be no occurrence of undefined or critical unspecified behaviour.
[Cwe] CWE-710 Improper Adherence to coding standard.

[Cwe] CWE-758 Reliance on undefined, unspecified or implementation-defined behavior.

4

Preprocessor and macros

4.1 Inclusion of the necessary header files

Only the necessary header files need to be included, but some additional rules need to be observed. When a header file itself includes other header files, these declarations will be propagated to all source or header files that include this first file. This C language mechanism results in the cascaded inclusion of header files and declarations.

If the inclusion of header files is not minimised, this generates unnecessary dependencies, increases compilation time, and makes the subsequent analysis of the code more complex (whether manual or tool-based). In order to reduce dependencies and unnecessary propagation of declarations, header file inclusions should be made in a ".c" file and not in a ".h" header file. However, in some cases, such as typical type definition, the inclusion of header files from the standard library (such as `stddef.h` and `stdint.h`) in another header file is justifiable.

RECO 3

RECOMMENDATION — Limit and justify header file inclusions in another header file

Header files should be included as needed during development and not "automatically" by the developer.

RULE 4

RULE — Only the necessary header files should be included

In addition, the header file inclusion mechanism can result in multiple inclusions of the same header file, making proofreading of the code difficult at best. Defining a specific symbol for each header file using the preprocessor directive (`#define`) and checking that this symbol has not already been defined (`#ifndef`) avoids repeated inclusion of a header file. This is referred to as a *multiple include guard macro*. Be sure to define a unique symbol for each file. The name of this symbol can be constructed by taking the file name and substituting the "." with a "_".

RULE 5

RULE — Use multiple include guard macros for a file

A guard macro against multiple inclusions of a file should be used to prevent the content of a header file from being included more than once:

```
// start of header file
```

```
#ifndef HEADER_H
#define HEADER_H
/* file content */
#endif
// end of header file
```

Warning

The use of the `#pragma once` directive is widespread but is not standardized. This solution is therefore not recognised in this guide, although it is supported by most compilers. Its use can be problematic since this directive is specific to each compiler (especially when managing header files that are duplicated in multiple physical sources or mount points).

Finally, for reasons of readability, the location of header file inclusions must comply with certain specific rules.

RULE 6

RULE — Header file inclusions are grouped at the beginning of the file

All header file inclusions must be grouped at the beginning of the file or just after preprocessor comments or directives, but always before the definition of global variables or functions.

RECO 7

RECOMMENDATION — System header file inclusions are made before user header file inclusions

GOOD PRACTICE 8

GOOD PRACTICE — Use alphabetical order in the inclusion of each type of header files

To avoid any redundancy in system or user header file inclusions, the developer can use the alphabetical order, which offers a deterministic inclusion order and facilitates the code review.

Information

Although these last three rules, recommendations and good practices address a problem of readability and maintainability, and not directly a security problem in the strict sense of the term, these first two aspects remain essential for any type of development.

When the inclusion of a header file is omitted, the compiler may provide a warning about the use of an implicitly declared function.

Information

Implicit function declarations are detected with GCC and CLANG using the option `-Wimplicit-function-declaration`.

Bad example

In the code below, the inclusion of `string.h` serves no purpose in `file.h` since the only declaration of `string.h` used by `file.c` is the `memcpy` function. The inclusion of `string.h` must therefore be moved to `file.c` with a multiple include guard in `file.h`.

```
/* header.h */
#include <string.h> /* should only be included in file.c */

void foo(uint8_t* val, uint32_t length);

/* file.c */
#include <string.h>
#include <stdint.h>
#include <stdlib.h>
#include "header.h"

#define BUFFER_LEN 8U

void foo(uint8_t *val, uint32_t length) {
  uint8_t buffer[BUFFER_LEN];
  if (NULL != val) {
    memcpy(buffer, val, min(BUFFER_LEN, length));
    ...
  }
}
```

Good example

The example below includes in the header file only the necessary definitions, and an include guard is present. Be careful however, when using the include guard, not to use an already reserved identifier as a name in the macro, which is a classic error when using the include guard.

```
/* header.h */
#ifndef HEADER_H /* include guard to avoid ultiple inclusion */
#define HEADER_H

void foo(uint8_t *val, uint32_t length);

#endif /*HEADER_H*/

/* file.c */
#include <string.h>
#include <stdint.h>
#include <stdlib.h>
#include "header.h"

#define BUFFER_LEN 8U

void foo(uint8_t *val, uint32_t length) {
  uint8_t buffer[BUFFER_LEN];

  if (NULL != val) {
    memcpy(buffer, val, min(BUFFER_LEN, length));
    ...
  }
}
```

4.1.1 References

[Misra2012] Dir. 4.10. Precautions shall be taken in order to prevent the contents of a header file being included more than once.
[Misra2012] Rule 20.1. `#include` directives should only be preceded by preprocessor directives or comments.
[Cert] Rec. PRE06-C Enclose header files in an include guard.

4.2 Non-inclusion of source files

The inclusion of a source file in another source file can generate problems with link editing (multiple definitions of global variables or identical functions) or duplication of binary code (in the event that the included elements have been declared with the keyword `static`). If a source file requires the use of functions from another source file, a corresponding header file must be declared and included in the source file that needs it. The code must be broken down into independent modules (".c" files).

If the purpose of including one module within another is to take advantage of the compiler's interprocedural optimizations (*inlining*, constant propagation, *etc.*), it is preferable to rely on Link Time Optimization (LTO), for instance with GCC by using the `-flto` flag when compiling **and** at link time: `gcc -o binary -flto file1.c file2.c`.

RULE
9

RULE – Do not include a source file in another source file

Only the inclusion of header files is authorised in a source file.

Bad example

In the following example, a source file inclusion is performed, which is prohibited.

```c
/* file1.c */
#include <stdint.h>

void foo(uint16_t val) {
  ...
}

/* file2.c */
#include "file1.c" /* prohibited */

void bar() {
  foo(MAGIC_VALUE);
}
```

Good example

The example below correctly breaks down the code into different modules.

```c
/* header.h */
#ifndef HEADER_H
#define HEADER_H

void foo(uint16_t val);

#endif
```

```
/* file1.c */
void foo(uint16_t val) {
  ...
}

/* file2.c */
#include <stdint.h>
#include "header.h"

#define MAGIC_VALUE 42U

void bar() {
  foo(MAGIC_VALUE);
}
```

4.3 Format of a file inclusion directive

Different file systems do not behave in the same way: some file systems are case sensitive, and the separator of the constituents of a path may vary.

When an operating system-specific path separator is used, this prevents the portability of the source code. When an `#include` directive includes a path to the header file to be included, the separating character for the path components must be the slash “/”, not the backslash “\”, to ensure portability of the source. In addition, the character “\”, as well as the following characters or sequences of characters: `'`, `"`, `/*` and `//` located between opening and closing chevrons (< and >) or between double quotes (*i.e.* `"`) lead to undefined behaviour.

Directory and file name case must also be preserved.

RULE 10

RULE — File paths must be portable and case sensitive

File paths, whether for an `#include` inclusion directive or not, must be portable while respecting the case of directory names.

Bad example

In the following example, portability is not assured and leads to undefined behaviour.

```
#include <sys\stat.h>
#include "Module_A\Sub_Module_A\Header.h"
```

Good example

The example below uses a correct format for including header files.

```
#include <sys/stat.h>
#include "module_a/sub_module_a/header.h"
```

In addition, certain specific rules must be respected in the header file name to avoid undefined behaviour.

RULE 11

RULE – The name of a header file must not contain certain characters or sequences of characters

The name of a header file must contain none of the following characters and character sequences: `'`, `"`, `\`, `/*` and `//`.

4.3.1 References

[Misra2012] Rule 20.2: The ”,” or ”\” characters and the ”/*” and ”//” character sequences shall not occur in a header file name.
[Misra2012] Rule 20.3: The `#include` directive shall be followed by either a `<filename>` or `"filename"` sequence.
[AnsiC99] Sections 6.4.7 and 6.10.2.

4.4 Comment and definition of preprocessor blocks

The compilation directives `#if`, `#ifdef`,`#ifndef`, `#else`, `#elif` and `#endif` form blocks. These may be long and impossible to display on a single screen. They may also contain nested blocks. It can then be very difficult to determine the interdependencies between the different directives. These directives must therefore be commented on carefully to explain the cases dealt with and the sequence of the different directives.

RECO 12

RECOMMENDATION – Preprocessor blocks must be commented on

Preprocessor block directives must be commented on in order to clarify the cases being dealt with and, in the case of “intermediate” and “closing” directives, these must also be associated with the corresponding “opening” directive by means of a comment.

For reasons of readability, double negation in preprocessor directive expressions should be avoided, typically by using `#ifndef` and a “non-mode” (*i.e.* a mode defined via the negation of another mode such as `NDEBUG`).

GOOD PRACTICE 13

GOOD PRACTICE – Double negation in the expression of preprocessor block conditions should be avoided

Finally, it is essential that all the directives associated with a preprocessor block (*i.e.* “opening”, “intermediate” and “closing” directives) are present in the same file. Furthermore, it should only be possible to evaluate the control conditions used in these directives to 1 or 0.

RULE 14

RULE – Definition of a preprocessor block in a single file

For a preprocessor block, all associated directives must be found in the same file.

RECO 15

RECOMMENDATION — Preprocessor directive control expressions must be correctly formed

Control expressions must be evaluated only to 0 or 1 and should only use defined identifiers (via #define).

Bad example

The following code should be modified to add comments and not use double negation for the #else and #endif directives.

```
/* file1.c */
#ifdef A /* no #endif */
#include "log.h"

/* log.h */

#endif /* #endif of file1.c */

#ifndef LOG_H
#define LOG_H

typedef enum {
  DEBUG = 0,
  WARN,
  INFO,
  ERROR,
  FATAL
} LogLevel_T;

void log_msg(LogLevel_T level, const unsigned char* sLogMessage);

#ifndef NDEBUG  /* double negation */
#define LOG_DEBUG(msg)  log_msg(DEBUG, (msg))
#define LOG_WARN(msg)   log_msg(WARN, (msg))
#define LOG_INFO(msg)   log_msg(INFO, (msg))
#define LOG_ERROR(msg)  log_msg(ERROR, (msg))
#define LOG_FATAL(msg)  log_msg(FATAL, (msg))
#else
#define LOG_DEBUG(msg)
#define LOG_WARN(msg)
#define LOG_INFO(msg)
#define LOG_ERROR(msg)
#define LOG_FATAL(msg)
#endif
#endif
```

Good example

In the following example, the directives are correctly commented on and the associated directives are in the same file.

```
/* file1.c */
#ifdef A
#include "log.h"
#endif

/* log.h */
#ifndef LOG_H
#define LOG_H

typedef enum {
  DEBUG = 0,
  WARN,
  INFO,
  ERROR,
  FATAL
} LogLevel_T;

void log_msg(LogLevel_T level, const unsigned char* sLogMessage);

#ifdef NDEBUG /* double negation removed */
/* Not debug mode */
#define LOG_DEBUG(msg)
#define LOG_WARN(msg)
#define LOG_INFO(msg)
#define LOG_ERROR(msg)
#define LOG_FATAL(msg)
#else /* #ifdef NDEBUG */
/* Debug mode */
#define LOG_DEBUG(msg)  log_msg(DEBUG, (msg))
#define LOG_WARN(msg)   log_msg(WARN, (msg))
#define LOG_INFO(msg)   log_msg(INFO, (msg))
#define LOG_ERROR(msg)  log_msg(ERROR, (msg))
#define LOG_FATAL(msg)  log_msg(FATAL, (msg))
#endif /* #ifdef NDEBUG */
#endif /* #ifndef LOG_H */
```

4.4.1 References

[Misra2012] Rule 20.8 The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1.
[Misra2012] Rule 20.9 All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #defined before evaluation.
[Misra2012] Rule 20.14 All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related.

4.5 Using the preprocessor operators # and

The evaluation order of several # (*stringification* or character string conversion operator) or ## (concatenation operator) or the mixture of these two operators is not specified.

RULE 16

RULE — Do not use more than one of the preprocessor operators # and ## in the same expression

It is also important, with these two operators, to have a good understanding of how it works, *i.e.* the steps resulting from the macro replacement.

RULE 17

RULE — Understand the macro replacement when using the preprocessor operators # and

Bad example

```
#include <stdio.h>
...
#define MYPRINT(s) printf(#s)
#define TWO 2

int main(void)
{
  MYPRINT(TWO); /* prints "TWO" */
  return 1;
}
```

Good example

```
#include <stdio.h>
...
#define MYPRINT2(s) PRINT(s) /* Additional indirection to expand
                                "TWO" */
#define PRINT(s) printf(#s)
#define TWO 2

int main(void)
{
  MYPRINT2(TWO); /* prints "2" */
  return 1;
}
```

4.5.1 References

[Misra2012] Rule 20.10 The # and ## preprocessor operators should not be used.
[Misra2012] Rule 20.11 A macro parameter immediately following a # operator shall not immediately be followed by a ## operator.
[Misra2012] Rule 20.12 A macro operator used as an operand to the # or ## operators which is itself subject to further macro replacement, shall only be used as an operand to these operators.
[Cert] PRE05-C Understand macro replacement when concatenating tokens or performing stringification.
[AnsiC90] Section 6.8.3.
[AnsiC99] Section 6.10.3.

4.6 Specific naming of macros

It is not always easy to distinguish the use of a preprocessor macro in the source code. The use of some macros may resemble function calls.

Furthermore, when a macro is not named in upper case, there is a risk that the name corresponds to a real function name or even a reserved word in the C language. This can therefore lead to the substitution of a function call with the code replaced by the preprocessor, or even to undefined behaviour.

RULE 18

RULE — Macros must be specifically named

To easily differentiate macros from functions and not use a reserved name from another C macro, preprocessor macros must be uppercase. In addition, the words making up the name must be separated by the underscore character "_", but without starting them with the underscore character, as this is a convention for reserved names in the C language.

Bad example

In the following example, the macro naming rule is not followed.

```c
#define cte 0x7EU /* lower case */
#define _my_squared(a) ((a)*(a)) /* lower case and starts with _ */
```

Good example

The following example shows suitable naming of the preprocessor macros.

```c
#define CTE 0x7EU
#define MY_SQUARED(a) ((a)*(a))
```

4.6.1 References

[Misra2012] Rule 5.4 Macro identifiers shall be distinct.
[Misra2012] Rule 5.5 Identifiers shall be distinct from macro names.
[Misra2012] Rule 21.1: A #define or #undef shall not be used on a reserved identifier or reserved macro name.
[Misra2012] Rule 21.2: A reserved identifier or macro name shall not be declared.
[Misra2012] Rule 20.4: A macro shall not be defined with the same name as a keyword.
[IsoSecu] Using Identifiers that are reserved for the implementation [resident].
[Cert] Rule DCL37-C Do not declare or define a reserved identifier.
[Misra2012] Dir. 4.5 Identifiers in the same name space with overlapping visibility should be typographically unambiguous.
[Cert] Rec. DCL02-C Use visually distinct identifiers.

4.7 A macro must not end with a semicolon

Macros are used to make it easier to read the code and avoid repeating the same code pattern several times. When expanding a macro, if the macro definition contains a semicolon, this one is also expanded, which can cause a complete and unexpected change in the control flow.

RULE 19

RULE — Do not end a macro with a semicolon

The final semicolon should be omitted at the end of the definition of a macro.

Bad example

The macro is not defined by applying the rule and ends with a semicolon.

```
#define SQUARED(n) (n) = (n) * (n); /* no semicolon at the end of a macro */
...
if (x >= 0)
  SQUARED(x); /* conditional without brace */
else
  x = -x;
...
```

on expansion, we have:

```
#define SQUARED(n) (n) = (n) * (n);
...
if (x >= 0)
  x= x * x;
; /* empty statement */
else  /* parsing error before the else */
  x = -x;
...
```

Good example

The macro is corrected:

```
#define SQUARED(n) (n) = (n) * (n)
/* ... */
if (x >= 0)
{
  SQUARED(x);
}
else
{
  x = -x;
}
```

on expansion, we have:

```
 #define SQUARED(n) (n) = (n) * (n)
/* ... */
if (x >= 0)
{
    x = x * x;
  }
else
{
  x = -x;
}
```

4.7.1 References

[Cert] Rec. PRE11-C Do not conclude macro definitions with a semicolon.

4.8 Give preference to `static` inline functions in “function type” macros

Inline functions have been available since the C99 version of the C language.

Information

As mentioned in section 13.5, *inline* functions must also be declared as `static`.

The use of a `static inline` function to replace these “function type” macros prevents errors in the order of operator evaluation when *inlining* macros, and makes the code easier to read.

Information

It is important to note that unused `static inline` functions may cause warnings to be issued on compilation for some compiler versions. In particular, some versions of the CLANG compiler issue warnings in this case, unlike the GCC compiler. This difference in behaviour between compilers occurs when the `-Wunused-function` option is enabled at compile-time, either explicitly or via other options such as `-Wall`. When the associated code really cannot be deleted (in a library for example), the developer may have to use compiler extensions to silence these warnings, but these additions must be clearly commented on and justified.

RECO 20

RECOMMENDATION — Use `static inline` functions instead of multi-statement macros

In addition to the above recommendations and rules, it is important to add that macros whose replacement defines functions in the code should not be used. The associated risk of error is too great and the readability of the code can suffer from this kind of practice.

RULE 21

RULE — The replacement of a developer-defined macro must not create a function

4.8.1 References

[Misra2012] Dir. 4.9: A function should be used in preference to a *function-like macro* where they are interchangeable.
[Cert] Rec. PRE00-C Prefer inline or static functions to function-like macros.

4.9 Multi-statement macros

The use of macros with multiple statements can lead to unexpected behaviour. Indeed, when defining a macro with several statements, the character “\” must be used to indicate to the preprocessor that a line break must be inserted. This makes the defined macro difficult to read and can also be a source of errors.

The grouping of statements in a do { ... } while(0) loop limits the possibilities of unexpected behaviour. A do { ... } while(0) loop is always executed exactly once and prevents changing the control flow of the function calling the macro by grouping all its statements in a loop.

RULE
22

RULE — Macros containing multiple statements must use a do { ... } while(0) loop for their definition

Bad example

The macro does not implement the rule.

```
#define HALF_SUM(a,b,c,d) \
  (a) = ((c) + (d)) / 2; \
  (b) = ((c) - (d)) / 2
/* leads to a different behaviour to the one required with a call in a conditional
    statement without braces */

  if(c > d)
    HALF_SUM(a, b, c, d);
  else
    /* ... */
```

Even if the macro were defined between braces:

```
#define HALF_SUM(a,b,c,d) { (a) = ((c) + (d)) / 2; (b) = ((c) - (d)) / 2 }
```

the replacement of the macro in the same conditional is always problematic because of the lack of braces in the conditional and the “;” following the call of the macro:

```
 if(c>d)
     { (a)=((c) + (d)) / 2; (b) = ((c) - (d)) / 2 };
 else
    /* ... */
```

Good example

In the following example the macro is correctly defined using a *do-while(0)* loop structure.

```
#define HALF_SUM(a, b, c, d) \
    do {                       \
        (a) = ((c) + (d)) / 2; \
        (b) = ((c) - (d)) / 2; \
    } while(0)
```

4.9.1 References

[Cert] Rec. PRE10-C Wrap multistatement macros in a do-while loop.

4.10 Arguments and parameters of a macro

During the replacement of a macro by the preprocessor, side effects not expected by the developer may occur if the macro parameters are not protected. Parentheses must systematically be added around parameters in the definition of a macro.

RULE 23

RULE — Mandatory parentheses around the parameters used in the body of a macro

The parameters of a macro must always be enclosed in parentheses when used, in order to preserve the desired order of evaluation of the expressions.

In general, it is best to avoid arguments of a macro resulting in an operation in the broadest sense. Apart from the side effects, even if the operation performed as an argument is constant for a given input, the performance of the code is not optimal.

RECO 24

RECOMMENDATION — Arguments of a macro carrying out an operation should be avoided

Moreover, if the operation carried out by the arguments of a macro leads to a side effect in the sense of compilation, this can also lead to unexpected behaviors such as multiple evaluations of the arguments of the macro or even to no evaluation at all.

RULE 25

RULE — Arguments in a macro must not contain side effects

Macro arguments with side effects can lead to unwanted multiple evaluations.

Finally, the use of preprocessor directives (`#define`, `#ifdef` ...) in macro arguments leads to undefined behaviour and is therefore to be avoided.

RULE 26

RULE — Do not use preprocessor directives in macro arguments

Bad example

In the following example, the result will not be the one expected upon execution.

```
#define ABS(x) (x >= 0 ? x : -x)

a = c + ABS(a - b) + d;
/* result:  a = c + (a - b >= 0 ? a - b : -a -b) + d */

m=ABS(n++);
/* additional increment of n:  m = ((n++ < 0) ? - n++ : n++) */
```

Good example

The following code defines a macro with parentheses correctly placed around its argument.

```
#define ABS(x) (((x) >= 0) ? (x) : -(x))

a = c + ABS(a - b) + d;
```

```
/* correct result: a = c + (((a - b) >= 0) ? (a - b) : -(a - b)) + d */

p=n++;
m=ABS(p);

/* only one increment of n:  m = (((p) < 0) ? - (p) : (p)); */
```

4.10.1 References

[Misra2012] Rule 20.7: Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses.
[Cert] Rec. PRE01-C Use parenthesis within macros around parameters names.
[Cert] Rec. PRE02-C Macro replacement lists should be parenthesized.
[Cert] Rule EXP30-C Do not depend on the order of evaluation for side effects.
[Cert] Rule. PRE31-C Avoid side effects in arguments to unsafe macros.
[Cert] Rule. PRE32-C Do not use preprocessor directives in invocations of function-like macros.
[Cert] Rec. PRE12-C Do not define unsafe macros.

4.11 Using the `#undef` directive

Use of the `#undef` directive frequently leads to confusion. Inadvertently, its use can lead to partial code deletion if the inclusion of the code is in fact controlled by the symbol whose definition is deleted. It must never be necessary to delete the definition of a preprocessor symbol. If the purpose of deleting the symbol is to limit its scope, it is preferable to check why the scope of the symbol needs to be limited.

The use of `#undef` may result from the risk of a clash in the name chosen for a preprocessor symbol. The symbol name must then be changed to prevent this clash.

RULE 27

RULE — The `#undef` directive should not be used

4.11.1 References

[Misra2012] Rule 20.5 #undef should not be used.

4.12 Trigraph and double question mark

Two successive question marks in C mark the beginning of a sequence associated with a trigraph. For example, the trigraph "`??-`" represents the character "~". All trigraphs will be replaced prior to preprocessor directives, regardless of the location of the trigraph. They must not therefore be used.

RULE
28

RULE — Do not use trigraphs

In addition, to avoid confusion with a trigraph, all comments, strings and other literals should not contain two successive question marks.

RECO
29

RECOMMENDATION — Successive question marks should not be used

This rule applies to all parts of the code, but also to comments.

Information

The `-Wtrigraphs` option issues an alert when a trigraph is detected.

Information

By default, trigraphs are disabled in GCC.

4.12.1 References

[Misra2012] Rule 4.2 Trigraphs should not be used.
[Cert] Rec. PRE07-C Avoid using repeated question marks.

5
Compilation

Compilation is an important step when developing software since it makes the connection between the code written by the developer and the code that will actually execute on the end user's machine. Mastering it is thus primordial. Moreover, the compiler is a precious ally to detect programming errors or dangerous uses of the language, and it also generally provides hardening features capable of strongly improving the security of the produced software.

Warning

In this guide and *a fortiori* this chapter, two compilers are frequently used for illustration purposes: GCC [GccRef] and CLANG [ClangRef]. This choice is largely motivated by the popularity of these compilers, which on top of that are open source. It doesn't mean in any way that this guide only recommends using one of these two compilers. Any alternative may be proposed but the developper shall transpose the various options presented in this guide himself.

5.1 Mastery of the compilation phase

Compilers offer different warning levels meant to inform the developper of the use of risky constructions or the presence of programming errors. The default level is usually rather low and reports few bad practices. It is thus insufficient and must be increased, which requires making used compilation options explicit. Furthermore, for the same version of the C standard, some default behaviors may vary from one compiler to another. Even warnings emitted when compiling are directly tied to the compiler version. It is thus of primary importance to precisely know the compiler in use, its version as well as all enabled options, ideally backed with justifications.

RULE
30

RULE — Precisely define compilation options

Options used for compiling must be precisely defined for the whole software sources. These options should in particular accurately establish:

- the C standard version in use (for instance C99 or C90);
- the name and version of the compiler in use;
- the warning level (for example `-Wextra` for GCC);
- preprocessor symbols definitions (for instance defining `NDEBUG` when compiling in release mode).

Moreover, any developer enabling compiler or linker options must be fully aware of the consequences regarding security of the generated executable or library.

RECO 31

RECOMMENDATION — Master actions performed by the compiler and the linker

The developer must know and document all actions stemming from enabled compiler and linker options, including when such options have to do with code optimization.

Warning

In particular, the use of compilation options such as `-fno-strict-overflow`, `-fwrapv`, `-fwrapv-pointer`, `-fno-delete-null-pointer-checks` or `-fno-strict-aliasing` is most of the time indicative of a risky usage of the C language.

Using a build automation software like *make*, *CMake* or *Meson* facilitates management of compilation options. The latter may be defined globally and applied to all source files to compile.

GOOD PRACTICE 32

GOOD PRACTICE — Make use of build automation software

5.1.1 References

[Misra2012] Subsection 4.2. Understanding the compiler.
[Cert] MSC06-C Beware of compiler optimizations.
[Cert] PRE13-C Use the standard predefined macros to test for versions and features.
[Cwe] CWE-14 Compiler Removal of Code to Clear Buffers.

5.2 Compilation without errors nor warnings

Making sure that the code compiles without any error nor warning is an excellent way to decrease the risk of programming errors or risky constructions remaining in the code base. Obviously, the idea is not to lower the stringency of compilation options in order to achieve this objective, but to actually fix all issues reported by the compiler. By default, compilation options shall be as strict as possible with the aim of increasing the stringency of the compiler to the fullest.

RULE 33

RULE — Compile the code without any error nor warning while enabling strict compilation options

High warning and error levels of the compiler and the linker must be enabled to ensure, as much as possible, the absence of potential issues related to incorrect use of the programming language.
All warnings and all errors reported by the compiler and the linker must be dealt with. It is incidentally very much advised, if using GCC or CLANG, to use the `-Werror` option in order to turn any warning into a compilation error, hence not running the

risk of ignoring it.

The relevance and accuracy of warnings emitted by the compiler directly depends on the precision of analyzes it conducts, which in turn rely on the various optimizations the compiler is able to perform. Specifying a reasonably high optimization level is thus beneficial.

RULE 34

RULE — Enable a reasonably high optimization level

For GCC and CLANG, the optimization level must be at least `-O1`, and ideally `-O2` or `-Os`.

Warning

It is the responsibility of the developer to make sure that a high optimization level does not suppress defensive code or manually added software countermeasures.

Information

For example, the minimal command line for compiling with GCC or CLANG is: `gcc/clang -O1 -Wall`[a] `-Wextra`[b] `-Wpedantic`[c] `-Werror -std=c99/c90`[d] `file.c -o file.exe`

RECO 35

RECOMMENDATION — Use the strictest compilation options

If a compilation option proves to be too strict for a given development and a choice is made to disable it, a justification shall be provided to explain it.

Information

Appendix B.2 draws up a non-exhaustive list of extra warnings for GCC and CLANG, which can serve as a starting point to the developer.

In order to suppress errors and warnings, the first thing to do is of course to fix the source code, while imperatively commenting on any resulting code edit. However, if it appears to be a false positive, several methods exist. Firstly, the complexity of a code snippet may occasionally suffice to mislead the compiler analysis, and it is then beneficial in general to simply rewrite this hunk in a more intelligible form, while obviously making sure it is semantically equivalent. Then, if it turns out that a warning cannot be eliminated by fixing the source code, and if the compiler allows for it (via a `#pragma` directive for example), this warning may be disabled locally.

[a] Enables all the warnings about risky language constructions that are easy to avoid. See the GCC and CLANG compilers manuals for a complete listing.

[b] Enables some extra warning flags that are not enabled by `-Wall`. See the GCC and CLANG compilers manuals for a complete listing.

[c] Enables all the warnings demanded by the C standard; disables all compiler extensions, including the ones that do not conflict with the standard.

[d] Specifies the C standard used by the compiler. See appendix B.1.

Warning

Using #pragma to suppress warnings can be quite dangerous and must therefore be perfectly understood in order not to disable one or more warnings in the whole code base by mistake. Moreover, as the use of #pragma is not standard, the developer must keep in mind that this is specific to each compiler (implementation-defined) and thus risky.

In case the developer opts for warning suppression, a clear justification needs to be provided with a comment:

```
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wunused-variable"
/* code */
/* some variables are not used in the new algorithm anymore but are kept for API compatibility */
/* warnings about such unused variables are thus disabled in the code following the pragma */
#pragma GCC diagnostic pop
```

5.2.1 References

[Misra2012] Dir. 2.1: All sources files shall compile without any compilation errors.
[Misra2012] Dir. 4.1: Run-time failures shall be minimized.
[Cert] MSC00-C: Compile cleanly at high warning levels.
[Cwe] CWE-563 Unused variable.
[Cwe] CWE-570 Expression is always false.
[Cwe] CWE-571 Expression is always true.

5.3 Use of security features provided by compilers

Modern compilers offer various options that make it possible to improve the robustness and defensiveness of the final executable. It can come to prevent a vulnerability from surfacing or to reduce its security impact, but also to harden the program against vulnerability exploitation attempts.

RULE 36

RULE – Make use of security features offered by compilers

Developers must, as much as possible, take advantage of compilation options that allow for improving the security of the final software product.

Warning

Throughout this section, when GCC or Clang options are given as examples, it is necessary to keep in mind that:

- these options apply to GCC 11 and Clang 13 respectively;
- some of them are already enabled by default, sometimes partially, depending on the compiler and its version, but it is preferable to specify them anyway;
- the accuracy of warnings emitted by the compiler may depend upon the selected optimization level;

- the impact on performances, if mentioned, is given for information purposes only and may vary a lot according to use cases; it is therefore the developer's job to ensure compliance with his needs.

5.3.1 Warnings oriented towards security bugs

A number of warnings offered by compilers focus very especially on detecting potential security issues. Enabling these warnings and dealing with any issue they report are thus a first step in reducing the risk of software vulnerabilities being introduced in the code base.

RULE 37

RULE – Enable warnings that focus on detecting security bugs and deal with any reported issue

For instance, the `-Wformat=2`[a] GCC and CLANG compilation option must be enabled and any reported issue needs to be methodically dealt with.
GCC options `-Wformat-overflow=2` and `-Wformat-truncation=2` may also be used.

Information

The GCC and CLANG compilers nowadays embed static source code analyzers capable of running deeper and more precise — but more expensive — analyses on programs in order to detect more programming errors, especially potential security vulnerabilities.
Even though these embedded analyzers are still rather basic and in the end yet quite experimental compared to standalone tools that are dedicated to static source code analysis, it may be convenient and interesting to use them. The interested developer may refer to the documentation [GccRef] of GCC option `-fanalyzer` as well as the *Clang Static Analyzer* section from the CLANG documentation [ClangRef].

5.3.2 Instrumentation of some particularly unsafe functions

The use of functions that handle memory or character strings is a large source of programming errors that frequently lead to vulnerabilities such as buffer overflows. In addition to choosing less dangerous versions of such functions when available (*cf.* rules from section 17.4), some compilers are capable of automatically enhancing them with simple checks, performed at compile or execution time, to detect potential buffer overflows.

RULE 38

RULE – Enable the use of hardened variants of unsafe functions

For instance, when using GCC to compile a program intended for a GNU/Linux system running the glibc, the `_FORTIFY_SOURCE` macro must be defined. The optimization level must be greater or equal to `-O1` for the added checks to be effective.
Run-time checks terminate the program as soon as an overflow is detected.

[a] This option adds additional compile-time checks on format strings, as well as on function calls that take them as arguments.

Information

Option `-Wstringop-overflow=n` of GCC allows for performing similar checks at compile-time only. Their stringency depends on the value of `n`.

5.3.3 Initialization of automatic variables

The use of uninitialized automatic variables[3] is another common error and source of vulnerabilities (see section 6.10 as well). A number of compilers however are capable of detecting some of these uses, as long as the appropriate warnings are enabled.

RULE 39

RULE — Enable compiler warnings related to the use of uninitialized variables

In particular, options `-Wuninitialized`[a], `-Winit-self` and `-Wmaybe-uninitialized`[b] must be enabled when GCC is used.
As for CLANG, options `-Wuninitialized`[c] and `-Wconditional-uninitialized` must be enabled.

Warning

It should be noted that these warnings do not detect all instances of uninitialized automatic variable uses, especially when such variables are passed to other functions by reference.

Furthermore, some compilers support automatic initialization of such variables. In practice, this forced initialization can use either the value zero or another particular value, called *pattern*. Automatic initialization to zero should be chosen for *release* builds (*cf.* next section 5.4) because it usually limits the exploitability of this type of bug. In contrast, during development, testing and debugging phases, pattern initialization is preferable since it is more likely to uncover certain bugs. Selecting the right pattern is then essential: for example, for pointer-type variables, it may be a non-canonical address so that any memory access through an uninitialized pointer systematically faults.

RULE 40

RULE — Enable forced initialization of automatic variables by the compiler

CLANG supports automatic initialization with the two aforementioned approaches:

- `-ftrivial-auto-var-init=pattern` for development, tests and debugging;
- `-ftrivial-auto-var-init=zero` for *release* builds, an option that currently necessitates appending option `-enable-trivial-auto-var-init-zero-knowing-it-will-be-removed-from-clang`.

[3] An automatic variable is a variable defined within a function, without the `static` storage class specifier. Its storage is allocated and deallocated automatically on the call stack.
[a] Automatically enabled by `-Wall`
[b] Automatically enabled by `-Wall`
[c] Automatically enabled by `-Wall`

Warning

Even when all automatic variables are forcibly initialized by the compiler, their use without prior explicit initialization by the developer is still a programming error that absolutely has to be fixed.
Automatic initialization by the compiler is thus a hardening only intended to limit the security impact of this type of bugs, and as a consequence the developer should never rely on this behavior.

5.3.4 Integer overflows

Signed integer overflows are not defined by the C standard and are thus particularly dangerous. For instance, depending on hardware architectures and compilers, a variable of type `int` reaching the value `INT_MAX` can *wrap* after another increment, that is become `INT_MIN`, which may prove to be quite problematic especially in the case of a variable that represents a reference counter for a memory allocation. The compiler may be able to detect certain kinds of signed integer overflows.

RECO
41

RECOMMENDATION — Enable compiler options that allow for detecting signed integer overflows

In particular, GCC and CLANG both support option `-ftrapv`, which makes the compiler instrument the source code in such a way as to emit a trap during program execution for any signed integer overflow on addition, subtraction or multiplication operations.

Warning

Even though **unsigned** integer overflows are actually a well-defined behavior of the C language, they are not any less dangerous and may just as well lead to the introduction of bugs and vulnerabilities in a piece of software. The developer should thus remain particularly careful when performing operations prone to overflows, even with unsigned operands.

Information

GCC and CLANG support many other options that are useful for detecting integer overflows, but they are part of the UBSan[a] *sanitizer*, the use of which as well as of other sanitizers is not addressed by this guide.

5.3.5 Call stack hardenings

In order to make it harder to exploit certain vulnerabilities, the memory area corresponding to the program stack should not be executable. Modern toolchains generally endeavor to enforce this rule by default.

[a] Undefined Behavior Sanitizer

Nonetheless, the use of nested functions, as supported by GCC, imposes an executable stack[4] and is thus prohibited[5]. Moreover, supporting this feature has complicated the way GNU linkers — namely BFD and gold — choose to mark an executable or a library as requiring executable stack; it is thus necessary to be very cautious when using them.

RULE 42

RULE — Do not use executable stack

In particular, GCC nested functions should not be used. Besides, for software intended for GNU/Linux or FreeBSD environments:

- option `-z execstack` of BFD, gold and lld linkers should **not** be used;
- option `-z nognustack` of lld should **not** be used;
- option `-z noexecstack` of BFD and gold **must** be used.

Stack buffer overflows certainly rank among the oldest and most common memory corruptions. Positioning guard variables with a random value, usually called *canaries*[6], makes it possible for example to detect some linear overflow attempts that try to overwrite the value of a return address saved on the stack. If need be, program execution is automatically terminated.

RULE 43

RULE — Enable stack canaries

With GCC and CLANG, compilation option `-fstack-protector-strong` must be enabled.
Section 19.2 also tackles manual implementation of a canary mechanism.

Depending on hardware architectures, platforms and compilers, it is also possible to use a guard variable whose value is different for each thread within the same process. This allows for reducing the severity of a potential leak of the canary value.

RECO 44

RECOMMENDATION — Use per-thread canaries

In particular, for x86 architectures, GCC and CLANG support option `-mstack-protector-guard=tls`, relying on glibc *Thread Local Storage*.

5.3.6 Dynamic loading

Randomization is a probabilistic security defense tactics aiming to make software vulnerability exploitation less reliable. In particular, common toolchains are able to produce executables that can be loaded at a random memory address, in order to make the most of address space layout randomization (ASLR) as implemented by the operating system.

[4] More precisely, it is the use of *closures*, implemented by GCC by means of such nested functions and of trampolines located on the stack, that is problematic.

[5] It is a language extension anyway, offered by a compiler, yet, as a reminder, only C code compliant with the C90 or C99 standards is allowed by the present guide.

[6] Alternatively known as *cookies*

RULE
45

RULE — Produce position independent executables

With GCC and CLANG, compilation option `-fPIE` must be enabled, as well as option `-pie` of BFD, gold and lld linkers.

To allow for relocation of shared libraries and executables, the dynamic loader must be able to modify some of their sections. If the corresponding memory mappings consequently remain writable during program execution, they may prove useful to an attacker trying to exploit a software vulnerability. However, it is possible at link-time to mark said sections so that the dynamic loader makes the underlying memory mappings read-only as soon as possible. This is referred to as `relro` or *partial* `relro` mode.

RULE
46

RULE — Use `relro` mode of linkers

For instance, with BFD and gold linkers, option `-z relro` must be used. lld enforces `relro` mode by default thus no extra option is required.

Nevertheless, since function symbols resolution usually happens in the course of program execution (*lazy binding*), a number of sections within shared libraries or executables remain writable regardless of `relro` mode. Is is then possible to force the dynamic loader to resolve all symbols when the program is started[7] so that it can go on to make the underlying memory mappings read-only. This is referred to as *full* `relro` or `BIND_NOW` mode.

RECO
47

RECOMMENDATION — Do not use lazy binding

For instance, option `-z now` of BFD, gold and lld causes these linkers to mark generated executables and libraries so as to tell the dynamic loader it needs to resolve all symbols at program startup.

i

Information

When `relro` mode is used and *lazy binding* is disabled, some linkers reorder sections inside generated binaries in order to prevent overflows of data located in one section from overwriting the contents of sensitive sections.
This is notably the case with BFD, gold and lld.

5.3.7 Reproducible builds

Among other things, reproducible builds allow users of a particular software to verify independently that a binary they are provided with is indeed the unaltered product of a precise state of its sources, by rebuilding it and comparing the result. This requires a fully deterministic build process, which is nontrivial and goes beyond the scope of this guide, but taking this problematic into consideration as soon as possible in a project can greatly facilitate the implementation of such a feature the day that it becomes an objective.

[7] Disabling *lazy binding* might thus slow down a large software's **startup**. It should also be noted that this becomes of little importance in practice for a *daemon*.

GOOD PRACTICE 48

GOOD PRACTICE — Ensure reproducible builds

For example, compilation options such as `-Wdate-time` from GCC and CLANG or `-frandom-seed=` from GCC may prove useful to limit the introduction of nondeterminism at compile-time.

5.4 Debug and release modes

Build modes *debug* and *release* are usually both available with any compiler and are very useful for software development because of the significant changes they can induce on build products.

Debug and release modes

In *debug* mode, which is mainly designed for debugging while developing, most optimizations are disabled and debugging information for all symbols is preserved, making it easier in particular to set breakpoints. Compilation is faster and uses less memory, but the generated code will generally be larger and slower to execute. In *release* mode, which corresponds to the final mode suitable for client delivery or deploying to production, optimizations are enabled and information that does not add value to program execution, such as symbols, is removed. Compilation takes more time then and consumes more memory, but allows for generating more complex machine code that will thereby be faster and more compact.

Debug mode enables the developer to better understand how a program works and fix reported errors whereas *release* mode is required for delivery for performance or program size reasons. For instance, in *debug* mode, some compilers make sure that all variables are automatically initialized to `0`, while in *release* mode this only applies to global variables, as per the standard.

Information

If `NDEBUG` is defined as a macro name at the point in the source file where the standard library header `assert.h` is included, then the `assert` macro is redefined to be disabled.

RULE 49

RULE — All production-ready code must be compiled in release mode

Compiling in *release* mode is mandatory when putting software into production.

This can seem redundant with rules and recommandations from sections 5.1 and 5.2 but it is definitely a rather common mistake in software engineering. In addition to different behaviors regarding memory management and code optimizations, *debug* mode may even sometimes increase software attack surface. It is thus very important that the developer make use of these modes in full knowledge of the cause.

RECO
50

RECOMMENDATION — Pay special attention to debug and release modes when building a project

The use of *debug* and *release* modes at compile-time must be done while making sure that all induced changes regarding memory management and optimization are well known. Every difference between these two modes must be documented exhaustively.

6

Declaration, definition and initialisation

Definition versus use of variables

Defining a variable means assigning a value to it (*i.e.* writing to the variable's memory address), while using a variable means reading the value of the variable (*i.e.* reading the value stored at the associated memory address).

6.1 Multiple variable declarations

The C language allows the sumultaneous declaration of several variables of the same type. In multiple variable declarations, each variable is separated with a comma. These multiple declarations are used to associate a given type with a group of variables or to group together related variables. However, this type of multiple declaration should only be used on simple variables (no pointers or structured variables) of the same type.

RECO
51

RECOMMENDATION — Only multiple declarations of simple variables of the same type are authorised

In order to also avoid errors in the initialisation of variables, initialisations coupled with a multiple declaration are to be prohibited. Indeed, in the case of a single initialisation at the end of the multiple declaration, only the last variable declared is actually initialised.

RULE
52

RULE — Do not make multiple variable declarations associated with an initialisation

Initialisations associated (*i.e.* consecutive and in the same statement) with a multiple declaration are prohibited.

Bad example

In the code below, several variables are declared in the same statement, but only the last variable is initialised.

```
uint32_t abs, ord = 0; /* caution, the variable abs is not set to zero here ! */
uint32_t a, *b; /* to be prohibited: mix of simple variable and pointer
    declaration */
struct blob_t g, h[35]; /* to be prohibited: mix of simple variable, pointer and
    array declaration */
```

Good example

```
uint32_t  a,
uint32_t *b; /* separation of the simple variable and the pointer */
struct blob_t g;
struct blob_t h[35]; /* as above for separation of the array and simple variable
     */
uint32_t  abs, ord; /* joint declaration of two functionally-related variables */
abs = 0; /* assignment of the two variables */
ord = 0;
```

6.1.1 References

[Cert] Rec. DCL04-C Do not declare more than one variable per declaration.

6.2 Free declaration of variables

Since C99, variables can be declared anywhere in the code. This feature seems practical, but its abuse can make reading the code significantly more complex and may lead to possible redefinition of variables.

RECO
53

RECOMMENDATION — Group variable declarations at the beginning of the block in which they are used

For reasons of readability and to avoid redefinition, variable declarations have to be grouped at the beginning of the file, of the function or of the block statement according to their scope.

Information

This recommendation is not strictly related to security but has an impact on the readability, portability and/or maintainability of the code, and concerns all types of development.
The `-Wdeclaration-after-statement` GCC and CLANG compiler option can help with enforcing this recommendation.

A very common practice for loop counters is to declare them directly in the associated loop. This "on the fly" declaration is accepted, but a special care must be taken to ensure that the associated variable does not mask one of the other variables used in the body of the loop.

Bad example

In the following code, the variables are declared "on the fly" and not in a grouped and structured manner. This type of practice makes it more complex to identify all the variables declared, thus increasing the risk of variable shadowing.

```
#include <stdint.h>
uint8_t glob_var; /* global variable */
uint8_t fonc(void)
{
    uint8_t var1; /* local variable */
    if (glob_var >=0)
    {
        /* ... */
    }
    else
        var1=glob_var;
    uint8_t var2; /* other local variable declared in the middle of a block */
    /* ... */
}
uint8_t glob_var2; /* other global variable declared between two functions */
void main(void)
{
    uint8_t x = fonc();
    /* ... */
}
```

Good example

The variables are declared in a grouped and structured way at the beginning of the blocks, which makes them easier to read.

```
#include <stdint.h>

uint8_t glob_var; /* global variable declared together */
uint8_t glob_var2;

uint8_t fonc(void)
{
    uint8_t var1; /* local variables declared together at the beginning of the
        function */
    uint8_t var2;
    if (glob_var >= 0)
    {
        /* ... */
    }
    else
    {
        var1 = glob_var;
    }
    /* ... */
}
void main(void)
{
    uint8_t x = fonc();
    /* ... */
}
```

6.3 Declaration of constants

The direct use of numerical values (or characters and constant or *literals*) makes the source code difficult to maintain. If a value is changed, one has to remember to change all the statements in which the value is used.

RULE 54

RULE – Do not use hard-coded values

The values used in the code must be declared as constants.

The constant declaration rule is also to be applied for all types of values appearing several times in the source code. Therefore, if a character or string is repeated several times, it must also be defined using a global `const` variable or, failing that, using a preprocessor macro.

Centralising the declaration of constants ensures that the change in their value is applied across the entire implementation.

GOOD PRACTICE 55

GOOD PRACTICE – Centralise the declaration of constants at the beginning of the file

To make it easier to read, the constants are declared together at the beginning of the file.

To identify these constants, several rules must be respected.

RULE 56

RULE – Declare constants in upper case

Constants that do not require type checking are declared with the keyword `#define`.

RULE 57

RULE – Constants that do not require type checking are declared with the `#define` preprocessing directive

RULE 58

RULE – Constants requiring explicit type checking must be declared with the keyword `const`

RULE 59

RULE – Constant values must be associated with a suffix depending on the type

To avoid misinterpretation, constant values must use a suffix based on their type:

- the suffix `U` must be used for all unsigned integer type constants;
- to indicate a `long` (or `long long` for C99) type constant, the suffix `L` (or `LL` respectively) must be used instead of `l` (or `ll` respectively) in order to avoid any ambiguity with the number `1`;
- floating values are by default considered as `double`; use the suffix `f` for the float type (or `d` for the double type respectively).

Warning

By default, integer values are considered `int` type and floating values are considered `double` type.

RULE
60

RULE — The size of the type associated with a constant expression must be sufficient to contain it

It must be ensured that the constant values or expressions used do not exceed the type associated with them.

To avoid any confusion, octal constants are to be prohibited. Some cases can be tolerated such as UNIX file modes, but they will be systematically identified and commented on.

RECO
61

RECOMMENDATION — Prohibit octal constants

Do not use octal constants or escape sequences.

Bad example

The following example does not centralise the definition of the constants. Certain constants have not been declared. There is also an absence of specific naming for the constants.

```
#define octal_const 075 /* octal base numerical constant and name in lower case */

const int64_t b = 0l; /* l and not L, and constant naming problem */
uint8_t buffer[0x82]; /* constant not declared and type check needed for this
     constant */
int16_t i;

for(i = 0; i < 0x82; i++) { /* hard-coded value */
   ...
}

printf("Message\012"); /* octal base escape sequence \012 = \n */
```

Good example

The following code applies the different rules and recommendations for the declaration of constants.

```
const uint32_t INIT_VALUE = 0x1294U; /* declared constant and with the necessary
     type check */

#define BUFFER_SIZE 0x82U /* declared constant with type check not necessary */

const int64_t B = 0L; /* correction of the suffix and specific naming of the
     constant */

uint8_t buffer[BUFFER_SIZE];
uint16_t i;

for(i = 0; i < BUFFER_SIZE; i++) {
   ...
}
```

6.3.1 References

[Misra2012] Rule 11.8 A cast shall not remove a const or volatile qualification from the type pointed to by a pointer.

[Misra2012] Rule 7.1. Octal constants shall not be used.
[Misra2012] Rule 7.2. A u or U suffix shall be applied to all integer constants that are represented in an unsigned type.
[Misra2012] Rule 7.3. The lowercase character l shall not be used as a literal suffix.
[Misra2012] Rule 7.4. A string literal shall not be assigned to an object unless the object’s type is ”pointer to a const-qualifed char”.
[Misra2012] Rule 12.4 Evaluation of constant expressions should not lead to unsigned integer wraparound.
[Misra2012] Dir. 4.5 Identifiers in the same name space with overlapping visibility should be typographically unambiguous.
[Cert] Rec. DCL16-C Use L, not l, to indicate a long value.
[Cert] Rec. DECL00-C Const-qualify immutable objects.
[Cert] Rec. STR05-C Use pointers to const when referring to string literals.
[Cert] Rec. DECL18-C Do not begin integer constants with 0 when specifying a decimal value.
[Cert] Rule EXP40-C Do not modify constants objects.
[Cert] Rule STR30-C Do not attempt to modify string literals.
[Cert] Rec. EXP05-C Do not cast away a const qualification.
[Cert] Rec. DCL02-C Use visually distinct identifiers.
[Cert] Rec. DCL06-C Use meaningful symbolic constants to represent literal values.
[Cwe] CWE-547 Use of Hard-coded, security relevant constants.
[Cwe] CWE-704 Incorrect type conversion or cast.
[IsoSecu] Modifying string literals [strmod].

6.4 Limited use of global variables

When global variables are used, it is difficult to identify every function that modifies these variables. Furthermore, if a global variable is not named according to clear naming conventions, reading the code of a function using this variable does not immediately identify the side effect of the function on this global variable. This specific naming scheme must be clear and remains the choice of the developer or of the developing team (use of upper case, prefix g_, *etc.*).

In addition, the use of global variables can quickly lead to problems of concurrency in the case of a multi-tasking application. For each of the global variables, the developer must study the possibility of limiting the scope of the variable systematically.

RULE
62

RULE — Limit global variables to what is strictly necessary

Limit the use of global variables and give preference to function parameters in order to propagate a data structure through an application.

Bad example

The following code uses a global variable. However, its use could easily be avoided.

```
static uint32_t g_state;

void foo(void) {
  ...
  g_state = 1;
}

void bar(void) {
  ...
  g_state = 2;
}

int main(int argc, char* argv[]) {
  foo();
  bar();
  ...
}
```

Good example

The following example does not use a global variable. The variable `state` is propagated from function to function by passing it as a parameter:

```
void foo(uint32_t* state) {
  ...
  (*state) = 1;
}

void bar(uint32_t* state) {
  ...
  (*state) = 2;
}
int main(int argc, char* argv[]) {
  uint32_t state = 0;
  foo(&state);
  bar(&state);
  ...
}
```

6.4.1 References

[Cert] Rec. DCL02-C Use visually distinct identifiers.
[Cert] Rule DCL30-C Declare objects with appropriate storage durations.
[Cert] Rec. DCL19-C Minimize the scope of variables and functions.
[Misra2012] Dir. 4.5 Identifiers in the same name space with overlapping visibility should be typographically unambiguous.
[Misra2012] Rule 8.9 An object should be defined at block scope if its identifier only appears in a single function.

6.5 Use of the `static` keyword

When a function is declared, defined and used only within a single source file, the `static` storage-class specifier is often forgotten. Conflicts may then arise at link-time. In addition, the absence of the `static` specifier makes code review more difficult because it does not allow to quickly notice that a function is “private/local”. The `static` keyword tells the compiler that the variable/function is indeed a global variable/function but that its visibility must be limited to the source file in which it is declared.

The same applies for variables global to a file that are not used outside that file. Global variables of this type should be systematically declared as `static`. This limits the scope of these variables only to the other functions defined in the same file, and therefore limits the exposure of said variables. These global functions and variables should not be declared in a header file.

RULE 63

RULE — Systematically use the `static` specifier for declarations

The `static` storage-class specifier must be used for all global functions and variables that are not used outside the source file in which they are defined.

6.5.1 References

[Cert] Rec. DCL15-C Declare file-scope objects or functions that do not need external linkage as static.
[Cert] Rule MSC40-C Do not violate constraints.
[Misra2012] Rule 8.7 Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.
[Misra2012] Rule 8.8 The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage.

6.6 Use of the `volatile` keyword

The `volatile` keyword must be used to qualify either a variable corresponding to a hardware area that represents an input/output port in memory, or a variable read or written to by an asynchronous interrupt function. Accesses to such variables should indeed be protected from compiler optimizations.

RULE 64

RULE — Only variables that can be modified outside the implementation should be declared `volatile`

Only variables associated with input/output ports or asynchronous interrupt functions should be declared as `volatile` to prevent optimisation or reorganisation on compilation.

Moreover, to avoid undefined behaviour, only a pointer that is itself qualified as `volatile` can access a `volatile` variable.

RULE 65

RULE — Only `volatile`-qualified pointers can access `volatile` variables

6.6.1 References

[Cert] Rec. DCL17-C Beware of miscompiled volatile-qualified variables.

[Cert] Rec. DCL22-C Use volatile for data that cannot be cached.
[Cert] Rule EXP32-C Do not access a volatile object through a nonvolatile reference.
[Misra2012] Rule 2.2 There shall be no dead code.
[Misra2012] Rule 11.8 A cast shall not remove any const or volatile qualification from the type pointed to by a pointer.
[Cwe] CWE-704 Incorrect type conversion or cast.
[Cwe] CWE-561 Dead code.

6.7 Implicit type declaration is prohibited

C90 allows the implicit declaration of variables in terms of omission of the type in certain circumstances, such as for the parameters of a function, elements of a structure or the declaration of a `typedef`.

Information

In practice, compilers issue a warning (`-Wimplicit-int`) but implicitly assume that the type is `int`.

RULE 66

RULE — No type omission is accepted when declaring a variable

All variables used must have been explicitly declared before use.

Furthermore, the *K&R*[8] function declaration syntax, such as for example:

```
int foo(a,p)
   int a;
   char *p;
{ ...
}
```

is also prohibited. Firstly, this type of declaration is obsolete and, secondly, it reduces the readability of the code and therefore, potentially, the checks made at compiler level.

Bad example

The following code (C90) contains several implicit type declarations.

```
...
const ACONST = 42; /* prohibited: type of constant not explicitly defined (
    implicite int) */
unsigned e; /* prohibited: type of e not explicitly defined (implicit unsigned int)
     */
signed f; /* prohibited: type of constant not explicitly defined
(implicit signed int) */
...
int foo(char a, const b) /* prohibited: type of b not explicitly defined (implicit
    int) */
{
...
}
```

[8] Kernighan & Ritchie's C syntax before ANSI standards

```
bar(char c, const int d) /* prohibited: type of return of the function not
     explicitly defined (implicit int) */
{
..
}
```

Good example

All types are now explicit.

```
...
const int ACONST = 42;
unsigned int e;
signed int f;
...
int foo(unsigned char a, const int b)
{
...
}
int bar(unsigned char c, const int d)
{
...
}
```

6.7.1 References

[Cert] Rule DCL31-C Declare identifier before using them.
[Misra2012] Rule 8.1 Types shall be explicitly specified.

6.8 Compound literals

Compound literals were introduced by C99 and allow the creation of unnamed objects from a list of initialisation values. This construction is often used with particular structures that have been passed as function parameters. The lifetime of a *compound literal* is either static or automatic depending on whether it is declared at file or block statement level.

Attempting to access the associated object outside of its scope will result in undefined behaviour. It is therefore essential to fully understand the scope associated with this type of construction.

RECO
67

RECOMMENDATION — Limit the use of compound literals

Due to the risk of mishandling *compound literals*, their use must be limited, documented and special attention must be paid to their scope.

Bad example

```
#include <stdio.h>
#include <stdint.h>
#define MAX 10

 struct point {
   uint8_t x,y;
  };
```

```
int main(void)
{
  uint8_t i;
  struct point *tab[MAX];
  for (i = 0; i < MAX; i++){
    tab[i] = &(struct point){i,2*i};
  }
  for (i = 0; i < MAX; i++){
    printf("%d\n", tab[i]->x); /* undefined behaviour because the compound literal
        defined in the previous loop does not exist anymore */
  }
  ...
}
```

Good example

```
#include <stdio.h>
#include <stdint.h>

struct point {
  uint8_t x,y;
};

#define MAX 10
int main(void)
{
  uint8_t i;
  struct point tab[MAX];
  for (i = 0; i < MAX; i++){
    tab[i] = (struct point){i, 2*i};
  }
  for (i = 0; i < MAX; i++){
    printf("%d\n", tab[i].x);
  }
  ...
}
```

6.8.1 References

[Cert] Rec. DCL21-C Understand the storage of compound literals.
[Cert] Rule DCL30-C Declare objects with appropriate storage durations.
[IsoSecu] Escaping of the address of an automatic object [addrescape].

6.9 Enumerations

The non-explicit value of a constant in an enumeration is 1 higher than the value of the previous constant. If the first constant value is not explicit then it is 0. If all the values in the enumeration are implicit, no problem arises, but if the developer makes certain values explicit, an error is possible. It is therefore better to avoid mixing constants with explicit and implicit values. If constants of the same enumeration have the same value, this leads to undefined behaviour. If values are made explicit then all values of the enumeration constants must be made explicit to ensure that none of the given values are repeated.

RULE 68

RULE — Do not mix explicit and implicit constants in an enumeration

Either all the constants in an enumeration must be made explicit with a single value, or none at all.

The constants of an enumeration are also subject to the rules of section 6.3, such as the use of upper case for the declaration of constants.

A use sometimes observed around enumerations is the declaration of anonymous enumerations for the declaration of constants. For example:

```
enum {
  ZERO,
  ONE
};
```

is used instead of:

```
const int ZERO=0;
const int ONE=1;
```

Enumerations are not made for this purpose; it is a misuse that can make the code harder to understand.

RULE 69

RULE — Do not use anonymous enumerations

Bad example

```
enum une_enum {
  enum1=1,
  enum2,
  enum3,
  enum4=3 /* enum4 et enum3 ont la même valeur */
};
```

Good example

All constants have a unique value and are in upper case.

```
enum une_enum {
  ENUM1=0,
  ENUM2=1,
  ENUM3=2,
  ENUM4=3
};
```

6.9.1 References

[Misra2012] Rule 8.12 Within an enumerator list, the value of an implicitly-specified enumeration shall be unique.
[Cert] Rec. INT09-C Ensure enumeration constants maps to unique values.

6.10 Initialising variables before use

If variables are not initialised on declaration, there is a risk of using the variable when it has not been initialised. The behaviour is then undefined.

Information

Global and static variables are automatically initialised when they are defined, but with a default value specified by the standard. Due to the possible lack of knowledge of these default values, it is recommended to explicitly initialise all variables.

One easy way to ensure this is to do it systematically when declaring a variable if it is declared alone, or immediately after declaration for multiple declarations.

RECO 70

RECOMMENDATION — Variables should be initialised at or immediately after declaration

All variables should be systematically initialised when they are declared, or immediately afterwards in the case of multiple declarations.

Information

The compiler can detect certain missing initialisations. GCC for example provides the -Wuninitialized option. Subsection 5.3.3 gives more details and also highlights the limits of such options. In particular, the absence of warning raised by this option is not sufficient to guarantee that all variables are properly initialized.

Bad example

In the following example, variables are not initialized when they are used.

```
/* declarations in the body of a function */
uint32_t  a;
uint32_t  b;
uint32_t c;

a = b + c; /* variables are used but without initialisation */
```

Good example

In the following code, the variables are correctly initialised before being used (as soon as they are declared here).

```
/* declarations in the body of the function */
uint32_t  a = 0;
uint32_t  b = 0;
uint32_t  c = 0;

a = b + c;
```

6.10.1 References

[Misra2012] Rule 9.1: The value of an object with automatic storage duration shall not be read before it has been set.
[Cert] Rule Exp33-C Do not read uninitialized memory.
[Cwe] CWE-457 Use of uninitialized variable.
[Cwe] CWE-758 Reliance on undefined, unspecified, or Implementation-defined behavior.
[Cwe] CWE-908 Use of uninitialized Resource.
[IsoSecu] Referencing uninitialized memory [uninitref].

6.11 Initialisation of structured variables

The C language offers multiple possibilities for initialising arrays, structures and other structured variables. As there are many such possibilities, they can be confusing and can also be misinterpreted.

RULE 71

RULE – Use only one initialisation syntax for structured variables

For the initialisation of a structured variable, only one initialisation syntax must be chosen and used.

Bad example

```
int tab[10] = { 0, [4] = 3, 5, 6, [1] = 1, 2 };
struct type_t o = { .a = 10, 0, "bob" };
```

Good example

```
int tab[10] = { 0, 1, 2, 3, 5, 6, 0, 0, 0, 0 };
struct type_t o = { 10, 0, "bob" };
struct type_t p = { .a = 10, .b = 0, .c ="bob" };
```

An initialisation of structured variables often used and accepted is:

```
int tab[N] = {0};
une_structure st = {0};
```

This initialisation ensures that *all* elements/fields of the structured variable are initialised to zero.

Warning

However, the semantics of this notation should not be misunderstood:

```
int tab[N] = {1}; /* does not mean that all the elements are 1, but that all are at
     zero and only the first element is 1 */
```

This is because, in case of an incomplete initialisation of a structured variable (*i.e.* if not all fields/elements are explicitly initialised), then the unlisted fields/elements are initialised at 0 by default. Please note that this initialisation does not extend to the padding space.

In addition, C99 introduced the possibility of initialising (a) given element(s) of an array, which adds yet another possible source of error and confusion or even multiple initialisations of the same elements with potentially different values.

RULE 72

RULE — Structured variables must not be initialised without specifying the initialisation value and each field/element of the structured variable must be initialised

Non-scalar variables must be initialised explicitly: each element must be initialised with a clear identification, without superfluous initialisation values. Alternatively, the {0} initializer can be used in the declaration. Finally, arrays must have their size explicitly set when initializing them.

Bad example

The initialisations are not precise in the following example: there are non-explicit initialisations of the elements of the structured variables and superfluous initialisation values.

```
int32_t y[5] = {1, 2, 3}; /* the initialisation is misleading here - in reality,
    the last two elements are initialised at zero */

int32_t z[2] = {1, 2, 3}; /* as above - in reality the value 3 is ignored */

int16_t vv[5] = { [0] = -2, [1] = -9, [3] = -8, [2] = 18 }; /* source of error, the
     indexes 2 and 3 are not in increasing order and 4 is forgotten */

struct person {
  unsigned char name[20];
  uint16_t roll;
  float marks;
  int grades[10];
};

struct person p1 = {" ", 0}; /* obscure */
struct person p2 = {"toto",67,78.3,{0},12}; /* everything is correctly initialised,
     including
all the elements of the array grades which are set to 0, but 12 is ignored */
```

Good example

Initialisations are now explicit and encompass all elements of the structured variables without superfluous initialisation values.

```
int32_t y[5] = { 1, 2, 3, 4, 5 }; /* full initialisation */

int32_t z[2] = { 1, 2 }; /* no superfluous elements */

int32_t w[3] = { 0 };  /* accepted notation to initialise all elements with the
    value 0 */

int16_t vv[5] = { [0] = -2, [1] = -9, [2] = 18, [3] = -8, [4] = 33 }; /* ok */

struct person {
  unsigned char name[20];
  uint16_t roll;
  float marks;
  int grades[10];
};

struct person p1 = { .name = "titi", .roll = 12, .marks = 10.0f, .note = {0}};
/* all elements are explicitly initialised */
```

```
struct person p2  = { .name="toto", .roll=67, .marks=78.3, .grades={0}}; /* another
     example of recognised initialisation, this time without superfluous elements
    */
```

6.11.1 References

[Misra2012] Rule 9.2 The initializer for an aggregate or union shall be enclosed in braces.
[Misra2012] Rule 9.3 Arrays shall not be partially initialized.
[Misra2012] Rule 9.4 An element of an object shall not be initialized more than once.
[Misra2012] Rule 9.5 Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.
[Cert] Rec. ARR02-C Explicitly specify array bounds, even if implicitly defined by an array initializer.
[Cwe] CWE-665 Incorrect or incomplete initialization.

6.12 Mandatory use of declarations

When identifiers are declared but are not used afterwards, it may mean that the developer made a mistake when writing the code and that one element was used instead of another or that its use was removed from the program.

Information

GCC and CLANG compilation options such as `-Wunused-variable` and `-Wunused-parameter` make it possible to detect this kind of patterns.

RECO
73

RECOMMENDATION — Every declaration must be used

All declared identifiers must be used, whether they are variables, functions, labels, function parameters or anything.

Warning

When developing a library, not all declared identifiers are necessarily used : functions and variables exported by the library may obviously not be used by the library itself.

Bad example

In the following code, variables declared but not used must be deleted.

```
uint32_t init_list(list_t** pp_list) {
    list_t* p_list = NULL;
    list_element_t*  p_element = NULL;
    uint32_t ui32_list_len = 0;

    if (NULL == pp_list) {
        return 0;
    }

    (*pp_list) = (list_t*)malloc(sizeof(list_t));
    if (NULL == (*pp_list)) {
        return 0;
    }

    (*pp_list)->p_head = NULL;
    (*pp_list)->p_tail = NULL;

    return 1;
}
```

Good example

In the following example, all declared variables are used.

```
uint32_t init_list(list_t** pp_list) {
    if (NULL == pp_list) {
        return 0;
    }

    (*pp_list) = (list_t*)malloc(sizeof(list_t));

    if (NULL == (*pp_list)) {
        return 0;
    }

    (*pp_list)->p_head = NULL;
    (*pp_list)->p_tail = NULL;

    return 1;
}
```

6.12.1 References

[Misra2012] Rule 2.2 There shall be no dead code.
[Misra2012] Rule 2.3 A project should not contain unused type declarations.
[Misra2012] Rule 2.4 A project should not contain unused tag declarations.
[Misra2012] Rule 2.5 A project should not contain unused macro declarations.
[Misra2012] Rule 2.6 A project should not contain unused label declarations.
[Misra2012] Rule 2.7 There should be no unused parameters in functions.
[Cert] Rec. MSC07-C Detect and remove dead code.
[Cert] Rec. MSC13-C Detect and remove unused values.
[Cert] Rec. MSC12-C Detect and remove code that has no effect or is never executed.

6.13 Naming of variables for sensitive data

It is imperative to use separate variables used to store sensitive and non-sensitive data. In the absence of a well-defined naming convention, the developer risks using variables to successively store sensitive and non-sensitive data.

RULE 74

RULE — Use separate variables for sensitive data and non-sensitive data

Separate variables should also be used for unencrypted sensitive data and sensitive data protected in confidentiality and/or integrity.

RULE 75

RULE — Use different variables for sensitive data that are protected in confidentiality and/or integrity than the ones used for unprotected sensitive data

These rules are more of a principle of secure coding to avoid handling non-sensitive, encrypted sensitive and unencrypted sensitive data in the same variable.

It goes without saying that hard-coding any sensitive information of any kind (password, login, encryption key, *etc.*) is forbidden.

RULE 76

RULE — Never hard-code sensitive data

Bad example

The code below does not use a naming convention.

```
   #define KEY_SIZE 32U
   #define BUFFER_SIZE 512U

    size_t   key_len = 0;
    size_t   clear_data1_len = 0;
    size_t   encrypted_data2_len = 0;
    uint8_t  key[KEY_SIZE];
    uint8_t  data1[BUFFER_SIZE];
    uint8_t  data2[BUFFER_SIZE];
    uint32_t error_code = 0;
    error_code = cipher_data(clear_data, clear_data_len, key, key_len,
    encrypted_data, encrypted_data_len);
```

Good example

In the following example, a naming convention is used so that the same variables are not used for encrypted or unencrypted sensitive data.

```
   #define KEY_SIZE 32U
   #define BUFFER_SIZE 512U

/* conventions :
 suffix s for sensitive data variables‘’
 clear prefix for unencrypted data‘’
 encrypted prefix for encrypted data */

 size_t   encrypted_key_len_s = 0;
 size_t   clear_data_len_s = 0;
 size_t   encrypted_data_len_s = 0;
 uint8_t  encrypted_key_s[KEY_SIZE];
 uint8_t  clear_data_s[BUFFER_SIZE];
 uint8_t  encrypted_data_s[BUFFER_SIZE];
 uint32_t encrypted_error_code_s = 0;

 encryptederrorCode = cipher_data(clear_data_s, clear_data_len_s,
 encrypted_key_s, encrypted_key_len_s, encrypted_data_s, encrypted_data_len_s);
```

6.13.1 References

[Misra2012] Dir. 4.5 Identifiers in the same name space with overlapping visibility should be typographically unambiguous.
[Cert] Rec. DCL02-C Use visually distinct identifiers.
[Cert] Rule MSC41-C Never hard code sensitive information.
[Cwe] CWE-259 Use of Hard-Coded Password.
[Cwe] CWE-798 Use of Hard-Coded Credentials.

7
Types and type conversions

7.1 Explicit size for integers

Warning

The C standard does not define an explicit size for each type of integer. In particular, for the `int` type, depending on the architecture, it can be on 16, 32 or 64 bits.

Therefore, use of the `int` type is risky since it is necessary to be sure of the associated size and possible values to avoid any overflow or unexpected behaviour such as a value wrap (for unsigned integers).

It is therefore best to avoid using this type unless the developer is certain that the associated value range is contained within its range (e.g. in loop counters).

The type name must include its size on the target machine explicitly, like those defined in the header file `stdint.h`, available in C99. Its use is to be preferred to the generic `int` type. In C90, equivalent types must be defined and used. The redefinition of integer types is possible but this redefinition must be explicit on both the associated size and sign.

RECO 77

RECOMMENDATION — Only integer types with an explicit size and sign should be used

Furthermore, the plain `char` type should not be used for numeric values as its sign is not specified by the C standard and is implementation-defined. This type must be restricted to character handling.

RULE 78

RULE — Only `signed char` and `unsigned char` types must be used to handle numeric values

Bad example

```
#define MAXUINT16 65535U
int value;
char c = 35; /* sign is not specified */
```

```
if (value >= MAXUINT16)
{
  /* depending on the architecture */
}
```

Good example

```
#include <stdint.h> /* if C99 */
#define MAXUINT16 65535U
unsigned char c = 35U; /* sign is explicit, for numeric value manipulation */
...
uint32_t  value; /* if C99 */

typedef unsigned char uint8_t; /* type definition in C90 */
if (value >= MAXUINT16)
{
  /* ... */
}
```

7.1.1 References

[Misra2012] Rule 10.1 Operands shall not be of an inappropriate essential type.
[Misra2012] Rule 10.3 The value of an expression should not be assigned to an object with a narrower essential type or of a different essential type category.
[Misra2012] Rule 10.4 Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.
[Misra2012] Rule 8.1 Types shall be explicitly specified.
[Misra2012] Directive 4.6 typedef that indicate size and signedness should be used in place of the basic numerical types.
[Cert] Rec. INT00-C Understand the data model used by your implementation(s).
[Cert] Rec. INT07-C Use only explicitly signed or unsigned char type for numeric values.
[Cert] Rule INT35-C Use correct integer precisions.
[Cert] Rec STR00-C Represent characters using an appropriate type.
[Cwe] CWE-682 Incorrect calculation.

7.2 Type alias

The typedef operator allows the redefinition of a type that has itself been redefined using typedef. It is then difficult to follow the actual type checking of a variable. There is a significant risk of confusion and duplication of type definitions. The types defined by the language or by external libraries should therefore not be redefined more than once. If this is done intentionally by the developer for *strong type checking*, it should be commented on and explained, but also controlled to avoid any type confusion.

RECO
79

RECOMMENDATION — Do not redefine type aliases

Bad example

The following example (C90) shows multiple aliases of the same type.

```
typedef unsigned short uint16_t; /* definition of type uint16_t */
typedef unsigned short uint16_type; /* type uint16_type is an alias of type
    uint16_t */
typedef uint16_t unsigned_short; /* type unsigned_short is a redefinition of type
    uint16_t */
```

Good example

In the following example (in C90, *i.e.* before the introduction of `stdint.h`), only one type is correctly defined from the definition of a standard type of the language.

```
typedef unsigned short uint16_t; /* definition of type uint16_t */
```

7.2.1 References

[Cert] Rec. PRE03-C Prefer typedefs to defines for encoding non-pointer types.

7.3 Type conversions

C compilers perform implicit conversions from one type to another. However, these type promotions and implicit conversions can lead to errors (loss of information, miscalculations). In addition, the absence of explicit conversions does not facilitate proofreading of the code. It is therefore necessary to add the type conversion operator systematically and not to mix signed and unsigned types in the same arithmetic operation (operators: +, -, *, /, %, ~, >, <, >=, <=, <<, >>, *etc.*).

Multiple implicit conversions are made, whether in C90 or C99.

On the one hand, *integer promotion* is performed on integer values whose type is smaller than the `int` type and when these integer values are subjected to an operation (binary operators, unary operators, shifts, *etc.*). These integer values are then automatically and systematically converted to `int` or `unsigned int`.

On the other hand, *type balancing* corresponds to the usual conversion to a common type when operands are of different types. Finally, the last implicit conversion corresponds to the assignment of a value in a different type.

Information

Details of integer promotion can be found in sections 6.2.1.1. and 6.3.1.1. of standards [AnsiC90] and [AnsiC99] respectively. For type balancing, the relevant sections are 6.2.1.5 and 6.3.1.8 of standards [AnsiC90] and [AnsiC99] respectively.

RULE 80

RULE – Detailed and precise understanding of the conversion rules

The developer needs to know and understand all the implicit conversion rules for integer types.

The developer must make explicit the conversions implicit in the code to avoid any errors. The classic case that is often a source of errors is an implicit conversion between signed and unsigned types.

RULE 81

RULE – Explicit conversions between signed and unsigned types

Prohibit implicit type conversions. Use explicit conversions, particularly between signed and unsigned types.

Bad example

```
signed int v1 = -1;
unsigned int v2 = 1;
if (v1 < v2)
{
  /* v1 converted to unsigned int and value -1 becomes UINT_MAX, therefore the if
      condition is always false */
}
```

Good example

```
signed int v1 = -1;
unsigned int v2 = 1;
if (v1 < (signed int)v2)
{
  /* v2 is explicitly converted to a signed integer - the condition is true */
}
```

Again for the same reasons, no implicit conversion should be made between an integer type and a floating type or from an integer type to a smaller integer type.

Bad example

In the following lines, conversions are implicit.

```
uint32_t u32;
int32_t s32;
uint16_t u16;
double dbl;
uint8_t idx;

s32 = 42;
u32 = s32; /* implicit conversion */
u16 = u32 + 2 * s32; /* implicit conversion to a smaller type */
dbl = u32 / u16; /* the result is 0 (integer division) */
s32 = dbl; /* implicit conversion float -> integer */

/* the following loop is infinite: idx being unsigned, idx >= 0 is always true
    since an unsigned value cannot be negative */
for(idx = 27; idx >= 0; idx--) {
  ...
}
```

Tolerated example

The following example shows a code in which the type conversions are explicit. This is called a tolerated example because other cleaner solutions such as incrementing the loop index exist.

```c
uint32_t u32;
int32_t s32;
uint16_t u16;
double dbl;
uint8_t idx;

s32 = 42;
u32 = (uint32_t)s32;
u16 = (uint16_t)((int32_t)u32 + 2 * s32);
dbl = (double)u16 / (double)u32;

/* the signed integer cast is used to avoid an infinite loop */
for(idx = 27; (int8_t)idx >= 0; idx--) {
  ...
}
```

Good example

```c
int32_t s32;
uint16_t u16;
double dbl;
uint8_t idx;

s32 = 42;
u32 = (uint32_t)s32;
u16 = (uint16_t)((int32_t)u32 + 2 * s32);
dbl = (double)u16 / (double)u32;

/* change of the loop */
idx=27;
while (idx>0)
{
  ...
}
```

Information

`-Wconversion` and `-Wsign-conversion` warnings provided by GCC and CLANG can help to detect such implicit conversions.

7.3.1 References

[Misra2012] Rule 10.1 Operands shall not be of an inappropriate essential type.
[Misra2012] Rule 10.3 The value of an expression should not be assigned to an object with a narrower essential type or of a different essential type category.
[Misra2012] Rule 10.4 Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.
[Misra2012] Rule 10.5 The value of an expression should not be cast to an inappropriate essential type.
[Misra2012] Rule 10.6 The value of a composite expression shall not be assigned to an object with wider essential type.

[Misra2012] Rule 10.7 If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type.
[Misra2012] Rule 10.8 The value of a composite expression shall not be cast to a different essential type category or a wider essential type.
[Cert] Rec. INT02-C Understand integer conversion rules.
[Cert] Rule INT30-C Ensure that unsigned integer operation do not wrap.
[Cert] Rule INT31-C Ensure that integer conversions do not result in lost or misinterpreted data.
[Cert] Rule INT32-C Ensure that operations on signed integers do not result in overflow.
[Cert] Rec. INT18-C Evaluate integer expressions in a larger size before comparing or assigning to that size.
[Cert] Rec. EXP14-C Beware of integer promotion when performing bitwise operations on integer types smaller than int.
[Cwe] CWE-190 Integer overflow or wraparound.
[Cwe] CWE-192 Integer coercion error.
[Cwe] CWE-197 Numeric Truncation Error.
[Cwe] CWE-681 Incorrect conversion between numerical types.
[Cwe] CWE-704 Incorrect Type Conversion or Cast.
[IsoSecu] Conversion of signed characters to wider integer types before a check for EOF [signconv].
[IsoSecu] Overflowing signed integers [intoflow].

7.4 Type conversion of pointers to structured variables of different types

Type conversion from or to structured variables via pointers can result in overflows when the target type is larger than the memory area pointed to. Indeed, a type conversion from one structure to a larger structure, for example, gives undesirable access to areas of memory outside the initial structure.

In addition, type conversion from/to structured variables makes proofreading the code more complex.

RECO 82

RECOMMENDATION — Do not use pointer type conversion on types structured differently

Bad example

In the following code, a structure type conversion will result in an overflow.

```
#define TAB_SIZE 16U

typedef struct {
  int32_t magic;
} s_a;

typedef struct {
  int32_t magic;
  int16_t s;
  uint8_t x[TAB_SIZE];
} s_b;

void foo(s_a* structa) {
  s_b* p = (s_b*)structa; /* type conversion to be excluded */

  p->magic = 0xBAADCAFE;
  p->s = 0xDEAD; /* overflow outside of the structure s_a */
  p->x[0] = 4; /* and risk of overwritten data (buffer overflow) */
}
```

Good example

In the following code, the structure type conversion is no longer performed.

```
#define TAB_SIZE 16U

typedef struct {
  int32_t magic;
} s_a;

typedef struct {
  s_a h;
  int16_t s;
  uint8_t x[TAB_SIZE];
} s_b;

void foo(s_b* structb) {
  structb->h.magic = 0xCAFEBABE;
  structb->s = 0xBEEF;
  structb->x[0] = 4;
}
```

7.4.1 References

[Misra2012] Rule 11.2 Conversions shall not be performed between a pointer to an incomplete type of any other type.
[Misra2012] Rule 11.3 A cast shall not be performed between a pointer to object type and a pointer to a different object type.
[Misra2012] Rule 11.8 A cast shall not remove a const or volatile qualification from the type pointed to by a pointer.
[Cert] Rule EXP36-C Do not cast pointer into more strictly aligned pointer types.
[Cwe] CWE-704 Incorrect type conversion or cast.
[IsoSecu] Converting pointer values to more strictly aligned pointer types [alignconv].

8
Pointers and arrays

In this section we are referring only to one-dimensional arrays, but as any multi-dimensional array can also be represented via a one-dimensional array, all the rules and recommendations therefore also apply to multi-dimensional arrays.

8.1 Standardised access to the elements of an array

Confusion between arrays and pointers is common, and it is often assumed that an array behaves as a constant pointer to its first element. This statement is a shortcut that proves to be false in general.

Therefore, for the following code:

```
int32_t tab1[6];
int32_t * tab2 = malloc(6 * sizeof(int32_t));
int32_t * tab3 = tab1;
int32_t * tab4 = tab2;

printf("tab1: %p, %p, %p\n", tab1, &tab1[0], &tab1);
printf("tab2: %p, %p, %p\n", tab2, &tab2[0], &tab2);
printf("tab3: %p, %p, %p\n", tab3, &tab3[0], &tab3);
printf("tab4: %p, %p, %p\n", tab4, &tab4[0], &tab4);
```

the result obtained is as follows:

```
tab1: 1559248928, 1559248928, 1559248928 /* tab1=&tab1[0]=&tab1 all represent the address of
                                            the first element in the array */
tab2: 911295072, 911295072, 1559248904   /* &tab2 is the address of the pointer returned by
                                            malloc, pointing to the array, and tab2 (or &tab2[0])
                                            is the address of the first element of the array
                                            tab2 */
tab3: 1559248928, 1559248928, 1559248912 /* similar case to tab2 */
tab4: 911295072, 911295072, 1559248920   /* similar case to tab2 */
```

The nuances between arrays and pointers are numerous and we can only draw the reader's attention to this point and urge them to tread carefully.

Another example of a confusing code is:

```
int *var[N];
int (*var2)[N];
```

The first line involves declaring `N` `int` type pointers in memory, *i.e.* an array of `N` `int` type pointers. The second line declares a pointer to an array of `N` `int` type elements in memory.

The standard clarifies this point by explaining that any array type *expression* is converted to a pointer type expression pointing to the first element of the array and is not an *lvalue* except when the initial expression is used as an operand of the operators sizeof, _Alignof or & or if the expression is a literal string used to initialise an array.

Expression

An expression is not an object in memory but a piece of source code such as a+b or &a, for example.

Lvalue

An *lvalue* (locator value) is an expression with a type, even incomplete, but different from void which is associated with an address in memory.

An array is not a modifiable *lvalue*, which means that it cannot be assigned, incremented or modified in general.

```c
int tab[N];
tab = 0; // error
tab--; // error
```

When the array expression is converted to a pointer type expression, this expression then produces a simple value and is no longer an *lvalue*.

Warning

For an array tab, the tab and &tab[0] notations represent the address of the first element of the array created in memory. The &tab notation, on the other hand, will vary. When an array is declared statically, the array address cannot change and there is no pointer creation as such on the array: the tab notation is similar to a label managed by the compiler containing the address of the array.

- Therefore, if the array is declared statically (case of tab1 in the previous example), &tab always represents the address of the first element of the array, *i.e.* the address of the array.
- On the other hand, if the array is declared dynamically, the tab notation represents the pointer containing the address of the array created in memory and therefore, &tab represents the address of the pointer to the array (case of tab2 in the example).

The C standard allows access to the iþelement of a tab array to be represented in a variety of ways, which can be a source of errors or confusion.

Warning

For a tab array, access to the iþelement can be written:

```c
*(tab+i); /* usual notation 1 */
tab[i]; /* usual notation 2 */
*(i+tab); /* interchangeable array and index ! */
i[tab]; /* interchangeable array and index ! */
```

These notations are all recognised by the standard and are therefore correct, but this can quickly reduce understanding of the code. Note that, even with demanding compilation options, neither GCC nor CLANG will issue alerts on these types of notations, which can be a source of errors.

In order to avoid any ambiguity and misunderstanding of the code and therefore potential errors, notations tolerated by the standard and intended to invert the index and the name of an array will not be used.

RULE 83

RULE — Access to the elements of an array will always be by designating as the first attribute the array and as the second attribute the index of the element concerned

Access to the iþelement of an array will always be written with the name of the array first followed by the index of the element to be reached.

Furthermore, again for reasons of transparency, the typical notation of the arrays using square brackets [] will be preferred.

RECO 84

RECOMMENDATION — Access to elements in an array should be using square brackets

In the case of an array type variable, the dedicated notation (via square brackets) must be used to avoid any ambiguity.

Bad example

```
for (i = 0; i< size_tab; i++) {
  *(i+tab) = i; /* the square brackets are not used and the index is in the first
      position */
  ...
}
```

Good example

```
for (i = 0; i < size_tab; i++) {
  tab[i] = i;
  ...
}
```

8.1.1 References

[Cert] ARR00-C Understand how arrays work.

8.2 Non-use of VLAs

The VLAs[9] introduced with C99 correspond to arrays whose size is not associated with a constant integer expression at compilation but with an integer variable. This therefore corresponds to implementing an object of variable size on the stack. If the size of the array is not strictly positive,

[9] Variable-Length Array

this corresponds to undefined behaviour of C. Moreover, for an excessive array size, the program may not behave as expected. Finally, if the size of said array can be controlled by the user, it is a vulnerability. For all of these reasons, VLAs should not be used.

Information

The `-Wvla` option provides alerts on the use of VLAs in the code.

RULE 85

RULE — Do not use VLAs

8.2.1 References

[Misra2012] Rule 18.8 Variable-length array types shall not be used.
[Cert] ARR32-C Ensure size arguments for VLA are in a valid range.
[Cert] MEM05-C Avoid large stack allocations.
[Cwe] CWE-758 Reliance on undefined, unspecified, or Implementation-defined behavior.
[IsoSecu] Tainted, potentially mutilated, or out-of-domain integer values are used in a restricted sink [taintsink].

8.3 Explicit array size

The size of the arrays must be explicit to avoid out of bounds access. This recommendation may seem redundant in relation to a previous rule imposing explicit declarations, but the focus here is on the size of the arrays.

RECO 86

RECOMMENDATION — Do not use an implicit size for arrays

In order to ensure that array accesses are valid, their size must be made explicit.

Bad example

In the example below, the size of the array is implicit on initialisation.

```
int32_t tab [] = { 1, 2, 3 }; /* array 3 elements, implicit size */
```

Good example

This time, the size of the arrays is explicitly specified.

```
int32_t tab[3] = { 1, 2, 3 }; /* array of 3 elements, explicit size, with
    initialisation */
int32_t tab2[2] = { 2, 3 };    /* array of 2 éléments, explicit size, with
    initialisation */
```

8.3.1 References

[Misra2012] Rule 8.11 When an array with external linkage is declared, its size should be explicitly specified.
[Misra2012] Rule 9.5 Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.
[Cert] Rule ARR30-C Do not form or use out-of-bounds pointers or array subscripts.
[Cert] Rec. ARR02-C Explicitly specify array bounds, even if implicitly defined by an initializer.
[Cwe] CWE-655 Incorrect or incomplete initialization.

8.4 Systematic check for array overflow

Accessing an array element outside the allocated size is a classic coding error. For each access to the element of an array, it must be checked whether the index used is strictly positive or null and strictly less than the number of elements allocated for that array.

RULE 87

RULE – Use unsigned integers for array sizes

RULE 88

RULE – Do not access an array element without checking the validity of the used index

The validity of the used array index must be checked systematically: an array index is valid if it is greater than or equal to zero and strictly less than the declared size of the array. In the case of a character array, the end of string character '\0' must be taken into account.

Bad example

```
i++;
tab[i] = i; /* no overflow check */
```

Good example

```
for (i = 0; i < size_tab; i++){ /* size_tab is the number of elements in the array
    */
  tab[i] = i;
  ...
}
```

8.4.1 References

[Cert] Rec. ARR02-C Explicitly specify array bounds, even if implicitly defined by an initializer.
[Cert] Rule ARR30-C Do not form or use out-of-bounds pointers or array subscripts.

[Cert] Rule STR31-C Guarantee that storage for strings has sufficient space for character data and the null terminator.
[IsoSecu] Tainted, potentially mutilated, or out-of-domain integer values are used in a restricted sink [taintsink].
[IsoSecu] Forming or using out-of-bound pointers or array subscripts [invptr].
[IsoSecu] Using a tainted value to write to an object using a formatted input or output function [taintformatio].
[IsoSecu] Tainted strings are passed to a string copying function [taintstrcpy].
[Cwe] CWE-119 Improper Restriction of Operations within the bounds of a Memory buffer.
[Cwe] CWE-120 Buffer Copy without Checking Sizeof Input (Classic Buffer Overflow).
[Cwe] CWE-123 Write-what-where Condition.
[Cwe] CWE-125 Out-of-bounds read.
[Cwe] CWE-129 Improper Validation of Array Index.
[Cwe] CWE-170 Improper Null termination.

8.5 Do not dereference NULL pointers

Dereferencing a *NULL* pointer leads to undefined behaviour. This may result in an abnormal termination of the program. Therefore, before dereferencing a pointer, ensure that it is not *NULL*.

RULE 89

RULE – A NULL pointer must not be dereferenced

Before dereferencing a pointer, the developer must ensure that it is not NULL.

Bad example

In the following function, the pointer passed as a parameter is used without being checked.

```
void function(const unsigned char *input)
{
  size_t size = strlen(input); /* the pointer may be NULL */
  ...
}
```

Good example

Handling of the error relating to the NULL pointer has been added.

```
void function(const unsigned char *input)
{
  if (NULL == input)
  {
    /* handling of the NULL pointer case */
  }
  else
  {
    size_t size = strlen(input);
    /* ... */
  }
}
```

8.5.1 References

[Cert] Rule EXP34-C Do not dereference null pointers.
[IsoSecu] Dereferencing an out-of-domain pointer [nullref].
[Cwe] CWE-476 NULL Pointer Dereference.
[AnsiC99] Section 6.5.3.2.
[AnsiC90] Section 6.3.3.3.

8.6 Assignment to NULL of deallocated pointers

Following the deallocation of the memory pointed to by a pointer, the pointer variable still stores its address. This is known as a *dangling pointer*.

Dangling pointer

A *dangling pointer* is a pointer that contains the memory address of an element that has been freed.

In case of bugs and incorrect use of the deallocated pointer, the memory may be corrupted. Once freed, the memory may (or may not) be reused by the system. The result of the use of the memory area (via the pointer) is then undefined and not necessarily visible, and may cause security problems (*use-after-free*). By assigning the pointer to NULL after deallocation, you can specify that the pointer no longer points to a valid memory area. And in case of an accidental use of the pointer, no memory area will be corrupted since the pointer no longer points to any valid memory area.

RULE 90

RULE — A pointer must be assigned to NULL after deallocation

A pointer must be systematically assigned to NULL following the deallocation of the memory it points to.

Bad example

In the code below, the pointer is not set to NULL following its deallocation.

```
list_t *p_list = NULL;
p_list = create_list();
...
if(p_list != NULL) {
   free_list(p_list);
}
/* setting of p_list to NULL is missing */
```

Good example

In the following example, the pointer is correctly set to NULL following the deallocation of the area being pointed to.

```
list_t *p_list = NULL;
p_list = create_list();
...
if(p_list != NULL) {
    free_list(p_list);
    p_list = NULL;
}
```

8.6.1 References

[Cert] Rule MEM30-C Do not access freed memory.
[Cert] Rec MEM01-C Store a new value in pointers immediately after free().
[Misra2012] Rule 18.6. The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist.
[Cwe] CWE-415. Double free.
[Cwe] CWE-416. Use after free.
[Cwe] CWE-672 Operation on a resource after expiration or release.
[IsoSecu] Accessing freed memory [accfree].
[IsoSecu] Freeing memory multiple times [dbfree].

8.7 Use of the `restrict` type qualifier

The `restrict` qualifier, introduced in C99, is a means of indicating to the compiler that the area being pointed to cannot be accessed without passing via the pointer marked `restrict`. A `restrict`-qualified pointer therefore implies that the object it points to is reached directly or indirectly only via this pointer. This requires that there are no other aliases on the object you are pointing to.

Alias

Two aliases are two variables or access paths to the same memory area.

Warning

The `restrict` qualifier is a declaration of the developer's intention to associate a single pointer with a memory area, not an actual fact. In practice, nothing prevents the code from reaching the same area via a different pointer.

The behaviour becomes undefined if objects pointed to by `restrict` pointers have common memory addresses. In addition, it is necessary to check for the absence of common memory addresses on each function call with `restrict` type parameters, but also during the execution of said functions.

Warning

Several functions in the standard library have restrict type parameters since C99 (`memcpy`, `strcat`, `strcpy`, *etc.*).

It is very easy to introduce undefined behaviour through the use of `restrict`, as it must be ensured that none of the pointers concerned share a memory area, while taking into account function calls from the standard library which also have restrict type parameters since C99. The use of the restrict qualifier directly by the user is therefore to be prohibited.

RULE
91

RULE — Do not use the restrict pointer qualifier

The restrict qualifier must not be used directly by the developer. Only indirect use, *i.e.* via the standard library function call, is tolerated, but the developer must ensure that no undefined behaviour will result from the use of such functions.

Information

The -Wrestrict GCC option makes it possible to alert to the misuse of restrict pointers.

Bad example

In the following example, restrict type pointers share memory areas and therefore cause undefined behaviour.

```c
uint16_t * restrict ptdeb;
uint16_t * restrict ptfin;
uint16_t tab[12];
unsigned char * pt1;
unsigned char * pt2;
unsigned char c_str[] = "blabla";
...
ptdeb = &tab[0];
ptfin = &tab[11];
ptdeb = ptfin; /* undefined behaviour */
...
pt1 = pt2 + 2;
memcpy(pt2, pt1, 3); /* undefined behaviour - restrict type memcpy parameters */
```

Good example

In the following example, the restrict qualifiers have been deleted and there is no longer any undefined behaviour.

```c
uint16_t *  ptdeb; /* deletion of restrict qualifier */
uint16_t *  ptfin; /* deletion of restrict qualifier */
uint16_t tab[12];
unsigned char * pt1;
unsigned char * pt2;
unsigned char c_str[] = "blabla";
...
ptdeb = &tab[0];
ptfin = &tab[11];
ptdeb = ptfin; /* ok */
...
pt1 = pt2 + 2;
memmove(pt2, pt1, 3); /* change of function */
```

8.7.1 References

[Misra2012] Rule 8.14 The restrict type shall not be used.
[IsoSecu] Passing pointers into the same objects as arguments to different restrict-qualified parameters [restrict].

[Cert] Rule EXP43-C Avoid undefined behavior when using restrict-qualified pointers.

8.8 Limit on the number of pointer indirections

When a pointer has more than two levels of indirection (for example: pointer of pointer of pointer `int32_t ***pppInt32`), it becomes difficult to understand the developer's intentions and the behaviour of the code.

RECO 92

RECOMMENDATION — The number of levels of pointer indirection should be limited to two

The number of levels of indirection for a pointer should not exceed two.

Bad example

The following code shows excessive levels of indirection.

```
void function(int8_t ***arr_pt) /* 3 levels */
{
  int8_t ***pt;
...
}
```

Good example

In the following example, temporary pointers are introduced to facilitate access to the data and limit the number of nesting levels.

```
typedef int8_t *int8ptr_t;
void function(int8ptr_t **arr_pt) /* reduction to two levels */
{
  int8_t *pt_temp; /* temporary pointer */
  int8ptr_t **pt;
...
}
```

8.8.1 References

[Misra2012] Rule 18.5 Declarations should contain no more than two levels of pointer nesting.

8.9 Give preference to the use of the indirection operator ->

Two writing methods are possible in the C language to reach a structure field via a pointer: the indirection operator `ptr->field` and dereferencing `(*ptr).field`. However, the second method is often a source of errors and comprehension problems. It is therefore best to avoid using dereferencing `(*ptr).field` to reach a field of a structure via a pointer.

RECO 93

RECOMMENDATION — Give preference to the use of the indirection operator ->

The indirection operator -> should be used to reach the fields of a structure via a pointer.

Bad example

In the following example, the access should be rewritten with the indirection operator.

```
(*list.p_head).pNext = NULL;
```

Good example

In the following code, the indirection operator is correctly used.

```
list.p_head->pNext = NULL;
```

8.10 Pointer arithmetic

The C language allows direct access to the memory using pointers. Arithmetic operations can be applied to the value of a pointer either to increment or decrement it.

Pointer arithmetic

Pointer arithmetic is the use of pointer values as integer values in an elementary arithmetic operation (subtraction and addition).

Pointer arithmetic is very often used in the case of a pointer to an array element to navigate between the different elements of the array. Apart from this case, arithmetic on memory addresses is very risky.

RULE 94

RULE — Only incrementing or decrementing array pointers is authorised

Incrementing or decrementing pointers should only be used on pointers representing an array or an element of an array.

Arithmetic on `void*` type pointers is therefore prohibited. No memory size is in fact associated with the `void*` type, which causes undefined behaviour, in addition to the violation of the previous rule.

RULE 95

RULE — No arithmetic on `void*` pointers is authorised

The use of any arithmetic on `void*` type pointers must be prohibited.

Even in the case of pointer arithmetic on elements in an array, a special care must be taken to ensure that the arithmetic will not cause dereferencing outside of the array.

RECO
96

RECOMMENDATION — Controlled pointer arithmetic on arrays

Arithmetic on pointers representing an array or an element of an array must be carried out ensuring that the resulting pointer will still point to an element of the same array.

As a result, subtractions or comparisons between pointers will only be relevant for pointers on the same array.

RULE
97

RULE — Subtraction and comparison between pointers in the same array only

Only subtractions and comparisons of pointers on the same array are authorised.

Finally, the assignment of a fixed address to a pointer is strongly discouraged.

RECO
98

RECOMMENDATION — A fixed address should not be assigned directly to a pointer

Bad example

```
#include <stddef.h>
#include <stdint.h>
void function(int8_t * ptr_param)
{
  int8_t tab1[10];
  int8_t tab2[100];

  int8_t *pt1=&tab1[0];
  int8_t *pt2=&tab2[0];

  ptr_param ++; /* it is not known whether ptr_param points to an array ... */
  pt1++; /* pt1 points to the next element of tab1 */
  ptr_param = pt1 + pt2; /* illegal memory access */

  if (pt2 >= 15) /* not relevant */
  {
    /* ... */
  }

  uint8_t nb_elem = pt2 - pt1; /* the two pointers are not on the same array and
      the type is not adapted */
 ...
}
```

Good example

```
#include <stddef.h>
#include <stdint.h>
void function(int8_t * ptr_param)
{
  int8_t tab1[10];
  int8_t tab2[100];

  int8_t *pt1 = &tab1[0];
  int8_t *pt2 = &tab2[0];
  pt1++; /* pt1 points to the next element of tab1 */
  pt2 = pt2 + 3; /* pt2 points to tab2[3] */
  pt1 = pt1 + 8; /* pt1 points to the last elements of tab1 */

  if (pt1 >= tab1 ) /* same array ok */
  {
    /* ... */
  }

  ptrdiff_t nb_elem = pt2 - tab2; /* both pointers are on the same array and
      dedicated type used (taken from stddef.h) */
  ...
}
```

8.10.1 References

[Misra2012] Rule 18.1 A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand.
[Misra2012] Rule 18.2 Substraction between pointer shall only be applied to pointers that address elements of the same array.
[Misra2012] Rule 18.3 The relational operators shall not be applied to objects of pointer type exception where they point into a same object.
[Misra2012] Rule 18.4 The +, -, += and -= operators shall not be applied to an expression of pointer type.
[Cert] Rule ARR36-C Do not substract or compare two pointers that do not refer to the same array.
[Cert] Rule ARR37-C Do not add or subtract an integer to a pointer to a non-array object.
[Cert] Rule ARR39-C Do not add or substract a scaled integer to a pointer.
[Cert] Rec. EXP08-C Ensure pointer arithmetic is used correctly.
[IsoSecu] Subtracting or comparing two pointers that do not refer to the same array [ptrobj].
[IsoSecu] Forming or using out-of-bounds pointers or array subscripts [invptr].
[Cwe] CWE-469 Use of pointer substraction to determine size.
[Cwe] CWE-468 Incorrect pointer scaling.
[Cwe] CWE-466 Return of pointer value outside of expected range.
[Cwe] CWE-587 Assignment of a fixed address to a pointer.
[AnsiC99] Sections 6.2.5, 6.3.2.3, 6.5.2.1, 6.5.6.
[AnsiC90] Sections 6.1.2.5, 6.2.2.3, 6.3.6, 6.3.8.

9
Structures and unions

9.1 Declaration of structures

In order to model an entity within a program, it is often necessary to want to associate several scalar data items (integers, characters, *etc.*). The definition of independent variables to represent this entity makes it difficult to understand the code and tedious to pass parameters to a function. A structure must be used to group together data representing the same entity. And as many structures must be defined as there are entities to model. You should not use a single structure and group data relating to different entities in it.

RULE 99

RULE — A structure must be used to group data representing the same entity

Linked data must be grouped within a structure.

Information

This rule is not linked to an immediate security risk, but is a common-sense rule to be applied to all developments.

Bad example

In the following example, the lack of structure results in function prototypes that are difficult to understand.

```
void rectangle (float x0, float y0, float x1, float y1, float x2,
float y2, float x3, float y3);
void pyramide(float* coords);  /* coords is an array */
```

Good example

The following example correctly uses independent structures to represent different geometric shapes.

```
typedef struct point_s {
    float x;
    float y;
} Point_t;

typedef struct rectangle_s {
    point_t xy0;
    point_t xy1;
    point_t xy2;
    point_t xy3;
} rectangle_t;
```

```
typedef struct pyramid_s {
    rectangle_t base;
    point_t     top;
} pyramid_t;

void rectangle (rectangle_t* rect);

void pyramid(pyramid_t* pyra);
```

9.2 Size of a structure

The size of a structure should not be assumed to be equal to the sum of the size of its elements. This is due to structure padding. It corresponds to a rearrangement of the fields in memory in order to properly align the structure (it is referred to as padding fields). For this reason, the size of a structure should not be calculated by adding up the size of its fields, as this does not take into account the size of the padding fields.

RULE 100

RULE — Do not calculate the size of a structure as the sum of the size of its fields

Because of the padding, the size of a structure should not be assumed to be the sum of the size of its fields.

Bad example

```
#define SIZE_TABL 100
...
typedef struct{
  int tabl[SIZE_TABL];
  size_t size;
} my_struct;
...
size_t sizestruct= sizeof(my_struct.tabl)+sizeof(my_struct.size);
/* assumes that the size of the structure is the sum of the size of the elements */
...
```

Good example

```
#define SIZE_TABL 100
...
typedef struct {
  int tabl[SIZE_TABL];
  size_t size;
} my_struct;
...
size_t sizestruct = sizeof(my_struct); /* good size */
...
```

Warning

The use of non-standard attributes such as `packed` is not considered in this guide.

9.2.1 References

[Cert] EXP42-C Do not compare padding data.
[Cert] EXP03-C Do not assume the size of a structure is the sum of the sizes of its members.
[Cert] Rule DCL39-C Avoid information leakage when passing a structure across a trust boundary.

9.3 bit-field

In C, you can specify the size (in bits) of elements of a structure or union, in particular to use memory more efficiently. Precautions must be taken when using *bit-field*. On the one hand, an `int` type bit-field will not necessarily be signed. In fact, an `int` variable is indeed signed by default except in the case of bit-field, where the sign becomes dependent on the compiler implementation.

RULE 101

RULE — All bit-fields must be explicitly declared as unsigned

Furthermore, the internal representation of structures with bit-fields is also implementation-dependent, so that no assumption should ever be made about this representation.

RULE 102

RULE — Do not make assumptions about the internal representation of structures with bit-fields

Bad example

```
typedef struct structure {
  int ok: 1; /* bit-field of size 1 */
  int value: 7; /* bit-field for which the sign depends on the compiler used */
} struct_bitfield;
..
struct_bitfield s;
int *pt_s;
pt_s=(int *) &s;
s.ok=1;
..
if(s.ok==1) /* if compiled with gcc for example, by default the bit-fields are
signed, therefore being of size 1, s.ok equals 0 or -1 ! */
{
  pt_s++;   /* ? */
  *pt_s=100; /* ? */
}
```

Good example

```
typedef struct structure {
  unsigned int ok: 1; /* unsigned bit-field */
  unsigned int value: 7; /* unsigned bit-field */
} struct_bitfield;
..
```

```
struct_bitfield s;
s.ok=1;
..
if(s.ok==1) /* no longer compiler dependent */
{
   s.value=100;
}
```

9.3.1 References

[Cert] Rule EXP11-C Do not make assumptions regarding the layout of structures with bit-fields.
[Cert] Rec. EXP12-C Do not make assumptions about the type of a plain int bit-field when used in an expression.
[Misra2012] Dir. 1.1. Any implementation-defined behaviour on which the output of the program depends shall be documented and understood.

9.4 Use of FAMs

FAMs [10] were introduced with C99. This corresponds to declaring as the last member of a structure an array without dimensions and therefore of flexible size by nature. If the associated structure is not allocated (or copied) dynamically but on the stack, no space is allocated for this array and accessing it causes undefined behaviour.

Furthermore, this means accepting arrays of undefined size, which contradicts the rule in section 8.3. FAMs are therefore prohibited.

RULE 103

RULE – Do not use FAMs

9.4.1 References

[Misra2012] Rule 18.7 Flexible array member shall not be used.
[Cert] MEM33-C Allocate and copy structures containing a flexible array member dynamically.
[Cert] Rule DCL38-C Use the correct syntax when declaring a flexible array member.

9.5 Do not use unions

The C language, via the union mechanism, allows the same memory space to be used to store different data types. However, there is a risk of misinterpretation and misuse of the data. The use of the same space for several data types should therefore be avoided as far as possible.

[10] Flexible Array Member

RECO 104

RECOMMENDATION — Do not use unions

The use of the same memory space for different data types is not authorised.

The use of unions must be strictly limited to cases where the type is verified by other means and only if it is necessary (for network frame parsing for example), and this must be justified with a comment in the code.

9.5.1 References

[Misra2012] Rule 19.2 The union keyword should not be used.

10
Expressions

10.1 Integer expressions

Several basic precautions should be taken whenever expressions handle integers.

For signed integer operations, it must be ensured that there is no overflow of the size of the associated type and for unsigned integer operations, that there will be no value wrap.

RULE 105

RULE — Remove all possible value overflows for signed integers

RECO 106

RECOMMENDATION — Detect all possible value wraps for unsigned integers

Bad example

In the following function, no overflow is checked.

```
#include <stdint.h>
void f(uint8_t i, int8_t j)
{
  uint8_t ibis = i+ 2;
  int8_t j_bis = j +3;
 /* ... */
}
```

Good example

In the following function, overflows are checked.

```
#include <stdint.h>

void f(uint8_t i, int8_t j)
{
  uint8_t ibis;
  int8_t j_bis;
  if (i > (UINT8_MAX - 2))
  {
    /* error */
  }
  else
  {
    ibis=i+2;
  }
  if (j > (INT8_MAX - 3))
  {
    /* error */
  }
  else
  {
    jbis=j+3;
  }
  /* ... */
}
```

Similarly, all potential errors due to division by zero must be avoided.

RULE
107

RULE – Detect and remove any potential division by zero

This check must be systematic for any division or computation of remainder of a division.

Bad example

In the following function, no check on a possible division by zero is performed.

```
#include <stdint.h>
void func(int8_t i, int8_t j)
{
  int8_t result;
  result = i / j;
  ...
}
```

Good example

In the following function, there is a check on a possible division by zero.

```
#include <stdint.h>
void func(int8_t i, int8_t j)
{
  int8_t result;
  if (0 == j)
  {
    /* error */
  }
  else
  {
  result = i / j;
  }
  ...
}
```

10.1.1 References

[Cert] Rule INT30-C Ensure that unsigned integer operation do not wrap.
[Cert] Rule INT31-C Ensure that integer conversions do not result in lost or misinterpreted data.
[Cert] Rule INT32-C Ensure that operations on signed integers do not result in overflow.
[Cert] Rule INT33-C Ensure that division and remainder operations do not result in divide-by-zero errors.
[Cert] Rec. INT08-C Verify that all integer values are in range.

[Cert] Rec. INT10-C Do not assume a positive remainder when using % operator.
[Cert] Rec. INT18-C Evaluate integer expressions in a larger size before comparing or assigning to that size.
[Cert] Rec. INT16-C Do not make assumptions about representation of signed integers.
[Cwe] CWE-190 Integer overflow or wraparound.
[Cwe] CWE-682 Incorrect calculation.
[Cwe] CWE-369 Divide by Zero.
[IsoSecu] Integer division errors [diverr].

10.2 Readability of arithmetic operations

Understanding an arithmetic calculation can be complex if no effort has been made to make it legible. Moreover, depending on the writing method chosen for the calculation, it may prove ambiguous.

A complex expression will need to be simplified to aid understanding. If the complexity is relevant (optimisation, *etc.*), a comment should explain and accompany the expression. A fairly common example is to use a n-bit left shift for a multiplication by 2^n (or a right shift for a division). Thus, the following expression:

```
a << b;
```

can be used to perform the operation $a * 2^b$. Such expressions do not help with understanding the code. In addition, these shifts must comply with precise rules taking into account the number of bit shifts requested and the size of the type concerned (*cf.* section 10.7). It is recommended to use bit shifts only when the purpose is to handle the bits of a register, for example.

RECO 108

RECOMMENDATION — Arithmetic operations should be written in a way that assists with readability

Arithmetic operations that are as explicit (natural) as possible and follow the logic of the program must be used.

Bad example

In the following example, the arithmetic operations are not readable. Understanding of the operations is not immediate.

```
/* In the following calculation, we want to calculate a² + 4ac + b² */
uint64_t res;
uint32_t a, b, c;
res = a * a + ((a * c) << 2) + b * b; /* an explanation would be welcome */
/* a<<b is equalivaent to a* 2^b but here the bit shift is used for a
    multiplication */
...
/* mask computation */
uint32_t bitfield = 0xCAFEBABE;
uint32_t n, bitmask;
n = 4;
bitmask = 1;
for(n = n; n > 0; n--) {
  bitmask = 2 * bitmask;
} /* on the contrary here, the bit shift would have been more logical */
```

```
bitfield = bitfield & (~bitmask);
```

Good example

The following code performs calculations using simple arithmetic operations.

```
/* computation of a² + 4ac + b² */
uint64_t res;
uint32_t a;
uint32_t b;
uint32_t c;
res = (a * a) + (4 * (a * c)) + (b * b); /* plus clair */
...
/* mask computation */
uint32_t bitfield = 0xCAFEBABE;
uint32_t n = 4;
uint32_t bitmask
bitmask = 2 << n;  /* bits handling */
bitfield = bitfield & (~bitmask);
```

10.2.1 References

[Cert] Rec. INT14-C Avoid performing bitwise and arithmetic operations on the same data.

10.3 Use of parentheses to make explicit the order of the operators

The C language has many operators, with different levels of priority in terms of their associativity. However, the absence of parentheses in an expression makes it difficult to understand and proofread.

The systematic use of parentheses in the calculations makes it possible to clearly show and choose the priority of the operations and the order in which the calculation is performed.

Information

The C language operators and their priorities are presented in appendix D.

RULE 109

RULE — Explanation of the order of evaluation of calculations through the use of parentheses

To avoid any ambiguity in an expression, its subexpressions must be surrounded by parentheses to make the order of evaluation of a calculation more explicit.

10.3.1 References

[Cert] EXP10-C Rec. Do not depend on the order of evaluation of subexpressions or the order in which side effects take place.
[Cert] Rule EXP30-C Do not depend on the order of evaluation for side effects.
[Misra2012] Adv. 12.1 The precedence of operators within expressions should be made explicit.
[Misra2012] Rule 13.2 The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders.
[Misra2012] Rule 13.5 The right hand operand of a logical `&&` or `||` operator shall not contain persistant side effects.
[Cwe] CWE-783 Operator Precedence Logic Error.

10.4 No multiple comparison of variables without parentheses

It is common to want to check the value of a variable against a lower and an upper limit, and the shortcut of doing this in a single statement without parentheses is an error. Let us take the following expression as an example: `(0<=x<=n)`. The left side, *i.e.* `0<=x`, is evaluated first. The result of this evaluation (0 or 1) is then compared with the value to the right, which will always be checked for any value of n greater than or equal to 1. The statement `(0<=x<=n)` is therefore semantically equivalent to `((0<=x)<=n)` and not to `((0<=x)&&(x<=n))`.

Another classic error is the combined `if(a==b==c)` equality test whose objective is, a priori, to verify that the three variables are equal. In practice, as in the previous case, this test does not behave as the developer expects. Indeed, this conditional will only be true if the three variables are 1 or if `c` is 0 and `a` and `b` are different.

Boolean expression

The C language does not have a true boolean type in C90. The boolean type was introduced with C99. It has an associated library (`stdbool.h`). However, we will use the term boolean expression for expressions in the C language, even before C99, where the result of the evaluation corresponds to a truth value, as is typically the case for comparison expressions. A boolean expression corresponds to the false truth value for an evaluation returning the value 0. Any other value returned by a boolean expression (whether it is 1 or a negative, positive, integer or non-integer value) corresponds to the true truth value.

Boolean expressions containing at least two relational operators are prohibited without parentheses and must be broken down either into nested conditionals or into several relational expressions.

RECO 110

RECOMMENDATION — Avoid expressions of comparison or multiple equality

RULE 111

RULE — Always use parentheses in expressions of comparison or multiple equality

Boolean expressions of comparison or equality containing at least two relational operators are prohibited without parentheses.

Bad example

```
#define N 100
...
if (0 <= x <= N) {
  /* statement 1 ALWAYS executed */
} else {
  /* statement 2 NEVER executed */
}
...
if (30 < x < 40) {
  printf("pb"); /* ALWAYS executed */
}
...
```

Good example

```
#define N 100
...
if ((0 <= x) && (x <= N)) { /* case 1 : breakdown into 2 relational expressions */
  /* statement 1 */
} else {
  /* statement 2 */
}
 ...
if (30 < x) { /* case 2 : breakdown into 2 nested conditional statements */
  if (x<40) {
    printf("pb");
 ...
}}
```

10.4.1 References

[Misra2012] Rule 10.1 Req. Operands shall not be of an inappropriate essential type.
[Misra2012] Rule 12.1 The precedence of operators within expressions should be made explicit.
[Cert] EXP00-C Rec. Use parentheses for precedence of operation.
[Cert] EXP13-C Rec. Treat relational and equality operators as if they were nonassociative.
[Cwe] CWE-783 Operator Precedence Logic Error.

10.5 Parentheses around elements of a boolean expression

When writing boolean expressions, the absence of parentheses and the exclusive use of associativity makes it difficult to understand the code. The systematic use of parentheses avoids programming errors by making the order of evaluation of operations explicit.

RULE
112

RULE – Parentheses around the elements of a boolean expression

The different elements of a boolean expression must always be placed in parentheses, so that there is no ambiguity in the order of evaluation.

Bad example

In the following example, it is necessary to know the associativity between the operators and the priority between them in order to understand the order of evaluation.

```
if (x > 0 && y * z > length) {
    ...
}
u = z > 100 && 100 == x || x + y < z;
```

Good example

In the following code, the use of parentheses makes it possible to explicitly know the order of evaluation.

```
  if ((x > 0) && ((y * z) > length)) {
    ...
  }
  u = (z > 100) && ((100 == x) || ((x + y) < z));
```

10.5.1 References

[Cert] EXP00-C Rec. Use parentheses for precedence of operation.
[Misra2012] Rule 12.1 The precedence of operators within expressions should be made explicit.
[Cwe] CWE-783 Operator Precedence Logic Error.

10.6 Implicit comparison with 0 prohibited

In C90, the “true” value corresponds to any value other than 0 (whether that value is negative, positive, integer or non-integer) and the “false” value corresponds to the value 0. As a result, it is possible to write boolean expressions where a comparison with 0 is implicitly made. Implicit comparisons make understanding and maintenance of the code difficult. Boolean expressions must use an explicit comparison operator:

==, !=, <, >, <=, >=

RULE
113

RULE – Implicit comparison with 0 prohibited

All boolean expressions must use comparison operators. No implicit test with a value equal to 0 or different from 0 must be performed.

RECO 114

RECOMMENDATION — Using the bool type in C99

In C99, the `bool` (or `_Bool`) type must be used for variables with boolean values.

In C99, the use of the `bool`-typed variables directly in a conditional is accepted.

Bad example

The comparisons implicit in the following code should be removed in favour of explicit comparisons.

```
#define MAX 10
uint8 z;
..
while (x) {
  ...
}
if (x < y) {
  ...
} else {
  ...
}
if (!z) {  /* implicit comparison with 0 and z should be of the bool type */
  ...
}
if (ptr) {
  ...
}
for (x = MAX; x; x--) {
  ...
}
```

Good example

In the following example, no implicit comparison is performed and the dedicated header file is used.

```
#include <stdbool.h>

bool z; /* use of the bool type */
while (x > 0) {
  ...
}
if (x < y) {
  ...
} else {
  ...
}
if (FALSE== z) { /* constant on the left and explicit comparison of z; z being of
    the bool type, if (! z) can be used */
  ...
}
if (NULL != ptr) {
  ...
}
for (x = MAX; x > 0; x--) {
  ...
}
```

10.6.1 References

[Cert] Rec. EXP20-C Perform explicit tests to determine success, true and false, and equality.

10.7 Bitwise operators

Generally speaking, bitwise operators should only be used with expressions of unsigned types. This way, many undefined or implementation-defined behaviours are avoided.

RECO 115

RECOMMENDATION — Bitwise operators must be used with unsigned operands only

Furthermore, some bitwise operators such as `&`, `|` or `^` can easily be mistakenly used — especially the first two — instead of logical operators such as `&&`, `||` and `!`. To avoid this confusion, it is important to check that the bitwise operators used in boolean expressions are actually the desired operators. Bitwise operators have no reason to be applied to any operand of type boolean or similar.

RULE 116

RULE — No bitwise operator on an operand of type boolean or similar

Bad example

```
if ((var >= 0) & (var < 120)) { /* operator confusion */
 ...
if ((val && FLAG) != 0) /* operator confusion? */
 ...
}
```

Good example

```
 if ((var >= 0) && (var < 120)) { /* correction */
 ...
 if ((val & FLAG) != 0) /* correct use here of the bitwise operator
                           in a boolean expression */
 ...
}
```

10.7.1 References

[Cert] Rec. INT13-C Use Bitwise operators only on unsigned operands.
[Cert] Rec. EXP14-C Beware of integer promotion when performing bitwise operations on integer types smaller than int.
[Cert] Rule INT34-C Do not shift an expression by a negative number of bits or by greater or equal the number of bits that exist in the operand.

[Cwe] CWE-682 Incorrect calculation.
[Cert] EXP46-C Do not use a bitwise operator with a Boolean-like operand.
[Cwe] CWE-480 Use of incorrect operator.
[Misra2012] Rule 10.1 Operands shall not be of an inappropriate essential type.

10.8 Boolean assignment and expression

The C language assignment operator returns a value. It is therefore possible to use this value. However, this is often an unintentional assignment of the developer resulting from confusion between the assignment operator = and the equality operator ==.

GOOD PRACTICE 117

GOOD PRACTICE – Do not use the value returned during an assignment

Information

During the compilation phase with the right options (`-Wall`, *etc.*), a warning will be emitted suggesting in particular to place the assignment in parentheses in the boolean expression (`-Wparentheses` option).

RULE 118

RULE – Assignment prohibited in a boolean expression

An assignment must not be made in any boolean expression. An assignment must be made in an independent statement.

In order to limit the risks of writing an assignment with the = operator instead of a comparison with the == operator, when the comparison is made between a variable and a constant operand, the constant operand should be written as the left operand of the == operator and the variable as the right operand. The compiler raises a warning when trying to assign a value to a constant operand.

GOOD PRACTICE 119

GOOD PRACTICE – Comparison with constant operand on the left

When a comparison involves a constant operand, this should preferably be set as the left operand to avoid an unintended assignment.

Information

This good practice is debatable, and is therefore not enforced but merely advised. Compiling this type of code with strict options (in particular with `-Wall`), as required by section 5.2, is actually sufficient to detect the use of the assignment operator instead of the comparison operator in a boolean expression.

Bad example

The following code contains assignments in Boolean expressions. One of the assignments in a conditional expression is a programming error.

```
if ((x = y + z) > 0) { /* assignment in a Boolean expression and use of the value
    returned by the assignment and constant on the right */
    ...
}
if (z = VALUE) { /* constant on the right and = instead of == */
    ...
}
```

Good example

In the following example, all assignments are made in independent statements and the various problems are corrected.

```
x = y + z;
if (0 < x) {
  ...
}
if (VALUE == z) {
  ...
}
```

10.8.1 References

[Misra2012] Rule 13.4 The result of an assignment operator should not be used.
[Cert] Rule EXP45-C Do not perform assignments in selection statements.
[IsoSecu] No assignment in conditional expressions [boolasign].
[Cwe] CWE-480 Use of incorrect operator.
[Cwe] CWE-481 Assigning instead of comparing.
[Cwe] CWE-482 Comparing instead of assigning.

10.9 Multiple assignment of variables prohibited

The C language allows the same value to be assigned to several variables with a single statement. This multiple assignment is often used for variable initialisations.

However, code containing multiple assignments is difficult to read and also difficult to maintain. Break the multiple variable assignment statement down into as many assignment statements as there are variables.

RULE
120

RULE — Multiple assignment of variables prohibited

Multiple assignment of variables is not authorised.

Bad example

The following example shows a multiple assignment.

```
...
a = b = c = d = 1;  /* multiple assignment */
```

Good example

The following code contains one assignment per variable.

```
...
a = 1;
b = 1;
c = 1;
d = 1;
```

10.10 Only one statement per line of code

The end of a statement in the C language is marked by the semicolon. The C language does not require the developer to write only one statement per line of code.

However, when several statements are present on the same line, the code is less readable. Debugging is also more difficult, since it is not possible to check the execution of code one statement at a time.

The presence of multiple statements per line of code also distorts the metric of the number of lines of code.

RULE 121

RULE – Only one statement per line of code

Bad example

In the following example, the code is difficult to understand.

```
int32_t a; int64_t b;
a = 4; b = a / 6; printf("a = %d, b = %lld\n", a, b);
```

Good example

```
int32_t a;
int64_t b;
a = 4;
b = a / 6;
printf("a = %d, b = %lld\n", a, b);
```

10.11 Use of floating-point numbers

Information

By floating-point numbers, we are referring to numbers using a floating-point representation.

The representation of floating-point numbers in a machine is a complex notion which is often not well known or badly understood and moreover, it is dependent on the precision associated with the type. These floating-point numbers are often a source of errors.

Not all actual values can be represented as floating-point numbers, and other "unnatural" phenomena such as *absorption*[11] and *cancellation*[12] can occur with the use of floating-point numbers, although these points will not be detailed in this guide. Further details are given in the standard IEEE754 [float], which ensures reproducible inter-compiler and inter-architecture behaviour in the presence of floating-point numbers.

In addition, the error associated with the use of these floating-point numbers can become greater than the result of the calculation using them.

Finally, certain elementary properties of real arithmetic are no longer true when using floats: commutativity, associativity, *etc.*

For all of these reasons, the use of floating-point numbers is strongly discouraged. Should the use of floating-point numbers prove necessary for numerical processing for example, the developer will have to ensure that the constant float values are representative and that they are correctly used in accordance with the associated precision.

GOOD PRACTICE 122

GOOD PRACTICE — Avoid floating constants

Do not use floating numerical constants in order to avoid loss of precision and other phenomena related to floating-point numbers. If this cannot be avoided, the representativeness of the float value in question must be checked.

RECO 123

RECOMMENDATION — Limit the use of floating-point numbers to what is strictly necessary

The use of floating-point numbers should be limited.

Float type loop counters are sources of error due to the limited representativeness of this type and the associated complexity.

[11] Phenomenon related to the precision of floats, such as $LargeFloatValue + FloatingEpsilon = LargeFloatValue$, *i.e.* a large float value will "absorb" a small float value

[12] Another phenomenon always related to the precision and representation of floating-points numbers, such that for two close float values, $FloatValue1 - FloatValue2 = 0$, whereas formally, $FloatValue1 \neq FloatValue2$

RULE
124

RULE – No float type loop counter

Loop counters must be of the integer type only, with type overflow verification of the counter values.

The handling of float values in Boolean expressions is also always very risky in connection with problems of representation and precision of these values. The use of the logic operators != or == on floats is incorrect in most cases. The results may depend on the level of optimisation, the compiler itself and the platform used.

RULE
125

RULE – Do not use floating-point numbers for comparisons of equality or inequality

Bad example

```
float y = 0.1; /* not representable in simple precision */
 ...
if (y == 0.1) /* comparison of float value AND 0.1 double value so promotion of y
    */
   printf("equal\n");
else
   if (y == 0.1f) /* 0.1f float value - condition checked here */
      printf("equal2\n");
   else printf("not equal\n");
...
for (float x = 0.1f; x <= 1.0f; x += 0.1f) { /* the loop will be carried out 9 or
    10 times */
}
```

Good example

```
double y = 0.1; /* type correction */
 ...
for (uint count = 1; count <= 10; count ++) {
  float x = count/10.0f; /* 10 passes exactly in the loop */
}
```

10.11.1 References

[Misra2012] Rule 14.1 A loop counter shall not have essentially floating type.
[Cert] Rule FLP30-C Do not use floating-point variables as loop counters.
[Cert] Rule FLP30-C Do not use object representations to compare floating-point values.
[Cert] Rec. FLP00-C Understand the limitations of floating-point numbers.
[Cert] Rec. FLP01-C Take care in rearranging floating-point expressions.
[Cert] Rec. FLP02-C Avoid using floating-point numbers when precise computation is needed.
[Cert] Rec. FLP03-C Detect and handle floating-point errors.
[Cert] Rec. FLP04-C Check floating-point inputs for exceptional values.
[Cert] Rec. FLP05-C Do not use denormalized numbers.
[Cwe] CWE-369 Divide by Zero.
[Cwe] CWE-681 Incorrect conversion between numerical types.

[Cwe] CWE-682 Incorrect calculation.

10.12 Complex numbers

Since C99, the C language supports the calculation of complex numbers with three new built-in types, `double_Complex`, `long double_Complex` and `float_Complex`. With the associated header file `complex.h`, these types are also accessible via `double complex`, `long double complex` and `float complex`. In addition, the three associated imaginary types are also supported: `double_Imaginary`, `long double_Imaginary` and `float_Imaginary` (or with the `double imaginary`, `long double imaginary` and `float imaginary` header).

As these complex numbers are based on a floating representation, their use is strongly discouraged.

RECO 126

RECOMMENDATION — No use of complex numbers

Complex numbers introduced since C99 should not be used.

11
Conditional and iterative structures

11.1 Use of braces for conditionals and loops

The C language does not require the statements of a conditional or a loop to be delimited by braces. A lack of braces makes a conditional or a loop more difficult to read. Furthermore, there is a risk of error if the code is modified: a statement could be added with the intention to be part of the conditional, it will instead end up outside of the conditional.

RULE
127

RULE — Systematic use of braces for conditionals and loops

Never omit braces to delimit a statement block. Braces must be written to delimit a block of statements after loops (for, while, do) and conditionals (if, else).

Bad example

In the code below, a conditional is not delimited by braces.

```
if (x == 0)      /* braces are required, even for a single statement */
  printf("X = 0\n");
  /* An indented statement under the printf may visually suggest that the statement
     is within the if, but it is not. For a jump statement like "goto", this may
     result in a significant portion of the code not being executed. */
  if (x != 0) {
    if (x < 0) {
      ...
      while (x < 0) {
        x++;
        ...
      }
    } else {
      while (x > 0) {
        x--;
        ...
      }
    }
  }
  /* example of ’Apples goto fail */
if (err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
  goto fail;
  goto fail;
  /* other checks, but not reachable due to the duplicate goto fail
     without braces */
fail:
  /* cleaning and releasing of buffers */
  return err;
```

Good example

The following example delimits all the conditionals and loops with braces.

```
if (0 == x) {
  printf("X = 0\n");
} else {
  if (x < 0) {
    ...
    while (x < 0) {
      x++;
      ...
    }
  } else {
    while (x > 0) {
      x--;
      ...
    }
  }
}
/* example of goto fail - Apple */
if (err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0) {
  goto fail;
}
/* other checks */
fail:
  /* cleaning and releasing of buffers */
  return err;
```

11.1.1 References

[Misra2012] Rule 15.6 The body of an iteration-statement or a selection statement shall be a compound-statement.
[Cert] Rec. EXP19-C Use braces for the body of an if, for, or while statement.

11.2 Correct construction and use of switch statements

The `switch` statement of the C language provides an elegant way of writing the handling of different cases based on the value of a variable or an expression. However, this statement is also a source of errors when omitting the break statement: unwanted code may be executed. In fact, since the successive conditions of the `switch` statement are not exclusive, several cases can be activated. It must also be ensured that processing is performed if the value of the expression does not correspond to any of the cases of the `switch` (default case).

RULE 128

RULE — Systematic definition of a default case in `switch`

A `switch-case` must always contain a default case placed last.

RECO 129

RECOMMENDATION — Use of `break` in each case of `switch` statements

By default, a `switch-case` must always contain a break for each case. The absence of a break to avoid duplicating code is tolerated but must be made explicit in a comment.

Information

The -Wimplicit-fallthrough option is used to check the correct application of this recommendation.

The code in each case should be simple and contain few statements. If complex processing is to be carried out in a case, then a function must be defined for this processing and this function must be called from the case.

RECO 130

RECOMMENDATION — No nesting of control structure in a switch-case

Even though C allows it, the nesting of control structures inside a switch should be avoided.

Finally, it is forbidden to declare, initialise variables or introduce code statements within a switch statement before the first label associated with the first case of the switch-case.

RULE 131

RULE — Do not insert statements before the first label of a switch-case

Bad example

In the following example, the final default statement is missing.

```
switch(var) {
  int i = 0; /* not authorized */
  case VAL1:
    ...
    break;
  case VAL2:
    ...
  case VAL3:
    ...
    break;
  /* absence of default and break and no comment to explain the case on the value
      VAL2 */
}
```

Good example

In the following example, a break statement is present for each case. There is also a final default statement to ensure that processing is carried out if the value did not match any of the cases.

```
int i = 0;  /* displaced declaration and initialisation */
switch(var) {
  case VAL1:
      ...
      break;
  case VAL2:
      ...
      /* break voluntarily missing */
  case VAL3:
      ...
      break;
  default:
        ...
}
```

11.2.1 References

[Misra2012] Rule 16.1 All switch statements shall be well-formed.
[Misra2012] Rule 16.2 A switch label shall only be used when the most closely-enclosing compound statement is the body of the switch statement.
[Misra2012] Rule 16.3 An unconditional break statement shall terminate every switch-clause.
[Misra2012] Rule 16.4 Every switch statement shall have a default label.
[Misra2012] Rule 16.6 Every switch statement shall have at least 2 switch-clauses.
[Misra2012] Rule 16.7 A switch expression shall not have a essentially boolean type.
[Cert] Rule DCL41-C Do not declare variables inside a switch statement before the first case label.
[Cert] Rec. MSC01-C Strive for logical completeness.
[Cert] Rec. MSC17-C Finish every set of statements associated with a case label with a break statement.
[Cwe] CWE-484: Omitted Break Statement in Switch.
[IsoSecu] Use of an implied default in a `switch` statement [swtchdflt].

11.3 Correct construction of `for` loops

The C language allows several expressions to be added to the first and last element of a `for` statement, separated by a comma. This allows for example the initialisation of several variables in the first element of the `for`, or the incrementation of several variables in the third element of the `for`. However, the presence of comma-separated expressions in the `for` statement makes it difficult to understand the code and is a source of errors. In addition, the sequencing of statements is already prohibited in section 14.1. If other variables are to be initialised at the start of the loop, they must be initialised just before the `for` statement. If several variables are to be incremented or decremented, they must be modified at the end of the loop.

Furthermore, the C language does not require each element (initialisation, stop condition and increment/decrement) of the `for` statement to be completed. It is possible to leave the initialisation empty, for example, or even to leave each element empty, resulting in an infinite loop. In the case of an infinite loop, the form `while (1) { ... }` is to be preferred to the form `for(;;) { ... }`.

RULE 132

RULE – Correct construction of `for` loops

Each element of a `for` loop must be completed and contain exactly one statement. Thus a `for` loop must contain an initialisation of its counter, a stop condition on its counter, and a loop counter increment or decrement.

Bad example

In the following example, the comma is used to separate several initialisations and modifications of variables in the first and third element of the `for`. In addition, `for` loops should be replaced by `while() { ... }` or `do { ... } while ()` loops.

```
#define MAX_LOOP    10U
for(;;) {
  data = read();
  if (0 == data) {
    break;
  }
  ...
}
for (i = 0;;) { /* missing elements */
  max = some_function(i);
  i++;
  if (i >= max) {
    break;
  }
}
...
for (i = 0, a = 0; i < MAX_LOOP; i++, a += 2) {
  ...
  some_function(a);
  ...
}
```

Good example

The following code only initialises the loop counter in the first `for` element, and only increments the counter in the third element. The for loops contain all its elements (initialisation, stop condition, counter increment).

```
#define MAX_LOOP    10U
for (i = 0; i < arraySize; i++) {
  ...
  dataArray[i] = some_function(i);
  ...
}
while (1) {
  data = read();
  if (0 == data) {
    break;
  }
  ...
}
i = 0;
do {
  max = some_function(i);
  i++;
}
while (i < max);
...
a = 0;
for (i = 0; i < MAX_LOOP; i++) {
  ...
  some_function(a);
  ...
  a += 2;
}
```

11.3.1 References

[Misra2012] Rule 14.2 A for loop shall be well-formed.
[Misra2012] Rule 15.6 The body of an iteration-statement or a selection statement shall be a compound-statement.
[Cert] Rec. EXP15-C Do not place a semicolon on the same line as an if, for, or while statement.
[Cert] Rec. EXP19-C Use braces for the body of an if, for, or while statement.

11.4 Changing of a for loop counter forbidden in the body of the loop

Changing the counter of a `for` loop within the loop makes it difficult to understand and maintain the code. Furthermore, this can lead to an infinite loop when the comparison operator in the loop condition is an equality or inequality. If the loop accesses an array, there is also a risk of overflow.

The loop counter should only be modified in the third part of the `for` loop.

It is common to modify within the body of the loop a flag or other variable that intervenes at the conditional stop expression of the `for` loop. This scenario must then be replaced by the use of a `break` allowing the exit from the loop.

RULE 133

RULE — Change to a counter of a `for` loop forbidden in the body of the loop

The counter of a `for` loop must not be changed inside the body of the `for` loop.

Bad example

The example below shows a `for` loop with a modification of its counter in its body. There is a risk of an infinite loop.

```
#define MAX_LOOP 10U

for (i = 0; i != MAX_LOOP; i++) {
  ...
  if (...) {
    /* if the condition is met with i == 9, we end up in an infinite loop. */
    i++;
  }
  ...
}
```

Good example

In the following code, no changes are made to the counter in the body of the `for` loop.

```
#define MAX_LOOP 10U

for (i = 0; i < MAX_LOOP; i++) {
  ...
  if (some_function()) {
    break; /* loop stopped */
  }
  ...
}
```

11.4.1 References

[Misra2012] Rule 14.2

[Cert] Rule ARR30-C Do not form or use out-of-bounds pointers or array subscripts.

12
Jumps in the code

12.1 Do not use backward goto

The use of a backward goto makes proofreading and maintenance of the code very difficult and is a source of errors such as unwanted infinite loops. If this need arises, it is because the algorithm to be implemented actually comprises a loop. Then use the control structures proposed by the language for the loops, in other words for, while or do-while.

RULE
134

RULE – No use of backward goto

Prohibit within a function the use of goto statements referring to a label that is placed before this goto statement.

Bad example

The following example contains a backward goto. This choice of implementation corresponds to an assembler approach, and does not take advantage of the higher level possibilities offered by the C language.

```
#define BUFFER_SIZE 100U
void foo() {
  uint8_t s[BUFFER_SIZE];
  uint8_t x = 0;
  my_loop:
    s[x] = x;
  x++;
  if(x < BUFFER_SIZE)
    {
      goto my_loop;         /* backward goto prohibited */
    }
}
```

Good example

The following example uses a loop type control structure. There is no need for a backward goto.

```
#define BUFFER_SIZE 100U
void foo() {
  uint8_t s[BUFFER_SIZE];
  uint8_t x = 0;
  for(x = 0; x < BUFFER_SIZE; x++) {
    s[x] = x;
  }
}
```

12.1.1 References

[Misra2012] Rule 15.1 The goto statement should not be used.

12.2 Limited use of forward goto

The *forward* `goto` can simplify error management and reduce the number of exit points from a function. However, with a *forward* `goto` outside of a conditional statement, code may be executed with variables that have not been initialised. Another possible serious consequence is, for example, forgetting to free memory or resources among other things. The code must therefore be modified in order to use control structures that avoid the use of `goto`.

The *forward* `goto` should only be used for error management, and the number of labels should be kept to a minimum.

RECO
135

RECOMMENDATION – Limited use of forward `goto`

The use of a forward `goto` is tolerated only in cases where it allows:

- the number of exit points from the function to be significantly limited;
- the code to be made much more readable.

The label(s) referenced by the `goto` statements must all be located at the end of the function.

Good example

The following code does not use a forward `goto`, but uses the control structures proposed by the C language.

```
#define BUFFER_LEN    (128U)
int32_t my_function(int32_t a) {
  FILE* f = NULL;
  uint8_t *buffer = NULL;
  int32_t result = ERR_UNDEFINED;
  f = fopen("/my/path", "r");
  if(NULL == f) {
    result = ERR_FOPEN;
  } else {
    buffer = (uint8_t *) malloc(BUFFER_LEN * sizeof(uint8_t));
    if(NULL == buffer) {
      result = ERR_MALLOC;
    }
    else
    {
      ...
      free(buffer);
      buffer = NULL;
      result = ERR_NOERROR;
    }

    fclose(f);
    f = NULL;
  }
  return result;
}
```

Tolerated example

The following example uses the forward `goto` for error management. This scenario is tolerated.

```c
int32_t my_function(int32_t a) {
  FILE* f = NULL;
  uint8_t *buffer = NULL;
  int32_t result = ERR_UNDEFINED;
  f = fopen("/my/path", "r");
  if(NULL == f) {
    result = ERR_FOPEN;
      goto cleanup;
  }
  buffer=(uint8_t *)malloc(BUFFER_LEN * sizeof(uint8_t));
  if(NULL == buffer) {
    result = ERR_MALLOC;
    goto cleanup;
  }
  ...
  result = ERR_NOERROR;
cleanup:
    if(NULL != f) {
        fclose(f);
        f = NULL;
      }
  if(NULL != buffer) {
    free(buffer);
    buffer = NULL;
  }
  return result;
}
```

12.2.1 References

[Misra2012] Rule 15.1 15.5.

13
Functions

13.1 Correct and consistent declaration and definition

Declaration/Prototype of function

The declaration of a function or its *prototype* is a statement which defines three elements: the return type of the function, its name and the list of its arguments, followed by a semicolon.

Definition of function

The definition of a function is the body of the function, *i.e.* the set of statements it executes. A function definition also contains a prototype of the function.

C90 allows for the implicit declaration of functions, whether it be the absence of a return type or the absence of a function declaration. C99 is stricter and imposes at least one type specifier.

The C language, in its successive versions, proposes different forms for the declaration of functions. Combining these different forms of function declaration is not recommended since this risks resulting in a much less precise analysis of the code, and leading to problems when editing links.

Information

Compilers raise a warning (`-Wimplicit-function-declaration` or `-Wimplicit-int` or `-Wreturn-type`), but assume an implicit extern int type, *i.e.* by default, a function not associated with a return type has an integer type.

RULE 136

RULE — Any (non-static) function defined must have a function declaration/prototype

RULE 137

RULE — The prototype declaration of a function must be consistent with its definition

The types of parameters used to define and declare a function must be the same.

RULE 138

RULE — Every function must have an explicit return type and parameter list associated with it

Each function is explicitly defined with a return type. Functions without a return value must be declared with a `void` type parameter. In the same way, a function without a parameter must be defined and declared with `void` as argument.

Activating the compiler warnings is used to find out which functions are not correctly declared (missing return type, inconsistency of types between declaration and definition).

Bad example

In the example below, the return type is missing for a declaration, a function without parameters is not correctly declared and there is an inconsistency between the declaration and the definition.

```
 /* header.h */
foo(uint8_t a);  /* the return type is missing */
uint32_t bar(uint16_t b);
void car();     /* void is missing in the parameter type to indicate that the
    function does not take parameters */
/* file.c */
foo(uint8_t a) {
    ...
}
uint32_t bar(int32_t b) { /* after the declaration, b must be a uint16_t */
    ...
}
void car() {
    ...
}
```

Good example

The following example makes a correct declaration and definition of functions.

```
/* header.h */
void foo(uint8_t a);
uint32_t bar(uint16_t b);
void car(void);
/* file.c */
void foo(uint8_t a) {
    ...
}
uint32_t bar(uint16_t b) {
    ...
}
void car(void) {
    ...
}
```

13.1.1 References

[Misra2012] Rule 8.1 Types shall be explicitly specified
[Misra2012] Rule 8.2 Function types shall be in prototype form with named parameters
[Misra2012] Rule 8.3 All declarations of an object or function shall use the same names and type qualifiers
[Misra2012] Rule 17.3 A function shall not be declared implicitly

[Misra2012] Rule 17.5 The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements
[Cert] Rec. DCL07-C Include the appropriate type information in function declarators
[Cert] Rec. DCL20-C Explicitly specify void when a function accepts no arguments
[Cert] Rule DCL31-C Declare identifier before using them
[Cwe] CWE-628 Function call with incorrectly specified arguments
[IsoSecu] Using a tainted value as an argument to an unprototyped function pointer [taintnoproto].
[IsoSecu] Calling functions with incorrect arguments [argcomp].

13.2 Documentation of functions

Incomplete documentation of a function can lead to programming errors. This includes the functionality of the function, a precise description of the parameters justifying the passages by value or reference, but also the conditions to be checked for the correct use of the function.

Passing of parameter by value or copying

When passing a parameter by value (or by copying), the value of the actual argument on the function call is sent (copied) to the respective formal argument of the called function. The direct consequence is that any change made to a formal argument is not propagated to the actual argument.

Passing of parameters by reference or pointer

When passing a parameter by reference, the address of the value of the actual argument on the function call is sent to the respective formal argument of the called function. The direct consequence is that any change made to a formal argument will be propagated to the actual argument.

RECO 139

RECOMMENDATION — Documentation of functions

All functions must be documented. This includes:

- a description of the function and the processing carried out;
- the documentation of each parameter, the direction of the parameter (input, output, input and output) and any condition existing on it;
- the possible return values must be described.

This also includes the conditions for proper use of the function to be specified in the prototype, especially in the case of portable code (Linux, Windows).

RECO 140

RECOMMENDATION — Specify call conditions for each function

13.3 Validation of input parameters

Programming errors can lead to invalid values being passed as function parameters. If there is no parameter validity check, the behaviour of the function is undefined.

It is therefore necessary to check:

- address consistency (non-null, alignment, *etc.*);
- parameter values (value ranges, *etc.*).

Some generic measures must be applied, such as the return of an error code for example.

RULE
141

RULE — The validity of all the parameters of a function must systematically be questioned

This includes:

- the validity of the addresses for pointer-type parameters must be checked (pointers must be non-null, properly aligned, *etc.*);
- the parameters must be checked to ensure that they belong to their domain.

This applies to the functions defined by the developer (*cf.* section 13.2) as well as to the functions of the standard library.

Bad example

In the following example, the validity of the parameters is not checked.

```
double division(int32_t n, int32_t d) {
  /* d != 0 not verified */
  return ((double)n) / ((double)d);
}
```

Good example

Example 1:
The following code shows an example where the validity of the parameters is checked:

```
double division(int32_t n, int32_t d) {
  double res = 0.0;
  if(0 == d) {
    /* error handling */
  }
  else {
    res = ((double)n) / ((double)d);
  }
  return res;
}
```

Example 2:
The following code shows a second example where the validity of the parameters is checked:

```
uint8_t encrypt(uint8_t *output, int32_t *output_len,
                 const uint8_t *input, const int32_t input_len,
                 encrypted_ctx *ctx) {
    uint8_t err = 0;
    if((NULL == output) || (NULL == output_len) || (NULL == encrypted_ctx)) {
        err++; /* error handling */
    }
    if((NULL == input) || (input_len <= 0) || (input_len > MAX_INPUT_LEN)) {
        err++; /* error handling */
    }
    if(0 == err) {
        /* API code */
    }
    return err;
}
```

13.3.1 References

[Misra2012] The validity of values passed to library shall be checked.
[Misra2012] Dir 4.1 Run-time failures shall be minimized
[Cert] API00-C Functions should validate their parameters
[Cert] Rule ARR38-C Guarantee that library functions do not form invalid pointers.
[Cert] Rec. MEM10-C Define and use a pointer validation function.
[Cwe] CWE-20 Insufficient input validation
[Cwe] CWE-628 Function call with incorrectly specified arguments.
[Cwe] CWE-686 Function call with incorrect argument type.
[Cwe] CWE-687 Function call with incorrectly specified argument value.
[IsoSecu] Calling functions with incorrect arguments [argcomp].

13.4 Use of the qualifier `const` for pointer-type function parameters

When reading a function prototype with pointer-type parameters, the absence of the `const` qualifier may suggest that a modification will be made to the memory area pointed to. The absence of this qualifier when it should be used makes the definition of interfaces unclear and complicates the proofreading of code. When declaring a function with parameter pointers, the developer should immediately consider how the pointers will be used and use `const` by default unless the memory area pointed to is changed when the function is executed.

RULE 142

RULE — Pointer-type function parameters which point to memory that is not to be changed must be declared as `const`

Mark as `const` all pointer-type parameters of a function that point to a memory area that is read-only in the body of the function. The `const` qualifier must be applied to the pointed object.

Bad example

The following example should use `const` for its parameter.

```
uint32_t foo(uint32_t *val) {
  /* val lue */
  uint32_t ret = 0;
  if(TEST_VALUE > *val) {
      ret = (*val) * 2;
  } else {
      ret = (*val);
  }
  return ret;
}
```

Good example

In the following example, `const` is correctly used for the pointer parameter. The memory area pointed to is not in fact modified in the body of the function.

```
uint32_t foo(const uint32_t *val) {
  uint32_t ret = 0;
  if (NULL != val){
    if(TEST_VALUE > *val) {
      ret = (*val) * 2;
    } else {
      ret = (*val);
    }
  }
  return ret;
}
```

13.4.1 References

[Cert] Rec. DECL00-C Const-qualify immutable objects.
[Cert] Rec DECL13-C Declare function parameters that are pointers to values not changed by the function as const.
[Cert] Rule EXP40-C Do not modify constants objects.
[Misra2012] Rule 8.13 A pointer should point to a const-qualified type whenever possible.
[Cwe] CWE-20 Improper Input Validation.
[Cwe] CWE-369 Divide by Zero.

13.5 Using `inline` functions

C99 introduced the new `inline` keyword as a function specifier. An `inline` function declared with external linkage but not defined in the same file leads to undefined behaviour. The declaration and definition of an `inline` function must therefore be in the same compilation unit.

Information

An `inline` function can be accessed by multiple files by being declared in a header file.

RULE 143

RULE — `inline` functions must be declared as `static`

To avoid undefined behaviour, an `inline` function is systematically `static`.

13.5.1 References

[Misra2012] Rule 8.10 An inline function shall be declared with the static storage class.
[Cert] Rec. DCL15-C Declare file-scope objects or functions that do not need external linkage as static.
[Cert] MSC40-C Do not violate constraints.

13.6 Redefining functions

A function name can be declared by the programmer even if it is a name already defined in the standard library or in another library. This declaration may lead to confusion. Every function must have its own name.

RULE 144

RULE — Do not redefine functions or macros from the standard library or another library

Identifiers, macros, or function names that are part of the standard library or another library used must not be redefined.

Bad example

In the following example, confusion will occur due to the use of a function name that already exists in the standard library.

```
/* do not reuse the name of the standard library */
void* malloc (size_t taille);
```

Good example

The following example defines a name that does not clash with the name of a function in the standard library.

```
/* renaming of the function */
void* mymalloc (size_t taille);
```

13.6.1 References

[Misra2012] Rule 5.8 Identifiers that define objects or functions with external linkage shall be unique.
[Misra2012] Rule 5.9 Identifiers that define objects or functions with internal linkage shall be unique.

13.7 Mandatory use of the return value of a function

A function, whose return type is not `void`, returns a value indicating the success or failure of the processing or the computation performed by this function. This function return is a very important source of information and allows unexpected behaviour or even errors to be identified as soon as possible. These function returns must therefore always be read and managed.

The calling function must test the value returned by the function to ensure its validity against the interface documentation (returned value within the value range or returned value corresponding to a success or error code).

RULE 145

RULE – The return value of a function must always be tested

When a function returns a value, the returned value must systematically be tested.

Bad example

In the following code, the value returned by the function is not tested and no processing is performed if an error has occurred.

```
struct stat o_stat_buffer;
stat("somefile.txt", &o_stat_buffer);
/* success of the stat function not tested */
...
```

Good example

In the following code, the value returned by the function is correctly tested.

```
struct stat o_stat_buffer;
uint8_t          i_result = 0;
i_result = stat("somefile.txt", &o_stat_buffer);
if (0 != i_result) {
    /* erreur */
    return 0;
}
...
```

13.7.1 References

[Cert] Rec. EXP12-C Do not ignore values returned by functions.
[Misra2012] Dir. 4.7: If a function returns error information, then that error information shall be tested.
[Misra2012] Rule 17.7 The value returned by a function having non-void return type shall be used.
[Cwe] CWE-252 Unchecked Return Value.
[Cwe] CWE-253 Incorrect Check of Function Return Value.
[Cwe] CWE-754 Improper check for unusual or exceptional conditions.

13.8 Implicit return prohibited for non-`void` functions

In the absence of an explicit return value for all paths of a function returning a value (non-`void` function), some compilers do not always generate an error. The behaviour of the code is then undefined. Some compilers return an arbitrary value.

RULE 146

RULE — Implicit return prohibited for non-`void` type functions

All paths of a non-`void` function must return a value explicitly.

Bad example

In the following example, there are paths that do not explicitly return a value.

```
uint32_t encr_data(const uint8_t *p_data, uint32_t ui32_data_len,
                   uint8_t **pp_encrypted_data, uint32_t *ui32_encrypted_data_len)
                       {
  uint8_t *p_encrypted_data = NULL;
  if (NULL != p_data
      && NULL != pp_encrypted_data
      && NULL != ui32_encrypted_data_len) {
      if (ui32_data_len > 0) {
         p_encrypted_data = (uint8_t *)calloc(ui32_data_len, sizeof(uint8_t));
         ...
         return 1;
       }
     }
     /* implicit return */
   }
```

Good example

In the following code, the function code always explicitly returns a value.

```
uint32_t encr_data(const uint8_t *p_data, uint32_t ui32_data_len,
                   uint8_t **pp_encrypted_data, uint32_t *ui32_encrypted_data_len)
                       {
  uint32_t ui32_result_code = 0;
  uint8_t *p_encrypted_data = NULL;
  if (NULL == p_data
  || NULL == pp_encrypted_data
  || NULL == ui32_encrypted_data_len) {
    ui32_result_code = 0;
    goto End;
  }
  if (0 == ui32_data_len) {
    ui32_result_code = 0;
    goto End;
  }
  p_encrypted_data = (uint8_t *)calloc(ui32_data_len, sizeof(uint8_t));
  ...
  (*pp_encrypted_data) = p_encrypted_data;
  ui32_result_code = 1;
End:
  return ui32_result_code;
}
```

13.8.1 References

[Misra2012] Rule 17.4 All exit paths from a function with non-void return type shall have an explicit return statement with an expression.
[Cert] Rule MSC37-C Ensure that control never reaches the end of a non-void function.

13.9 No passing by value of a structure as function parameter

It is possible with the C language to pass structure as parameters of a function. These are then copied to the stack. However, this is detrimental to performance and increases the risk of stack overflow or even leakage of sensitive data.

The parameter corresponding to a structure must be passed in the form of a `const`–qualified pointer. Only the address of the structure is then copied to the stack. Furthermore, the `const` modifier prevents changes to the pointed object (which is desirable when passing the structure by value).

RULE
147

RULE – Structures must be passed by reference to a function

Do not pass structure type parameters by copying when calling a function.

Bad example

In the following example, the parameter is passed by value and not by address.

```
#define STR_SIZE 20U
typedef struct
{
  unsigned char surname[STR_SIZE];
  unsigned char firstname[STR_SIZE];
} person_t;

uint32_t add_person(person_t person) {
  size_t   sz_surname_len = 0;
  size_t   sz_firstname_len = 0;
  sz_surname_len = strlen(person.surname);
  sz_firstname_len = strlen(person.firstname);
  if (0 != sz_surname_len && 0 != sz_firstname_len) {
    ...
    ui32_result = 1;
  } else {
    ui32_result = 0;
  }
  return ui32_result;
}
...
void some_function() {
  person_t person;
  ...
  add_person(person);
  ...
}
```

Good example

The following code correctly passes a structure type parameter using a pointer.

```
#define STR_SIZE 20U
typedef struct
{
  unsigned char surname[STR_SIZE];
  unsigned char firstname[STR_SIZE];
} person_t;
```

```
uint32_t add_person(const person_t *person) {
  uint32_t ui32_result = 0;
  size_t   sz_surname_len = 0;
  size_t   sz_firstname_len = 0;
  if (NULL != person) {
    sz_surname_len = strlen(person->surname);
    sz_firstname_len = strlen(person->firstname);

    if (0 != sz_surname_len && 0 != sz_firstname_len) {
      ...
      ui32_result = 1;
    } else {
      ui32_result = 0;
    }
  }
  else {
      ui32_result = 0;
  }
  return ui32_result;
}

void some_function() {
  person_t person;
  ...
  add_person(&person);
  ...
}
```

13.10 Passing an array as a parameter for a function

When a function takes a pointer as a parameter, it is not possible to determine whether the pointer is the address of the first element of an array or whether it points to a single element.

To suppress this ambiguity, it is preferable to use the form with [] for an array type parameter as indicated in sub-section 8.1.

RECO 148

RECOMMENDATION — Passing of an array as a parameter for a function

There are several ways to pass an array as a parameter for a function. When passing by pointer, it must be specified in the function documentation that the parameter corresponds to an array and also use the dedicated array notation.

Warning

For a multi-dimensional array, only the first dimension of the array can remain undefined when passing as a parameter, which therefore means defining the following dimensions. For example, for a two-dimensional array, using tab[][] as a parameter is a mistake, as a minimum the second dimension must be specified.

Bad example

The following example shows a prototype function with a pointer-type parameter. It is an array passed as a parameter but this cannot be guessed from the function signature. In this example, tab can:

- either be an integer passed by address

- or be an array of integers.

```
void func(int32_t *tab, uint32_t count);
```

Good example

In this second example, the notation and comments ensure that it can be immediately determined that the parameter is indeed an array.

```
void func(int32_t tab[], uint32_t count); /* tab is an array of count elements */
```

13.11 Mandatory use in a function of all its parameters

Failure to use a parameter in the implementation of a function is usually a developer error. It also consumes unnecessary space on the stack.

The prototype of the function must be modified if the parameter is not useful.

However, in some cases, it may be justified not to use one (or more) parameters of a function:

- the function corresponds to a callback function whose prototype is imposed;
- for reasons of compatibility with existing code when upgrading a library. A previously used parameter is no longer used;
- in the case of a future development in which the parameter will be used.

In all these cases, a comment must then explicitly state why the parameter is ignored.

RECO 149

RECOMMENDATION — Mandatory use in a function of all its parameters

All the parameters present in the prototype of the function must be used in its implementation.

Information

The `-Wunused-parameter` option is used to provide alerts in this scenario.

Bad example

In the following example, a parameter of the function is not used and should therefore be deleted.

```
uint32_t compute_data(uint32_t ui32A, uint32_t ui32B, uint32_t ui32C) {
  uint32_t ui32_result = 0;
  ui32_result = 2 * ui32A + 2 * ui32B;
  return ui32_result;
}
```

Good example

In the following code, there are no unused parameters in the implementation of the function.

```
uint32_t compute_data(uint32_t ui32A, uint32_t ui32B) {
  uint32_t ui32_result = 0;
  ui32_result = 2 * ui32A + 2 * ui32B;
  return ui32_result;
}
```

13.11.1 References

[Misra2012] Rule 2.7 There should be no unused parameters in functions.
[Cert] Rule EXP37-C Call functions with the correct number and type of arguments.

13.12 Variadic functions

Variadic functions (*i.e.* those with a variable number of arguments or with varying types) can pose several problems. It is not advisable to define variadic functions, but the standard library itself contains several such definitions that are often used. The type of arguments of a variadic function is not checked by the compiler, by default, which can, if these functions are used incorrectly, lead to some surprises such as abnormal termination or unexpected behaviour.

Information

The `-Wformat=2` option, required by subsection 5.3.1, allows the compiler to extend its checks to arguments of variadic functions.

When `NULL` is passed to a classic function, `NULL` is converted to the correct type. This type conversion does not work with the variadic functions since the “right type” is not known. In particular, the standard allows `NULL` to be an integer constant or a pointer constant so on platforms where `NULL` is also an integer constant, passing `NULL` for variadic functions can lead to undefined behaviour.

RULE 150

RULE – Do not call variadic functions with `NULL` as an argument

Bad example

```
...
unsigned char *string = NULL;
printf("%s %d\n", string, 1); /* undefined behaviour */
...
```

Good example

```
...
unsigned char *string = NULL;
printf("%s %d\n", (string ? string : "null"), 1); /* not passing NULL */
...
```

13.12.1 References

[Misra2012] The features of <stdarg.h> shall not be used.
[Cert] Rec. DCL10-C Maintain the contract between the writer and the caller of variadic functions.
[Cert] Rec. DCL11-C Understand the type issues associated with variadic functions.
[Cert] Rule EXP47-C Do not call va_arg with an argument of the incorrect type.
[Cert] Rule MSC39-C Do not call va_arg on a va_list that has an indeterminate value.
[Cwe] CWE-628 Function call with incorrectly specified arguments.
[IsoSecu] Calling functions with incorrect arguments [argcomp].

14
Sensitive operators

14.1 Use of the comma prohibited for statement sequences

The comma should be used as a separator for the parameters of a function or as a separator for initialising fields in a structure or array. Use of the comma is also tolerated when making a declaration, subject to compliance with the other rules on multiple declarations. However, the use of commas to string together statements in the C language makes the code difficult to read, and may lead to unexpected results.

RULE
151

RULE — Use of the comma prohibited for statement sequences

The comma is not authorised when sequencing code statements.

The comma must be replaced by a semicolon for statement sequences. This means that:

- braces become necessary;
- the parameters of `for` loops must be reorganised.

Bad example

The following example makes use of the comma in expressions. It is not possible to know the result of these statements when reading the code.

```
int32_t i = (j = 2, 1);
y = x ? (a++, a + 4) : c;
z = 3 * b + 2, 7 * c + 42;
a = (b = 2, c = 3, d = 4);
for(i = 0, j = SZ_MAX; i < SZ_MAX; i++, j--) {
  ...
}
```

Good example

In the following code, the comma is only used for the declaration of variables.

```
int32_t i, j;
i=1;
j=2;
if (0 != x) {
  a++;
  y = a + 4;
} else {
  y = c;
}
z = 3 * b + 2;
j = SZ_MAX;
```

```
for(i = 0; i < SZ_MAX; i++) {
  ...
  j--;
}
```

14.1.1 References

[Misra2012] Rule 12.3: The comma operator shall not be used.

14.2 Using pre/postfix ++ and -- operators and compound assignment operators

When the pre/postfix ++ and -- operators are used within a calculation, it is very difficult to establish the result of the calculation when analysing the code. Moreover, it is also a source of confusion and even errors for the developer. These operators must therefore be used alone in a statement. As a result, since pre/postfix operators are semantically equivalent when used in isolation in a statement (*i.e.* `i++; ++j;`), only postfix operators are authorised in order to avoid any ambiguity[13]. Prefix operators will not be used.

RECO 152

RECOMMENDATION — The prefix operators ++ and -- should not be used

Pre-increment and pre-decrement operators will not be used.

Lastly, complex statements will be broken down into simple elements.

RECO 153

RECOMMENDATION — No combined use of postfix operators with other operators

Post-increment and post-decrement operators should not be mixed with other operators.

Lastly, and again for readability reasons, it is recommended not to use combined assignment operators (>>=, &=, *=, *etc.*).

RECO 154

RECOMMENDATION — Avoid the use of combined assignment operators

Bad example

The code below uses postfix operators mixed with other operators. The behaviour of this code is not specified. It depends on the compiler used.

[13] The choice of postfix operators can be discussed since the two operators used in isolation are equivalent.

```
#define TAB_SIZE 25U

...

uint32_t x;
uint8_t  b[TAB_SIZE] = { 0 };
uint16_t i = 0;
x = foo(i++, i);  /* not specified: problem with the order of evaluation of the
    parameters */

...

uint32_t foo(uint16_t a, uint16_t b) {
  return a * b;
}
```

Good example

In the following example, postfix operators are used in isolated statements.

```
#define TAB_SIZE 25U

...

uint32_t x;
uint8_t  b[TAB_SIZE] = { 0 };
uint16_t i = 0;
i++;  /* N.B. replacing this statement with ++i; would not change the behaviour of
    the program in any way */
x = foo(i, i);

...

uint32_t foo(uint16_t a, uint16_t b) {
  return a * b;
}
```

14.2.1 References

[Misra2012] Rule 13.3 A full expression containing an increment or decrement operator should have no other potential side effects other than that caused by the increment or decrement operator. [Cert] Rule EXP30-C Do not depend on the order of evaluation for side effects.

14.3 No nested use of the ternary operator “?:”

The ternary operator `?:` can be used to concisely write an assignment of a variable based on a condition.

However, when the ternary operator is used with a complex conditional expression, or if several ternary operators are nested, understanding the code and its maintenance becomes difficult.

In the case of complex expressions, an `if-else` conditional must be used.

RULE 155

RULE – No nested use of the ternary operator ? :

The nesting of ternary operators `?:` is prohibited.

Moreover, if the types of expressions are different in the two “branches”, this implies an implicit cast according to the value of the condition of the ternary operator.

RULE
156

RULE — Correct construction of the expressions with the ternary operator ? :

The expressions resulting from the ternary operator `?:` must be exactly of the same type to avoid any type conversion.

Bad example

In the following example, the nested ternary expression makes it difficult to understand the code, and the expressions of the two branches are not of the same type.

```c
y = (x<42) ? 1042 : (t> 0) ? -1042 : 0.0;
/* integer and float, therefore implicit cast */
```

Good example

The following example uses several `if-else` conditionals in order to handle the assignment of a value to the variable y which depends on several conditions.

```c
if(x < 42) {
  y = 1042;
} else {
  if(t > 0) {
    y = -1042;
  } else {
      y = 0;
  }
}
```

14.3.1 References

15
Memory management

15.1 Dynamic memory allocation

For all objects dynamically allocated by the developer, different rules must be respected. First of all, the developer must ensure that they have allocated enough memory for the object in question. A common error is to apply the `sizeof` operator on a pointer of the object to be allocated instead of the object to be allocated directly, or not to apply this operator to the correct type.

RULE 157

RULE — Dynamically allocate sufficient memory space for the allocated object

For a `ptr` pointer, it is preferable to use `ptr=malloc(sizeof(*ptr));` wherever possible.

In addition, any dynamically allocated memory should be freed as soon as possible.

RULE 158

RULE — Free dynamically-allocated memory as soon as possible

Any dynamically-allocated memory space must be freed up when it is no longer needed.

This rule echoes that of section 8.6.

For objects storing sensitive data, the memory areas must be reset before being freed.

RULE 159

RULE — Sensitive memory areas must be reset before being freed

Warning

It is crucial to ensure that this memory reset code is not optimised and is retained on compilation. Most compilers consider this reset as dead code since the associated variables are not used afterwards. Generally speaking, the levels of optimisation should not be pushed at compilation but sometimes, even at a low level of optimisation, one must unfortunately recode one's own `memset` to avoid this kind of inconvenience.

It is also important to note that memory release is only authorised for dynamically-allocated objects.

RULE 160

RULE — Do not free memory not allocated dynamically

Finally, realloc must not be used to modify memory space allocated dynamically. This function can in fact change the memory space allocated to an object by increasing or decreasing its size, but can also free the memory of the object passed as parameter. Because of the risks associated with memory handling or the potential double memory release corresponding to undefined behaviour, use of this function should be avoided.

RULE 161

RULE — Do not change the dynamic allocation via realloc

Warning

If this fails, the realloc function returns NULL, but the initial memory location remains intact and is therefore still accessible.

Bad example

```
#include <stdlib.h>
void fonc(size_t len){
  long *p;
  p = (long *) malloc(len *sizeof(int)); /* incorrect type */
  ...
  p = (long *) realloc(p,0); /* release of p via realloc */
  ...
  free(p); /* double release of p */
}
```

Good example

```
#include <stdlib.h>
void fonc(size_t len){
  long *p;
  p = (long *) malloc(len *sizeof(long)); /* corrected type */
  ...
  free(p); /* realloc deleted and replaced by free */
}
```

15.1.1 References

[Cert] Rec. MEM00-C Allocate and free memory in the same module, at the same level of abstraction.
[Cert] Rule MEM31-C Free dynamically allocated memory when no longer needed.
[Cert] Rule MEM34-C Only free memory allocated dynamically.
[Cert] Rule MEM35-C Allocate sufficient memory for an object.
[Cert] Rule MEM36-C Do not modify the alignment of objects by calling realloc.
[Cert] Rec. MEM03-C Clear sensitive information stored in reusable resources.

[Cert] Rec. MEM04-C Beware of zero-length allocations.
[Misra2012] Rule 22.1 All resources obtained dynamically by means of Standard Library functions shall be explicitly released.
[Misra2012] Rule 22.2 A block of memory shall only be freed if it was allocated by means of a Standard Library function.
[Cwe] CWE-226 Sensitive information uncleared before release.
[Cwe] CWE-244 Failure to clear heap memory before release (”heap inspection”).
[Cwe] CWE-590 Free of memory not on the heap.
[Cwe] CWE-672 Operation on a resource after expiration or release.
[Cwe] CWE-131 Incorrect Calculation of Buffer Size.
[Cwe] CWE-680 Integer Overflow to Buffer Overflow.
[Cwe] CWE-789 Uncontrolled Memory Allocation.
[IsoSecu] Accessing freed memory [accfree].
[IsoSecu] Freeing memory multiple times [dblfree].
[IsoSecu] Reallocating or freeing memory that was not dynamically allocated [xfree].
[IsoSecu] Taking the size of a pointer to determine the size of the pointed-to type [sizeofptr].
[IsoSecu] Allocating insufficient memory [insufmem].

15.2 Use of the `sizeof` operator

The `sizeof` operator is essential in C in order to know the size of an object in memory. However, careless use of this operator can lead to unexpected behaviour and result in an incorrect memory size or lead to an unevaluated expression.

It is preferable to use the object type and not the identifier of a variable as a parameter of the `sizeof` operator. Using the identifier has the advantage of being “resistant” to the associated type change, but it is then necessary to ensure that the use of the `sizeof` operator is correct.

In order to avoid problems related to the alignment of the members of a structure, it is common to use:

- either the pre-compilation directive *pack*;
- or an explicit padding field.

Warning

The *pack* directive is not standard.

If an alignment of the members of a structure is required, the pre-compilation *pack* directive or padding fields can be used.

Moreover, the use of the idiomatic expression `sizeof(array)/sizeof(array[0])` to determine the number of elements in an array is quite traditional, but great care must be taken in its use. This expression is only correct if the `sizeof` operator is applied to the array in the block in which the array is declared. The result of this expression will be quite different if the `sizeof` operator is applied to an array passed as parameter because this array will then be a pointer type and no longer an array.

RULE 162

RULE — Correct use of the sizeof operator

An expression contained in a sizeof must not:

- contain the operator "=", since the expression will not be evaluated;
- contain pointer dereferencing;
- be applied to a pointer representing an array.

Warning

The sizeof operator does not return the size of the object, but the size used in memory.

Bad example

The following example shows incorrect use of the sizeof operator.

```
uint8_t tab[LEN];
typedef struct s_example {
  uint32_t ui_field1;
  uint8_t ui_field2;
} t_example;

int32_t i, isize;
t_exemple test;
t_exemple* ptr;

i = 5;
ptr = NULL;
isize = sizeof(i = 1234); /* the expression i= 1234 will not be processed */
/* i has the value 5, and not 1234. isize equals 4 */
isize = sizeof(t_example); /* the value returned by sizeof is 8 for 32-bit
    alignment */
isize = sizeof(*ptr); /* the expression sizeof(*ptr) returns the size of the
    structure t_example */
void crawl_tab(uint8_t tab[])
{
  for (size_t i = 0; i < sizeof(tab) / sizeof(tab[0]); i++) /* tab is therefore a
      pointer type parameter! */
  ...
}
```

Tolerated example

The following example makes good use of the alignment directive and does not use an expression as a parameter of the call to the sizeof operator.

```
#pragma pack(push, 1) /* 1 byte alignment - NOT STANDARD - */
uint8_t tab[LEN];
typedef struct s_example
{
  uint32_t  ui_field1;
  uint8_t   ui_field2;
} t_exemple;
#pragma pack(pop) /* return default alignment */

int32_t i;
size_t isize;
```

```
i = 5;
isize = sizeof(int32_t);
i = 1234;
isize = sizeof(t_example); /* the value returned by sizeof is 5 since the structure
     was declared with a 1 byte alignment */
void crawl_tab(uint8_t tab[], size_t n) /* known array size */
{
  for (size_t i = 0; i < n; i++) /* tab is therefore a pointer type parameter! */
  ...
}
```

15.2.1 References

[Cert] Rec. EXP09-C Use sizeof to determine the size of a type of a variable.
[Cert] Rule EXP44-C Do not rely on side effects in operand to sizeof, _Alignof, or _Generic.
[Cert] Rec. ARR01-C Do not apply the sizeof operator to a pointer when taking the size of an array.
[Misra2012] Rule 13.6 The operand of the sizeof operator shall not contain any expression which has potential side effects.
[IsoSecu] Taking the size of a pointer to determine the size of the pointed-to type [sizeofptr].
[Cwe] CWE-131 Incorrect Calculation of Buffer Size.
[Cwe] CWE-467 Use of `sizeof()` on a pointer type.
[Cwe] CWE-805 Buffer access with incorrect length value.

15.3 Mandatory verification of the success of a memory allocation

When a memory allocation is made, it may fail if there is no more free memory in the system. Failure to test the pointer returned by the allocation function will cause the program to crash the first time the pointer is used.

Usually, if allocation fails, the memory allocation function returns a `NULL` pointer. It is therefore necessary to check that the pointer returned by the allocation function is not `NULL`.

If the allocation function behaves differently on an allocation error, refer to the documentation of the function for how to handle the error appropriately.

RULE 163

RULE — Mandatory verification of the success of a memory allocation

The success of a memory allocation must always be checked.

This rule is a special case of section 13.7 but with special attention paid to the handling of memory allocation errors.

Bad example

In the following example, the memory allocation success check is missing.

```
point_t *p_point;
p_point = (point_t *)malloc(sizeof(point_t));
/* no check on function return */
p_point->x = 0.0f;
p_point->y = 0.0f;
```

Good example

In the following code, the success of the memory allocation is checked before the pointer is used.

```
point_t  *p_point = NULL;

p_point = (point_T *)malloc(sizeof(point_t));

if (NULL != p_point) {
    p_point->x = 0.0f;
    p_point->y = 0.0f;
} else {
    /* error handling */
}
```

15.4 Isolation of sensitive data

When sensitive data is loaded into the memory (e.g. encryption keys), it remains in memory after the program has finished accessing it. Another program can access our program's memory via *auxiliary channels* [14].

It is therefore necessary to associate memory areas with their use: data representing different values is stored in separate memory spaces. If a shared memory area is recycled, make sure that it is erased before being reused.

All memory areas that contain sensitive data must be explicitly deleted once the program no longer needs to access this data.

Warning

Clearing buffers so that data does not remain on the stack via a `memset`, for example, may be considered unnecessary by the compiler and the associated calls can therefore be deleted in order to optimise the code. The developer should be aware of this risk and consult the documentation of the compiler used in order to ensure that the calls in question are properly stored.

RULE 164

RULE — Sensitive data must be isolated

Check the correct use of a memory area storing sensitive data, *i.e.* minimise memory exposure, minimise copying and delete the area(s) that contained the sensitive data as soon as possible.

[14] There are many illustrations of this: Meltdown, Spectre, ZombieLoad, *etc.*

Bad example

In the example below, the same buffer is used to store the key and then the initialisation vector.

```
#define WORK_SIZE 32U

#include <stdlib.h>

void process(const uint8_t *key, uint16_t key_size, const uint8_t *init,
             uint16_t init_size) {
  uint8_t buffer[WORK_SIZE];

  if ((NULL == key) || (NULL == init)) {
    /* error handling */
    ...
  }

  memcpy(buffer, key, min(key_size, WORK_SIZE));
  print_key(buffer);

  /* buffer contains 4 bytes of the initialisation vector and the last 12 bytes of
      the key */
  memcpy(buffer, init, min(init_size, WORK_SIZE));
  printIV(buffer);
  ...
  /* no secure deletion */
}
```

Good example

The following example shows a partition between the key and the initialisation vector.

```
#define KEY_SIZE 16U
#define IV_SIZE 4U

#include <stdlib.h>

void process(const uint8_t *key, uint16_t key_size, const uint8_t *init,
             uint16_t init_size) {
  uint8_t my_key[KEY_SIZE];
  uint8_t iv[IV_SIZE];

  if ((NULL == key) || (NULL == init)) {
    /* error handling */
    ...
  }

  memcpy(my_key, key, min(key_size, KEY_SIZE));
  print_key(my_key);

  memcpy(iv, init, min(init_size, IV_SIZE));
  printIV(iv);
  ...
  /* deletion of the buffers, so that the data does not remain on the stack
  ATTENTION: the compiler can optimise and delete these calls, which may be
      considered unnecessary.
  It is therefore necessary to consult the compiler documentation in order for the
      calls to be retained. */
  memset(my_key, 0, KEY_SIZE);
  memset(iv, 0, IV_SIZE);
}
```

15.4.1 References

[Cert] Rec. MSC18-C Be careful while handling sensitive data, such as passwords, in program code.

16
Error management

16.1 Correct use of errno

The errno variable, activated by the header file, <errno.h>, has type int, and different functions in the standard library change its value with a positive value in case of error. It is therefore important to initialise errno before any function call from the standard library that changes its value, and it is therefore also necessary to consult its value at the end of the execution of such functions.

RULE 165

RULE — Initialise and view the value of errno before and after any execution of a standard library function that changes its value

Bad example

```
#include <stdlib.h>
void try1 (const unsigned char * len)
{
  unsigned long res;
  res = strtoul(len,NULL,5); /* conversion character string to unsigned long */
  /* the function strtoul writes in errno */
  if (res == ULONG_MAX)
  {
    /* problem management */
  }
...
}
```

Good example

```
#include <stdlib.h>
#include <errno.h>
void try1 (const unsigned char * len)
{
  unsigned long res;
  errno = 0; /* init errno */
  res = strtoul(len,NULL,5); /* conversion character string to unsigned long */
  /* strtoul written in errno */
  if (res == ULONG_MAX && errno != 0) /* errno reading */
  {
    /* problem management */
  }
...
}
```

16.1.1 References

[Cert] Rule ERR30-C Set errno to zero before calling a library function known to set errno and check errno only after the function returns a value indicated failure.
[Cert] Rule ERR32-C Do not rely on indeterminate values of errno.
[IsoSecu] Incorrectly setting and using errno [inverrno].
[Cwe] CWE-456 Missing Initialisation of a variable.

16.2 Systematic consideration of errors returned by standard library functions

Most of the standard library functions return values to indicate the correct operation of the function, but also an error when the function is executed. Failure to test the return value may lead to the use of erroneous data produced by the function.

RULE 166

RULE — All errors returned by standard library functions must be handled

Any function return must be read in order to set up the appropriate processing following the execution of the function.

This rule is a specific case of section 13.7 but with special attention paid to the error management of standard library functions.

Bad example

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  FILE *fp = fopen("myfile.txt", "w"); /* returned value not read */
  fputs("hello\n", fp);
  ...
}
```

Good example

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  FILE *fp = fopen("myfile.txt", "w");
  if (fp != NULL) {
    fputs("hello\n", fp);
  ...
  }
}
```

16.2.1 References

[Misra2012] Dir. 4.7: If a function returns error information, then that error information shall be tested.
[Cert] EXP12-C Do not ignore values returned by functions.
[Cert] ERR33-C Detect and handle standard library errors.
[Cert] FIO37-C Do not assume that fgets() or fgetsw() returns a nonempty string when successful.
[Cwe] CWE-241 Improper Handling of Unexpected Data Type.
[Cwe] CWE-252 Unchecked Return Value.
[Cwe] CWE-253 Incorrect Check of Function Return Value.
[Cwe] CWE-391 Unchecked Error Condition.
[IsoSecu] Failing to detect and handle library errors [liberr].
[IsoSecu] Forming invalid pointers by library function [libptr].

16.3 Documentation and structuring of error codes

Incomplete documentation of the prototype of a function can lead to programming errors, especially in error management, if all return codes are not indicated with their meaning.

A documentation template for error codes must be defined. This should contain, for each return code, the associated error and, in the event that several error codes may occur at the same time, the priority between these codes must be specified for error management.

RULE 167

RULE – Error code documentation

All error codes returned by a function must be documented. If several error codes can be returned at the same time by the function, the documentation must define the priority for handling these codes.

Error codes must contain information. Without structuring, the information indicated by the return code is often insufficient. The structuring of return codes via masks is one possibility. The return codes must also be structured in such a way that it can be determined whether the value comes from a normal execution of the function, or on the contrary whether an external element has interfered (buffer overflow, *etc.*).

RECO 168

RECOMMENDATION – Structuring of return codes

The return codes must be structured in such a way that information on the progress of the function called can be obtained easily:

- error;
- error type;
- alarm;
- alarm type;
- ok;

- *etc.*

16.3.1 References

[Cert] Rec. ERR00-C Adopt and implement a consistent and comprehensive error-handling policy.

16.4 Return code of a C program depending on whether it executed successfully

The management of the return code of a program is not identical from one operating system to another or from one shell to another. This can therefore cause problems of code portability. From one operating system to another, or from one shell to another, the authorised value range for the return code of a program varies:

- on Windows, the shell `cmd.exe` authorises signed 32-bit integers (value accessible in the `ERRORLEVEL` variable);
- on Linux, the shell authorises a value between 0 and 255 (even if some codes are reserved for signals; the value is accessible via the variable `$?`).

The use of a return code between 0 and 127 protects against the risks of modification (by type conversion) or misinterpretation of the return code of a program:

- values between 0 and 127 can be coded over 7 bits;
- they have the same coding whether the integer type is signed or unsigned.

RULE 169

RULE – Return code of a C program according to the result of its execution

The return code of a C program must have a meaning in order to indicate that the program has run correctly or that an error has occurred:

- the value of the return code must be between 0 and 127;
- the value 0 indicates that the program was executed without errors;
- the value 2 is generally used in Unix to indicate an error in the arguments passed as parameters for the program.

The meaning of the program's return codes must be indicated in its documentation.

Bad example

The following code presents a portability problem between Windows and Linux. The value -1 is in fact converted to 255 on Linux with the bash shell.

```
int main(int argc, char* argv[]) {
  if (argc != 2) {
    /* incorrect number of arguments */
    return -1; /* the return code will not be interpreted correctly on Linux */
  }
  ...
  return 0;
}
```

Good example

In the following example, the return codes do not pose a portability problem.

```
#define RESULT_OK (0U)
#define ARG_ERROR (2U)

int main(int argc, char* argv[]) {
  if (argc != 2) {
    /* incorrect number of arguments */
    return ARG_ERROR;
  }
  ...
  return RESULT_OK;
}
```

16.5 Ending of a C program following an error

When multiple exit points are defined in a C program, this makes it difficult to set up tests for that program or the libraries used by that program. Error management must be carried out using error codes. If a critical error is encountered, the program should not be terminated by calling the `abort()` function or the `_Exit()` function (C99) in the code where the error was detected. In fact, these two functions do not terminate the program *cleanly*, *i.e.* they bypass the normal termination routines (closing files, deleting temporary files, writing data, *etc.*). The error must be traced by means of an error code to the main function `main()`, which will then terminate the program.

RECO 170

RECOMMENDATION — Give preference to error returns via return codes in the main function

A C program must have a minimum `main()` function. Error returns are made by a dedicated (and therefore documented) code return of this function.

RULE 171

RULE — Do not use the `abort()` or `_Exit()` functions

The `exit()` function results in a normal termination of the program and is not dependent on the implementation. This exit from the program can be used, but excessively frequent use of this function in the program can make it difficult to understand.

RECO 172

RECOMMENDATION — Limit calls to `exit()`

Calls to the `exit()` function must be commented and not overused. The developer should replace them wherever possible with an error code return in the main function.

Finally, the `setjmp()` and `longjump()` functions defined in the `setjmp.h` library mainly used for exception handling in C can easily lead to undefined behaviour and should therefore not be used. In particular, their use creates problems with signal management.

RULE 173

RULE – Do not use the `setjmp()` and `longjump()` functions

Bad example

```
#include <stdlib.h>
#include <stdio.h>
int read_file(void)
{
  FILE *f = fopen("C:\\myfile.txt", "w");
  if (NULL == f)
  {
    /* problem when opening file */
    _Exit(12); /* not authorized */
  }
  fprintf(f, "%s", "blablabla");
  ...
  abort(); /* not authorized */
  ...
  return 0;
}
int main(void)
{
  int val = read_file();
  ...
  return 1;
}
```

Good example

```
#include <stdlib.h>
#include <stdio.h>
int read_file(void)
{
  FILE *f = fopen("C:\\myfile.txt", "w");
  if (NULL == f)
  {
    /* problem when opening file */
    return 12; /* error code documented for this problem */
  }
  fprintf(f, "%s", "blablabla");
  ...
  return 10; /* other documented error code */
  ...
  return 0; /* no problem */
}
int main(void)
{
  int val = read_file();
  if (val == 0)
  { /* no problem in the function */
    ...
    return 0;
  }
  else
  { /* error handling according to the error code */
    ...
    return 1;
  }
}
```

16.5.1 References

[Cert] Rule SIG30-C Call only asynchronous functions with signal handlers.

[Cert] ERR00-C Adopt and implement a consistent and comprehensive error-handling policy.

[Cert] ERR04-C Choose an appropriate termination strategy.

[Cert] ERR06-C Understand termination behavior of assert() and abort().

[IsoSecu] Calling functions in the C Standard Library other than `abort`, `_Exit` and `signal` from within a signal handler [asyncsig].

[IsoSecu] Calling `signal` from interruptible signal handlers [sigcall].

[Cwe] CWE-479 Signal Handler Use of a Non-reentrant Function.

17
Standard library

17.1 Prohibited standard library header files

Several header files in the standard library only introduce functions that contradict the rules or recommendations of this guide:

- `setjmp.h`;
- `stdarg.h`.

Therefore, these header files must not be used as they violate several of the above rules.

The library `<stdarg.h>`, for example, introduced in the C90 standard, declares a type and defines 3 macros: `va_start`, `va_arg`, `va_end`. A new macro (`va_copy`) is introduced with C99. The purpose of this library is to allow the definition of functions with a variable number and type of arguments. In addition, the use of these features can, in many cases, lead to undefined behaviour.

Inconsistent typing in the call of a variadic function can lead to an unexpected termination of the function or even undefined behaviour.

RULE 174

RULE – Do not use the `setjmp.h` and `stdarg.h` standard libraries

17.1.1 References

[Misra2012] Rule 17.1 The features of <stdarg.h> shall not be used.
[Cert] Rec. DCL10-C Maintain the contract between the writer and caller of variadic functions.
[Cert] Rec. DCL11-C Understand the type issues associated with variadic functions.
[Cert] Rule MSC39-C Do not call va_arg() on a va_list that has an indeterminate value.
[Misra2012] 21.4 The standard header file <setjmp.h> shall not be used.
[Cert] MSC22-C Use the setjmp(), longjmp() facility securely.
[Cert] ERR04-C Choose an appropriate termination strategy.
[Cert] ERR05-C Application independent code should provide error detection without dictating error handling.

17.2 Not recommended standard libraries

Use of the following libraries should be limited and retained only if necessary: `float.h`, `complex.h`, `fenv.h` and `math.h`.

RECO 175

RECOMMENDATION — Limit the use of standard libraries handling floating-point numbers

The standard libraries `float.h`, `complex.h`, `fenv.h` and `math.h` should only be used if absolutely necessary as in the case of digital processing.

17.2.1 References

[Misra2012] 21.11 The standard header file <tgmath.h> shall not be used.
[Misra2012] 21.12 The standard header file <fenv.h> shall not be used.
[Cert] FLP32-C Prevent or detect domain and range errors in math functions.
[Cert] FLP03-C Detect and handle floating point errors.
[Cwe] CWE-682 Incorrect calculation.

17.3 Prohibited standard library functions

Other libraries contain dangerous functions such as the `atoi()`, `atol()`, `atof()` and `atoll()` functions of `stdlib.h` which lead to undefined behaviour if the resulting value cannot be represented. The `strto*()` functions are to be preferred as they have the same action, but without the risk of undefined behaviour.

RULE 176

RULE — Do not use the functions `atoi()`, `atol()`, `atof()` and `atoll()` from the library `stdlib.h`

Equivalent functions `strto*()` have to be used.

The `rand()` function of the standard library for pseudo-random number generation does not give any guarantee as to the quality of the generated randomness.

RULE 177

RULE — Do not use the `rand()` function of the standard library

17.3.1 References

[Misra2012] The `atof`, `atoi`, `atol` and `atoll` functions shall not be used.
[Cert] ERR07-C Prefer functions that support error checking over equivalent functions that don't.
[Cert] Rec MSC25-C Do not use insecure or weak cryptographic algorithms.

[Cert] Rule MSC30-C Do not use the rand() function for generating pseudorandom numbers.
[Cwe] CWE-327 Use of a Broken or Risky Cryptographic Algorithm.
[Cwe] CWE-338 Use of Cryptographically Weak Pseudo-random Number Generator (PRNG).
[Cwe] CWE-676 Use of potentially dangerous functions.

17.4 Choice between different versions of standard library functions

When a function from the standard library offers a “less dangerous” version — in the sense that it adds extra security —, this version should be favored.

Warning

We refer to “less dangerous” versions because such functions for instance add a limit to the size of an input parameter, but they can still result in undefined or unspecified behaviours.

Information

In later versions of the C language (especially for C11), new and genuinely more secure versions are proposed such as strcpy_s().

String handling functions of the type strxx will be replaced by the equivalent functions strnxx when it is possible to limit the number of characters to be handled.

RULE 178

RULE — Use the “more secure” versions for standard library functions

When different versions of functions from the standard library exist, the “more secure” version must be used.

Likewise, all obsolete or outdated functions must not be used. The best known example is that of the gets() function, deprecated in the third technical patch of C99 [AnsiC99] and which was removed from subsequent standards.

RULE 179

RULE — Do not use obsolete library functions or those which become obsolete in subsequent standards

RULE 180

RULE — Do not use library functions that handle buffers without taking the buffer size as an argument

17.4.1 References

[Cert] Rec. PRE09-C Do not replace secure functions with deprecated or obsolescent functions.
[Cert] Rec. MSC24-C Do not use deprecated or obsolescent functions.
[Cwe] CWE-20 Insufficient input validation.
[Cwe] CWE-120 Buffer Copy without checking Size of Input('Classic Buffer Overflow').
[Cwe] CWE-676 Use of potentially dangerous function.
[Cwe] CWE-684 Failure to provide specified functionality.

18
Analysis, evaluation of the code

18.1 Proofreading of the code

It is good for any developer, although not mandatory, to have their code read through at least once by a dedicated proofreader or another developer to check the maintainability and clarity of their code.

GOOD PRACTICE 181

GOOD PRACTICE — All code should be proofread

18.2 Indentation of long expressions

For long expressions, in the absence of adequate indentation, it is very difficult to understand the code and the intention of the developer. The use of space characters for indentation of expressions and statements allows more flexibility for indentation than the use of the tab character.

RECO 182

RECOMMENDATION — Indentation of long expressions

When a statement or expression is spread over several lines, indentation is essential in order to facilitate understanding of the code. compréhension du code.

Bad example

The code in the following example should be re-indented in order to make it easier to understand.

```
if((OPT_1 == opt)
 || ((c >= a) && (0xdeafbeef == b)
 && (NULL == p_point)))
{
  /* processing */
}
if(0 != a_function_name_really_to_extend_to_be_as_explicit_as_possible(
    A_CONSTANT_ALWAYS_WITH_A_VERY_EXPLICIT_NAME,
    A_SECOND_CONSTANT_ALWAYS_WITH_A_NAME_TO_EXTEND, 5, 50))
{
  /* processing */
}
```

Good example

The following code shows a correct indentation of a conditional over several lines.

```
if((OPT_1 == opt)
  || ((c >= a)
  && (0xdeafbeef == b)
  && (NULL == p_point)
  ))
  {
    /* processing */
  }
if(0 != a_function_name_really_to_extend_to_be_as_explicit_as_possible(
      A_CONSTANT,A_CONSTANT_ALWAYS_WITH_A_VERY_EXPLICIT_NAME,
      A_SECOND_CONSTANT_ALWAYS_WITH_A_NAME_TO_EXTEND,
      5,
      50))
  {
    /* processing */
  }
```

18.3 Identifying and removing any dead or unreachable code

The presence of dead code or unreachable code hinders the proofreading and understanding of the code.

Unreachable code

Code is considered unreachable if there is no input that allows this point of the program to be reached (statements in an always false conditional, statements located after a `return` statement, *etc.*).

Dead code

"Dead code" is understood to mean code for which the execution has no effect (no modification of variables, no impact on the control flow, *etc.*).

Furthermore, from a security point of view, dead or unreachable code can be used during a bypass of the execution stream. This unreachable code may be debugging code, disabling security checks.

RULE 183

RULE — Identify and remove any dead code

RULE 184

RULE — The code must have no unreachable code other than defensive code and interface code

There must never be any unreachable code, except for defensive code or interface code, and in both cases it must be specified as a comment.

18.3.1 References

[Misra2012] Rule 2.1 A project shall not contain *unreachable* code.
[Misra2012] Rule 2.2 There shall be no dead code.
[Cwe] CWE-561 Dead code.
[Cwe] CWE-563 Assignment to Variable without use.
[Cwe] CWE-570 Expression is always False.
[Cwe] CWE-571 Expression is always True.

18.4 Tool-based evaluation of the source code to limit the risk of execution errors

Despite the application of coding conventions, good programming practices and test execution, errors frequently remain in software. Some of these residual errors can be discovered using code analysis tools. The code analyser must be used as development progresses, which limits the impact of the modifications and corrections made to the code but also the complexity of these modifications and corrections.

When running the tests, a dynamic analysis can be performed to identify memory leaks. Code coverage should also be measured in order to identify parts of the software that have not been tested.

RECO 185

RECOMMENDATION — Tool-based evaluation of the source code to limit the risk of execution errors

The source code of the software should be analysed using at least one code analysis tool. The results produced by the analysis tool should be studied by the developer and corrections must be made in relation to the problems discovered.

18.4.1 References

[Misra2012] Rule 1.3 There shall be no occurrence of undefined or critical unspecified behaviour.
[Misra2012] Dir. 4.1: Run-time failures shall be minimized.

18.5 Limiting cyclomatic complexity

Cyclomatic complexity

Cyclomatic complexity is a metric that measures the structural complexity of a computer program (module, function). It corresponds to the number of existing paths.

It is often observed that the greater the cyclomatic complexity, the more difficult the computer program is to test and maintain. High cyclomatic complexity indicates a high probability of introducing errors during development or maintenance of the program.

In the event of significant cyclomatic complexity, the code should be reorganised in order to simplify it. This can be done, for example, by writing additional functions.

RECO 186

RECOMMENDATION – Limitation of cyclomatic complexity

The cyclomatic complexity of a function should be limited as far as possible.

18.6 Limiting the length of functions

In line with the previous section, each function of a program must have a clear corresponding action. Too often, C functions actually perform several actions/processes at once, which complicates the reading of the code, its updating and maintenance. A function that is too long, in terms of the number of lines of code, is often a sign of a function that is too complex with multiple actions, and which could therefore be split into several sub-functions. In such cases, the function code should be reorganised in order to simplify it and reorganise it into different functions of smaller sizes, associated with precise processing.

RECO 187

RECOMMENDATION – Limitation of the length and complexity of a function

A function should ideally be associated with a single process and should therefore correspond to a reasonable number of lines of code.

18.7 Do not use C++ keywords

C and C++ are two **different** programming languages, although they have many similarities, and C++ incorporates most of the features of the C language.

A developer may inadvertently use C++ keywords (for example: `class`, `new`, `private`, `public`, `delete`, *etc.*) within a C code, whether to name a function, variable or something else. However, this hinders proofreading of code, and risks confusing the analysis tools. Moreover, this can hamper maintenance and can cause compilation problems if the compiler also includes C++. A search for these keywords in the sources of a C program can be easily automated. When one of the keywords is found, the name of the variable, type or function must be changed.

Appendix C provides the list of C++ keywords.

RULE 188

RULE – Do not use C++ keywords

No C++ keywords must be used in the source code of a C program.

Bad example

In the code below, the names of the `new` and `delete` functions should be changed to `new_point` and `delete_point` for example.

```
/* point.h */

typedef struct
{
  float x;
  float y;
} point_t;

point_t *new();

void delete(point_t *p);
```

Good example

In the following example, no C++ keywords are used.

```
/* point.h */
typedef struct
{
  float x;
  float y;
} point_t;
point_t *new_point();
void delete_point(point_t *p);
```

19

Miscellaneous

19.1 Comment format

Comments accepted according to C90 can only take the form:

```
/* comments that can be over several lines */
```

In C99, the notation of comments on a line is extended with the following format:

```
// comments on a single line
```

The character sequences /* and // are prohibited in all comments, and the line splicing character \ is prohibited in a comment introduced by // because it leads to undefined behaviour.

RULE
189

RULE – Prohibited character sequences in comments

The /* and // sequences are prohibited in all comments. And a comment on a line introduced by // must not contain a line splicing character \.

19.1.1 References

[Misra2012] Rule 3.1 The characters sequences /* and // shall not be used within a comment.
[Misra2012] Rule 3.2 Line-splicing shall not be used in // comments.

19.2 Implementation of a “canary” mechanism

Already introduced in subsection 5.3.5, “canaries” provide a protection against some programming errors that could for instance enable control flow hijacking by overwriting a function return address saved on the stack.

If the toolchain does not support automatic insertion of canaries, such a mechanism must be implemented by the developer himself. This can be achieved by passing an additional argument to each critical function and verifying its value at the beginning and at the end of this function, as illustrated in the code sample hereinbelow.

RULE 190

RULE — Manually implement a “canary” mechanism when not already supported by the toolchain

This mechanism must at least be applied to critical program functions.

When this is not feasible, it is still possible to undertake a thorough analysis of the source code in order to, for example, guarantee that no local arrays are used, to avoid any control flow hijacking due to a stack buffer overflow.

Warning

Preference should be given to the “automatic” use of canaries by means of the toolchain. Indeed, developing a canary mechanism remains a complicated task, often prone to programming errors or even vulnerabilities.

Good example

The keyword `volatile` is used to prevent possible optimisations of the compiler for access to the values of the `canary` and `canaryRef` variables. In fact, it is necessary to systematically go and read the `canary` and `canaryRef` values in memory.

```c
typedef volatile uint32_t fid_t;

#ifdef ACTIVATE_CANARIES
static inline void verifcanari(fid_t canari, fid_t canariRef) {
  uint8_t res = !!(canari != canariRef);

  if (0 != res)
  {
    /* context-specific processing */
  }
}
#else   /* ifdef ACTIVATE_CANARIES */
static inline void verifcanari(fid_t canari, fid_t canariRef) { }
#endif  /* ifdef ACTIVATE_CANARIES */
void foo(fid_t canari) {
  /* checking of the canary parameter at the beginning of the function */
  verifcanari(canari, FID_FOO);

  /* body of the function... */

  /* checking of the canary parameter at the end of the function */
  verifcanari(canari, FID_FOO);
}
```

19.3 Assertions of development and assertions of integrity

Two types of assertions can be distinguished in software:

- assertions for the purpose of development. These are intended to be removed from the software once the qualification phase is over (for example, to check that a pointer parameter is not null);
- assertions designed to check the integrity of the software during execution: these are intended to ensure that the software runs normally and to detect a hardware failure or an attempt to modify it externally (e.g. a fault attack).

A software integrity assertion should not be written using the macro `assert()`. Indeed, this macro is deleted from the code generated on compilation in release mode. Furthermore, these assertions should only be used for debugging purposes and are in particular not recommended for verification purposes, especially due to initialisations activated in debug mode which will no longer be present outside of debug mode.

It may be that a code checking the integrity of a software program is detected as code unreachable by the compiler or a static analysis tool (in fact, the code can check a set of conditions that cannot occur during normal program execution). The purpose of this code must be clearly documented, and it must be ensured that compiler optimisations do not result in the deletion of this code in the generated binary.

RULE
191

RULE – No development assertion on a code in production

Development assertions must not be present in production.

RECO
192

RECOMMENDATION – Management of integrity assertions should include emergency data deletion

Integrity assertions should appear in production. If an integrity assertion is triggered, the processing code should result in the emergency deletion of sensitive data.

19.4 Last line of a non-empty file must end with a line break

The absence of a line break at the end of a non-empty file leads to undefined behaviour according to the C90 and C99 standards.

Warning

The vast majority of publishers, particularly in a Linux environment, automatically and invisibly add this line break when closing files.

In addition, all preprocessor directives and comments must be closed.

RULE
193

RULE – All non-empty files must end with a line break and the preprocessor directives and comments must be closed

A non-empty file must not end in the middle of a comment or preprocessor directive.

Appendix A
Acronyms

ANSI American National Standards Institute

API Application Programming Interface

ASLR Address space layout randomization

FAM Flexible Array Member

IDE Integrated Development Environment

ISO International Standards Organization

MISRA Motor Industry Software Reliability Association

VLA Variable Length Array

Appendix B
Further information on gcc and Clang options

Information given in this appendix originates from [GccRef] and [ClangRef], for GCC 11 and CLANG 13 respectively.

B.1 Definition of the C language standard in use

Option `-std` allows for specifying the version of the C standard — or of the corresponding GNU dialect — used by the compiler. Without this option, GNU dialect of ISO C17 is selected by default.

Information

Option `-ansi` is equivalent to option `-std=c90`, which is itself equivalent to `-std=c89` and `-std=iso9899:1990`.

Information

Option `-std=iso9899:199409` corresponds to ISO C90 as modified in amendment 1 in 1995.

Information

Option `-std=iso9899:1999` is equivalent to `-std=c99`.

B.2 Additional warnings

The following options are neither included in `-Wall` nor `-Wextra` nor `-Wpedantic` and were not mentioned in chapter 5, but may nonetheless prove useful [15]:

- `-Wbad-function-cast`
- `-Wcast-align`
- `-Wcast-qual` (warns whenever a pointer is cast so as to remove — or introduce in an unsafe way — a type qualifier like `const`)

[15] Only options whose names are deemed not meaningful enough are explained. In any case, the reader is encouraged to refer to GCC and CLANG compilers manuals for more detailed explanations.

- `-Wconversion` (warns for implicit conversions that may alter a value, including conversions between signed and unsigned)
- `-Wfloat-equal`
- `-Wnull-dereference`
- `-Wshadow` (warns whenever a local variable or type declaration reuses an identifier that is already bound to another variable, parameter or type)
- `-Wstack-protector` (warns about functions that are not instrumented with a stack canary)
- `-Wstrict-prototypes`
- `-Wswitch-enum` (warns whenever the controlling expression of a `switch` statement is of an enumerated type but lacks a `case` for one or more of the named constants defined with this type)
- `-Wmissing-prototypes`
- `-Wundef`
- `-Wvla`

The following options are specific to GCC:

- `-Wduplicated-branches`
- `-Wduplicated-cond`
- `-Wformat-signedness`
- `-Wjump-misses-init`
- `-Wlogical-op` (warns about suspicious uses of logical operators)
- `-Wnested-externs` (warns on declarations that use the `extern` storage class specifier within a function)
- `-Wnormalized` (warns about any identifier that is not in normalized form)
- `-Wold-style-definition`
- `-Wshift-negative-value`
- `-Wshift-overflow=2`
- `-Wstrict-overflow=3` (warns about a number of cases where the compiler optimizes based on the assumption that signed overflow does not occur)
- `-Wsuggest-attribute=format` (warns for cases where adding a `format` GCC attribute may be beneficial)
- `-Wsuggest-attribute=malloc` (warns for cases where adding a `malloc` GCC attribute may be beneficial)
- `-Wswitch-default` (warns whenever a `switch` statement does not have a default case)
- `-Wtraditional-conversion` (warns if a prototype causes a type conversion that is different from what would happen to the same argument in the absence of a prototype)

- `-Wtrampolines` (warns if trampolines, mentioned in a footnote in subsection 5.3.5, are generated)
- `-Wwrite-strings` (adds type qualifier `const` to constant strings so that copying the address of one into a pointer to a non-`const`-qualified type produces a warning; this helps the developer to find at compile-time code that tries to write into a string constant — provided that the `const` keyword has indeed been used in declarations and prototypes, otherwise this warning becomes unhelpfully very noisy)

The following options are specific to CLANG:

- `-Warray-bounds-pointer-arithmetic`
- `-Wassign-enum` (warns whenever an integer constant assigned to a variable of an enumerated type does not belong to the range of values defined for this type)
- `-Wcast-function-type`
- `-Wcomma`
- `-Wcovered-switch-default`
- `-Wduplicate-enum`
- `-Widiomatic-parentheses` (warns whenever an assignment expression is used as a condition without being enclosed in parentheses)
- `-Wloop-analysis`
- `-Wformat-non-iso`
- `-Wformat-pedantic`
- `-Wformat-type-confusion`
- `-Wfour-char-constants`
- `-Wimplicit-fallthrough`
- `-Wpointer-arith`
- `-Wpragmas`
- `-Wreserved-identifier`
- `-Wshift-sign-overflow`
- `-Wsigned-enum-bitfield`
- `-Wstatic-in-inline`
- `-Wtautological-constant-in-range-compare`
- `-Wthread-safety`
- `-Wunreachable-code`
- `-Wunreachable-code-aggressive`

- -Wunused-macros
- -Wused-but-marked-unused
- -Wvariadic-macros
- -Wzero-as-null-pointer-constant

Information

With CLANG, option -Wall automatically enables option -Wmost, which itself enables many additional warnings. Therefore, the options that correspond to the latter are not listed above.

B.3 Clang and the -Weverything option

CLANG features a -Weverything option[16] which enables *all* of the warnings supported by CLANG without exception.

Using -Weverything may be interesting to discover new warnings supported by the compiler or in case of highest level of requirement on a given code base. It should not be used systematically though, since it might for instance cause trouble for project builds following a tool update.

[16] Not to be confused with -Wall, -Wextra and -Wmost.

Appendix C
C++ reserved words

The following list contains reserved words from C++ that do not belong to the C language. Words suffixed with an asterisk are reserved words added in C++11. Additional semantics have been added to the reserved word delete in C++11 when declaring a class.

alignas *
alignof *
and
and_eq
asm
thread_local *
bitand
bitor
char16_t *
char32_t *
catch
class
compl
constexpr *
const_cast
decltype *
delete *
dynamic_cast
explicit
export
final*
friend
mutable
namespace
new
noexcept *
not_eq
nullptr *
operator
or
or_eq
override *
private
protected
public
reinterpret_cast
static_assert *
static_cast
template
this
throw
try
typeid
typename
using
virtual
xor
xor_eq

Appendix D
Operator priority

The order adopted is in descending order of priority. Operators present in the same cell of the table have the same priority level, even if they are located on a different row of the cell.

L. to R. stands for “left-to-right associativity”, and R. to L. stands for “right-to-left associativity”.

Category	Operator	Name	Associativity
Reference	() [] -> .	Function call Access to an element in an array Access to a field of a given address in a structure Access to a field of a structure	L. to R.
Unary	+ - ++ – ! ~ & (cast) sizeof	Identity Opposite Increment Decrement Logic negation Inversion of all bits Pointer referencing Pointer dereferencing Type conversion Size of an object	R. to L.
Arithmetic	* / % + -	Product Division Modulo Sum of two numbers or a pointer and a number Substraction of two numbers or two pointers	L. to R.
Shift	<< >>	Binary shift to the left Binary shift to the right	L. to R.
Comparison	< <= > >= == !=	Strictly less than, less than or equal to Strictly greater than, greater than or equal to Equal à Different from	L. to R.
Bit processing	& ^ |	Bitwise And Bitwise exclusive Or Bitwise Or	L. to R.
Logic	&& ||	Logic And Logic Or	L. to R.

Conditional	?:	Ternary conditional operator	R. to L.
Assignment	= += -= *= /= %= &= ^= |= <<= >>=	Assignment Increment, decrement, product, division then assignment Modulo, logic operation, shift then assignment	R. to L.
Sequence	,	Argument or expression separator	L. to R.

Appendix E
Example of development conventions

At the beginning of an IT project, the development team should always agree on the coding conventions to be applied. The aim is to produce a coherent source code. Furthermore, the right choice of conventions helps to reduce programming errors.

> **Information**
>
> The following points are an **example** of coding conventions. Some choices are arbitrary and open for discussion. This example of conventions can be used or taken as a foundation, if no development convention has been defined for the project to be produced. Different tools or advanced editors are able to automatically implement some of these coding conventions.

Where agreements have been defined in the context of the implementation of a project, the document clearly specifying these conventions must accompany the project in question.

E.1 Files encoding

The source files are encoded in UTF8 format.

The line feed character is `\n` (“line feed” in Unix format).

E.2 Code layout and indentation

E.2.1 Maximum lengths

A line of code or comment should not exceed 100 characters.

A line of documentation should not exceed 100 characters.

A file should not exceed 4000 lines (including documentation and comments).

A function should not exceed 500 lines.

E.2.2 Code indentation

The code is indented with spaces: one level of indentation corresponds to 4 space characters. The use of the tab character as an indentation character is prohibited.

The declaration of variables and their initialisation must be aligned using indentations.

A space character is systematically left between a keyword and the opening parenthesis that follows it.

The opening brace of a block is placed on a new line. The block closing brace is also placed on a new line.

A space character is left before and after each operator.

A space character is left after a comma.

The semicolon indicating the end of a statement is stuck to the last operand of the statement.

For a function call with many parameters, if it is necessary to place the parameters on several lines, these parameters are indented to be positioned at the opening parenthesis of the function call.

Good example

```
...
uint32_t processing_function(linked_list_t *p_param1, uint32_t ui32_param2,
                             const unsigned char *s_param3)
 {
   uint32_t ui32_result  = 0;
   element_t *pp_out_param4 = NULL;

   if ((NULL == p_param1) || (NULL == s_param3))
   {
     ui32_result = 0;
     goto End;
   }

   ui32_result = function_with_many_params(p_param1, ui32_param2, s_param3,
       pp_out_param4);

   if (1 == ui32_result)
   {
     ...
   }

   End:
   return ui32_result;
 }
```

E.3 Standard types

If the `stdint.h` header is present, it must be included in order to benefit from the integer types that it defines. In its absence, it is necessary to define the integer types as presented in section 7.

If the `stdbool.h` header is present, it must be included in order to benefit from the boolean type it defines. In its absence, the `bool` type must be defined as presented on the following code (header file from GCC version 4.8.2). The `_Bool` type is defined for the compilers compatible with standard C99 and subsequent standards.

```
/* Copyright (C) 1998-2013 Free Software Foundation, Inc. */
```

```
/*
 * ISO C Standard:  7.16  Boolean type and values  <stdbool.h>
 */

#ifndef _STDBOOL_H
#define _STDBOOL_H

#ifndef __cplusplus

#define bool    _Bool
#define true    1
#define false   0

#else // __cplusplus

/* Supporting <stdbool.h> in C++ is a GCC extension. */
#define _Bool   bool
#define bool    bool
#define false   false
#define true    true

#endif // __cplusplus

/* Signal that all the definitions are present. */
#define __bool_true_false_are_defined   1

#endif  // stdbool.h
```

E.4 Naming

E.4.1 Language for implementation

The language used for naming libraries, header files, source files, macros, types, variables and functions must be English. This use of English avoids mixing words in French with the keywords of the C language which are in English within the code. The entire source code produced is thus more coherent.

The language used for documentation and comments should be English from the beginning of development and for all documentation and comments.

E.4.2 Naming of source file directories

The source files must be organised in libraries. In the case of a large library, it is recommended to create a tree structure to organise the source files. The top-level directory must be named with the name of the library. Sub-directories must be named in such a way that they reflect the criteria for grouping source files.

The following example shows the organisation of directories for a library containing utility functions:

Tree structure	Comment
utils	Basic directory of the library
utils/includes	Directory containing all the header files of the library (API)
utils/collection	Directory containing the implementation of all collection type data structures (lists, stack, array, hash table, *etc.*)
utils/concurrency	Directory containing the implementation of mutex, semaphores, conditional variables
utils/threads	Directory containing the implementation of threads
…	…

E.4.3 Naming of header files and implementation files

Header files and source files must be prefixed with the name of the library to which they belong. If the library name is long, it is advisable to use an abbreviation as a prefix. This abbreviation must be chosen in such a way that it does not conflict with an already existing library (standard libraries, third party libraries, *etc.*).

The following list gives examples of header file and source file names: `utils_linked_list.h`, `utils_linked_list.c`, `utils_mutex.h`, `utils_mutex.c`, `utils_thread.h`, `utils_thread.c` …

E.4.4 Naming of macros

Preprocessor macros must have upper case names. The words making up a macro name must be separated by the underscore character. The name of a macro should not match an already existing name of another macro: for example a macro belonging to a header file of a standard library. The parameters of a macro must respect the variable naming convention.

Good example

```
#define LOG_DEBUG(sMessage) write_log_message(sMessage)
```

The name of a macro, defined to avoid the multiple inclusion of a header file, uses the name of the header file in upper case. The full stop character is replaced by an underscore character.

Good example

```
#define UTILS_LINKED_LIST_H
```

E.4.5 Naming of types

The name of a type defined using the keyword `typedef` must be written in lower case, with the suffix `_t`. The words making up the type name must be separated by the underscore character.

When defining a type for an enumeration or structure, the name following the keyword `enum` or `struct` must have the suffix `_tag`. The name of the type after the closing brace defining the type must be the same name, with `_tag` replaced by `_t`.

Good example

```
typedef enum status_tag {
  ...
} status_t;
typedef signed long sint32_t;
typedef struct linked_list_tag
{
  ...
} linked_list_t;
```

E.4.6 Naming of functions

The name of a function must be prefixed by the name (or abbreviation of the name) of the library to which it belongs. The words making up the name of the function must be separated by the underscore character. The name of a function must be written in lower case.

Good example

```
status_t utils_create_linked_list(linked_list_t **pp_list);
status_t utils_delete_linked_list(linked_list_t *pp_list);
```

E.4.7 Naming of variables

Variable identifiers will consist of words separated by the underscore character, without spaces or upper case letters. Each element of the identifier is used to specify the associated variable (type, sign, size, role, *etc.*).

The following table shows the prefixes for the variable names according to type, as well as an example for each type of variable:

Prefix	Variable type	Example
i8	Signed 8-bit integer	int8_t i8_byte = 0;
ui8	Unsigned 8-bit integer	uint8_t ui8_byte = 0U;
i16	Signed 16-bit integer	int16_t i16_option = 0;
ui16	Unsigned 16-bit integer	uint16_t ui16_port = 0U;
i32	Signed 32-bit integer	int32_t i32_value = 0L;
ui32	Unsigned 32-bit integer	uint32_t ui32_counter = 0UL;
i64	Signed 64-bit integer	int64_t i64_big_value = 0LL;
ui64	Unsigned 64-bit integer	uint64_t ui64_big_counter = 0ULL;
b	Boolean	bool b_is_set = false;
c	Character	char c_letter = ’\0’;
f	Float	float f_value = 0.0f;
d	Double	double d_precised_result = 0.0d;
sz	Type size_t	size_t sz_string_length = 0U;
e	Enumerated type variable	status_t e_status_code = STATUS_ERR;
st	Structure type variable	linked_list_t st_list;
a	Array	uint32_t a_values[10];
p	Pointer type variable	linked_list_t* p_list = NULL;

Prefix	Variable type	Example
pp	Pointer of pointer type variable	linked_list_t** pp_list = NULL;
s	String type variable	char* s_message = NULL;
ws	String type variable in unicode	wchar_t* ws_message = NULL;

E.5 Documentation

E.5.1 Format of tags for documentation

The source code documentation must be produced using the *Doxygen* tool's tag system. *Doxygen* tags must all begin with the `@` character. The *Doxygen* tool also allows the backslash character. However, in order to have uniformity for the documentation of the source code, the at-sign prefix for *Doxygen* commands is imposed.

A documentation comment begins with `/*!` and ends with `*/`.

The following points outline the minimum documentation that must be present in a header file.

E.5.2 File header title block

All header files and all source files must begin with a header title block used to identify:

- the software and / or the library to which the header/source file belongs;
- the company (and if necessary the author) and copyright associated with the file;
- the *Doxygen* `@file` tag. The `@file` tag can optionally be followed by the file name. In the absence of the file name, the file name is automatically deduced from the file in which the `@file` tag is located.

  It is essential to use the `@file` tag in the header files and the source files. In its absence the documentation on functions, global variables, type definitions and enumerations present in the file is not included in the *Doxygen* documentation produced.

If the file is part of a library, the command `@addtogroup <label> [title]` must be used. This serves to group the documentation of all the functions of a library within a module in the documentation produced. The label is the name of the group to be used in all files belonging to the library. The title is optional. It is used to name the group in the documentation.

The `@addtogroup` command must be supplemented by the `@{` and `@}` tag pair in order to delimit the elements of the file belonging to the group.

E.5.3 Documentation of a structure

The definition of a structure must be documented with a comment preceding its definition. This comment must indicate the role of the structure. Each field of the structure must be documented.

E.5.4 Documentation of an enumeration

The definition of an enumeration must be documented with a comment preceding its definition. This comment must indicate in which framework the enumeration is to be used. Each value in the enumeration must be documented.

E.5.5 Documentation of a global variable

A global variable must be documented with a comment preceding its definition. This comment must indicate the role of the variable, its initialisation value, and any invariants that must be respected.

E.5.6 Documentation of a function

The documentation of a function must precede the definition of the function prototype in the header file. The documentation of a function consists of:

- a brief comment;
- a detailed comment explaining the feature offered by the function;
- the presentation of each parameter, specifying whether it is an input, output or both input and output parameter;
- the value returned by the function. In the case of an error code, the success case(s) must be indicated, along with the different error codes that can be returned and their priority;
- a pre-condition, if any, when the function is called;
- a post-condition, if any, after calling the function;
- any additional remarks or warnings.

Good example

The following lines show the minimum documentation for a header file.

```
#ifndef UTILS_LINKED_LIST_H
#define UTILS_LINKED_LIST_H

/*!
* @file linked_list.h
* @author DEV 1
*
* @brief Linked List
*
* Function declarations for the manipulation of linked list.
*
* @addtogroup utils Library Utils
* @{
*/

  /*!
  * @brief Enumeration of status codes
  *
  * Status codes to indicate the success or the failure of functions
  */
  typedef enum status_tag {
    STATUS_SUCCESS = 0,     //!< success
    STATUS_GENERIC_ERROR,   //!< generic error
    STATUS_MEMORY_ERROR,    //!< memory allocation error
    STATUS_INVALID_PARAM    //!< invalid parameter
  } status_t;

  /*!
  * @brief Element of the linked list
  */
  typedef struct linked_list_element_tag
  {
    struct linked_list_element_tag*  pNext;     //!< next element
    struct linked_list_element_tag*  pPrevious; //!< previous element
    void*                           pData;     //!< data of the element
  } linked_list_element_t;

  /*!
  * @brief Double linked list
  *
  * Structure to define a double linked list. The type data of the list is void.
  */
  typedef struct linked_list_tag
  {
    linked_list_element_t *pHead;  //!< first element
    linked_list_element_t *pTail;  //!< last element
  } linked_list_t;

  /*!
  * @brief New linked list
  *
  * Creation of a new linked list by allocating the memory for the structure and by
       initializing the list.
  * The new list is empty.
  *
  * @param[out] ppList is the new list
  * @return #STATUS_SUCCESS the creation and the initialization are done with
      success
  * @return #STATUS_INVALID_PARAM if ppList is NULL or
  *                               if (*ppList) != NULL
  * @return #STATUS_MEMORY_ERROR fail of the memory allocation
  * @pre ppList != NULL and (*ppList) == NULL
  * @note the created list has to be deleted
  *       by calling #utils_delete_linked_list
  */
  status_t utils_create_linked_list(linked_list_t **ppList);

  /*!
  * @brief Deletion of the list
  *
  * All the elements of the list are deleted and the used memory is freed.
  * @warning The memory used by the data in the list is not freed..
  *
  * @param[in, out] ppList the list to delete.
  * @return #STATUS_SUCCESS if the deletion of the list is a success
  * @return #STATUS_INVALID_PARAM if  ppList is NULL
  *                               or if  (*ppList) is NULL
  * @pre ppList != NULL and (*ppList) != NULL
  * @post (*ppList) == NULL
  */
  status_t utils_delete_linked_list(linked_list_t **ppList);

  ...

  /*! @} */

#endif // UTILS_LINKED_LIST_H
```

Index

List of rules, recommendations and good practices

Bibliography

[float] *IEEE Standard for Floating-Point Arithmetic*.
Standard, IEEE.

[AnsiC90] *ISO/IEC 9899:1990, Programming Languages - C*.
Standard, International Organization for standardization.

[AnsiC99] *ISO/IEC 9899:1999, Programming Languages - C*.
Standard, International Organization for standardization.

[Cert] *SEI CERT C Coding Standard*.
Standard, Carnegie Mellon University.

[ClangRef] *CLANG'S Documentation*.
Public documentation, https://clang.llvm.org/docs/.

[Cwe] *CWE Common Weakness Enumeration*.
Technical report, MITRE.

[ETALAB] *Licence ouverte / Open Licence v2.0*.
Page web, Mission Etalab, avril 2017.
`https://www.etalab.gouv.fr/licence-ouverte-open-licence`.

[GccRef] *GCC: Reference Documentation*.
Public documentation, http://www.gnu.org/software/gcc/onlinedocs.

[IsoSecu] *ISO/IEC TS 17961 Information Technology - Programming languages, their envrionments and system software interfaces - C Secure Coding Rules*.
Technical report, Switzerland, Genève.

[Misra2012] *MISRA-C:2012 Guidelines for the use of the C language in critical systems*.
Guidelines, https://www.misra.org.uk/MISRAHome/MISRAC2012.

Version 1.4 - 24/03/2022 - ANSSI-PA-073

Licence ouverte / Open Licence (Étalab - v2.0)

AGENCE NATIONALE DE LA SÉCURITÉ DES SYSTÈMES D'INFORMATION

ANSSI - 51, boulevard de La Tour-Maubourg, 75700 PARIS 07 SP
www.ssi.gouv.fr / conseil.technique@ssi.gouv.fr

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

Premier ministre

SGDSN