# Introduction

MISRA (Motor Industry Software Reliability Association) is the world-leading best practice guidelines for the **safe and secure application** of both embedded control systems and standalone software.

For C, there is **MISRA C: 2012** which follows C90, C99 standards. For C++, there is **MISRA C ++: 2008** which follows C++03 standard.

# Rule Levels

NOTE: This is my self-defined rule levels. It’s not available in MISRA official document.

L1 has highest priority, while L2 has the lowest:

|  |  |  |
| --- | --- | --- |
| **Level** | **Description** | **Example** |
| **L1** |  |  |
| **L2** |  |  |
| **L3** |  |  |
| **L4** |  |  |
| **L5** |  |  |

# Terms

|  |  |
| --- | --- |
| **Term** | **Meaning** |
| POD (Plain Old Data) variable | A variable of **simple type that does not have user-defined constructors**, destructors, virtual functions, base classes, or non-static data members of reference type.  Example:  int a;  char\* b; |
| Only one use | The "use" is either an assignment (explicit initialization) or a reference. The "one" is only one usage or is referenced in only one location within the codebase. |
| Non-volatile variable | A variable that is **not declared with the volatile qualifier**. In programming, particularly in C and C++, the volatile keyword is used to tell the compiler that a variable's value may be changed by factors outside the control of the code being executed, such as hardware or concurrent threads.  When a variable is declared as non-volatile, the compiler assumes that it can optimize access to that variable and that its value will remain consistent throughout the context in which it's being accessed. This allows for more efficient code generation, as the compiler can make certain assumptions about how that variable behaves. |
| Dead code | Any executed **statement whose removal would not affect program output**. |
| Translation unit | C++ compilers treat each source file (with included headers) as separate "translation units" where each translation unit is compiled in isolation.  A .cpp file, not a .h file. |
| Well-formed *for* loop | A for loop that satisfies Rule 6–5–1 to Rule 6–5–6. |
| Ellipsis notation | The three dot  E.g. void MyPrintf (char\_t \* pFormat, ...); |
| Data class |  |

# Most-Common Rule Summary

NOTE: This is my self-defined rule summary for quick memory and code review. It’s not available in MISRA official document.

|  |  |
| --- | --- |
| **Rule** | **Simple Explanation** |
| Rule 0-1-1: A project shall not contain unreachable code | No deadcode |
| Rule 0-1-3: A project shall not contain unused variables | No unused variables |
| Rule 0-1-4: A project shall not contain non-volatile POD variables having only one use | Defining a variable means that we have to use it somewhere.  Workaround: Add (void) to the variable. |
| Rule 0-1-6: A project shall not contain instances of non-volatile variables being given values that are never subsequently used |
| Rule 0-1-11: There shall be no unused parameters (named or unnamed) in non-virtual functions | No unused params  Workaround: Add (void) to the variable. |
| Rule 0-1-12: There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it |
| Rule 2-7-1: The character sequence /\* shall not be used within a C-style comment | No block comment  Exception: Doxygen comments |
| Rule 2-7-2: Sections of code shall not be "commented out" using C-style comments | No code comment out |
| Rule 2-7-3: Sections of code should not be "commented out" using C++ comments |
| Rule 2-13-3: A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type | Add postfix U to unsigned numbers, not u. |
| Rule 2-13-4: Literal suffixes shall be upper case |
| Rule 3-1-1: It shall be possible to include any header file in multiple translation units without violating the One Definition Rule | Don’t define functions in header files |
| Rule 3-4-1: An identifier declared to be an object or type shall be defined in a block that minimizes its visibility | If a variable is used ONLY within a block, then do not declare it outside the block. |
| Rule 3-9-1: The types used for an *object*, a *function return type*, or a *function parameter* shall be token-for-token identical in all declarations and re-declarations | OK: void fn(int32\_t num); void fn(int32\_t num) { ... }  NG: void fn(int32\_t num); void fn(int32\_t number) { ... } |
| Rule 3-9-2: typedefs that indicate size and signedness should be used in place of the basic numerical types | OK: int32\_t  NG: int |
| Rule 5-0-13: The condition of an if-statement and the condition of an iteration-statement shall have type *bool* | Condition statement must be of type boolean.  OK: if (a == 1)  NG: if (a)  OK: if (ptr != NULL)  NG: if (ptr) |
| Rule 5-0-15: Array indexing shall be the only form of pointer arithmetic |  |
| Many rules about type casting | Types to be compared or assigned must be always matched and have the same signedness. If they’re not matched, then cast them. |
| Rule 5-0-21: Bitwise operators shall only be applied to operands of unsigned underlying type | Bitwise operations is ONLY for unsigned number. |
| Rule 5-2-1: Each operand of a logical && or || shall be a postfix-expression | OK: if ((a == 1) && (b == 1))  NG: if (a == 1 && b == 1) |
| Rule 5-2-4: C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used | OK: int32\_t a = static\_cast<int32\_t>(n);  NG: int32\_t a = (int32\_t)(n); |
| Rule 5-3-1: Each operand of the ! operator, the logical && or the logical || operators shall have type bool | OK: if (ptr == NULL)  NG: if (!ptr) |
| Rule 6-2-2: Floating-point expressions shall not be directly or indirectly tested for equality or inequality |  |
| Rule 6-6-4: For any iteration statement there shall be no more than one *break* or *goto* statement used for loop termination | In a loop, only ONE break is allowed. |
| Rule 6-6-5: A function shall have a single point of exit at the end of the function | In a function, only have one return statement. |
| Rule 7-1-2: A pointer or reference parameter in a function shall be declared as pointer to *const* or reference to *const* if the corresponding object is not modified | If a param is not modifed in a function, it must have a const. |
| Rule 7-3-1: The global namespace shall only contain *main*, namespace declarations and *extern "C"* declarations | Except main() and extern "C", all others must have a namespace. |
| Rule 7-3-4: The *using*-*directives* shall not be used. Use the *using-declaration* instead | OK: using namespace abc::xyz;  NG: using namespace abc; |
| Rule 7-5-4: Functions should not call themselves, either directly or indirectly | Using recurion is banned. |
| Rule 8-5-1: All variables shall have a defined value before they are used | All variables must be initialized. |
| Rule 9-5-1: Unions shall not be used | Using union is banned. |
| Rule 11-0-1: Member data in non-POD class types shall be private | Member data in a class must be private.  But how about the struct? |
| Rule 16-0-3: Macro *#undef* shall not be used |  |
| Rule 16-0-4: Function-like macros shall not be defined |  |
| Rule 16-2-1: The pre-processor shall only be used for *file inclusion* and *include guards* | Using macro is banned, except for file inclusion and include guards.  Must use constant variables instead. |
| Rule 18-0-2: The library functions *atof*, *atoi* and *atol* from library *<cstdlib>* shall not be used |  |
| Rule 18-4-1: Dynamic heap memory allocation shall not be used | Using dynamic heap memory is banned. |

# Rules

## Language Independent Issues – Unnecessary Constructs

### Rule 0-1-1: A project shall not contain unreachable code

Example 1:

int16\_t func(int16\_t para) {

    int16\_t local;

    ...

    return para;

    para++;     // unreachable – Non-compliant

}

Example 2:

int16\_t func(int16\_t para) {

    while (1) {

// do something

}

    printf(...);     // unreachable – Non-compliant

}

### Rule 0-1-2: A project shall not contain infeasible paths

Example 1:

int func(unsigned int x) {

    if (x > 0) { // x is positive integer

        return 1;

    } else {

        return 2;   // infeasible path

    }

}

Example 2:

int func(unsigned int x) {

bool isRemain = true;

// isRemain = FindLeftover();

bool isSent = SendItem();

bool flag = isRemain || isSent; // infeasible path

...

}

### Rule 0-1-3: A project shall not contain unused variables

int print(int number, std::string extraInfo) {

printf("%d", number); // extraInfo is not used in the function

}

🡪 fix to:

int print(int number, std::string extraInfo) {

(void)extraInfo; // if you intent to not use extraInfo at all

printf("%d", number);

}

### Rule 0-1-4: A project shall not contain non-volatile POD variables having only one use

Example 1:

int func(int value) {

    int result = value \* 2;

    return result; // only used once

}

🡪 fix to:

int func(int value) {

    return value \* 2;

}

Example 2:

int func(int value) {

    bool isOk = checkValue(value);

// do other things but never call "isOk"

}

🡪 fix to:

int func(int value) {

    bool isOk = checkValue(value);

if (!isOk) {

// print error

}

}

### Rule 0-1-5: A project shall not contain unused type declarations

int16\_t funce() {

    typedef int16\_t local\_Type;     // unused

    return 67;

}

### Rule 0-1-6: A project shall not contain instances of non-volatile variables being given values that are never subsequently used

int func(int a, int b) {

    int result;

    if (a > b) {

        result = a - b;

    } else {

        result = b - a;

    }

    if (b > 100) {

        return result;

    }

    // "result" is never used when b <= 100

    return 0;

}

### Rule 0-1-7: The value returned by a function having a *non-void* return type that is not an overloaded operator shall always be used

uint16\_t func1(uint16\_t para) {

    return para;

}

void func2(uint16\_t para) {

    func1(para);        // shouldn't - return value must be assigned somewhere

    (void)func1(para);  // it's ok to do so

}

### Rule 0-1-8: All functions with void return type shall have external side effect(s)

A non-return function which does not have external side effects will **only consume time as it won’t contribute to the generation of any outputs**, which may not meet developer expectations.

The following are examples of external side effects:

* Reading or writing to a file, stream, etc.
* Changing the value of a non local variable
* Changing the value of an argument having reference type
* Using a volatile object
* Raising an exception

Exception: The use of design patterns such as observer/listener, some empty interfaces or use of parameters as out pointer may always be permitted by deviation.

Example:

uint16\_t funcOverriden(uint16\_t para) {

    // do nothing

}

### Rule 0-1-9: There shall be no dead code

int calculateSum(int a, int b) {

    int sum = a + b;

    int product = a \* b; // Dead code

    return sum;

}

### Rule 0-1-10: Every defined function shall be called at least once

### Rule 0-1-11: There shall be no unused parameters (named or unnamed) in *non-virtual* functions

int print(int number, std::string extraInfo) {

printf("%d", number); // extraInfo is not used in the function

}

🡪 fix to:

int print(int number, std::string extraInfo) {

(void)extraInfo; // if you intent to not use extraInfo at all

printf("%d", number);

}

### Rule 0-1-12: There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it

## Language Independent Issues – Storage

### Rule 0-2-1: An object shall not be assigned to an overlapping object

Example 1:

struct Point {

    int x;

    int y;

};

void assignPoint(Point& dest, const Point& src) {

    dest = src;  // Violation: Overlapping assignment

}

int main() {

    Point p1{1, 1};

    Point p2{2, 2};

    assignPoint(p1, p2);

    // p1 is now overwritten with the values of p2, leading to overlapping objects

}

🡪 fix to:

void assignPoint(Point& dest, const Point& src) {

    Point temp{src.x, src.y};

    dest = temp;  // No overlapping assignment

}

Example 2:

struct s {

int16\_t m1 [ 32 ];

};

struct t {

int32\_t m2;

struct s m3;

};

void fn() {

union // Breaks Rule 9–5–1

{

struct s u1;

struct t u2;

} a;

a.u2.m3 = a.u1; // Non-compliant

}

## Language Independent Issues – Runtime Failure

### Rule 0-3-1: Minimization of run-time failures shall be ensured

This can be done by the use of at least one of:

* static analysis tools/techniques
* dynamic analysis tools/techniques
* explicit coding of checks to handle run-time faults

Run-time checking is an issue (not specific to C++) to which developers need to pay special attention, especially as the C++ language is weak in its provision of any run-time checking. C++ implementations are not required to perform many of the dynamic checks that are necessary for robust software. It is therefore an issue that C++ developers need to consider carefully, adding dynamic checks to code wherever there is the potential for run-time errors to occur.

### Rule 0-3-2: If a function generates error information, then that error information shall be tested

A function, whether it’s part of the standard library, a third party library or a user defined function, may provide some means of indicating the occurrence of an error.

## Language Independent Issues – Arithmetic

### Rule 0-4-1: Use of scaled-integer or fixed-point arithmetic shall be documented

### Rule 0-4-2: Use of floating-point shall be documented

### Rule 0-4-3: Floating-point implementations shall comply with a defined floating-point standard

## General – Language

### Rule 1-0-1: All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1"

### Rule 1-0-2: Multiple compilers shall only be used if they have a common, defined interface

“Multiple compilers” includes:

* Mixed languages
* Different compilers
* Different versions of the same compiler
* Different configurations of the same compiler

where "compiler" includes any tool used to translate source code or link object code.

If a module is to be implemented in a language other than C++, or compiled using a different C++ compiler, then it is essential to ensure that the module will integrate correctly with other modules. Some aspects of the behaviour of the C++ language are implementation-defined, and therefore these must be understood for the compiler being used. Examples of issues that need to be understood include:

* Stack usage
* Parameter passing
* The way in which data values are stored (lengths, alignments, aliasing, overlays etc.).

Note that this includes the use of *extern "C"*.

### Rule 1-0-3: The implementation of integer division in the chosen compiler shall be determined and documented

## Lexical Conventions – Character Sets

### Rule 2-2-1: The character set and the corresponding encoding shall be documented

## Lexical Conventions – Trigraph Sequences

### Rule 2-3-1: Trigraphs shall not be used

Rationale

Trigraphs are denoted by a sequence of 2 question marks followed by a specified third character (e.g. ??- represents a "~" (tilde) character and ??) represents a "]"). They can cause accidental confusion with other uses of two question marks.

Example

The string

"(Date should be in the form ??-??-??)"

would probably not meet developer expectations, since the compiler would interpret it as

"(Date should be in the form ~~]"

## Lexical Conventions – Lexical Conventions

### Rule 2-5-1: Digraphs should not be used

The digraphs are:

<% %> <: :> %: %:%:

The use of digraphs may not meet developer expectations.

## Lexical Conventions – Comments

### Rule 2-7-1: The character sequence /\* shall not be used within a C-style comment

/\* A block comment like is a violation of this rule.

So should not be used \*/

func();

Exception: C style comments are accepted (even recommended) for Doxygen comments.

### Rule 2-7-2: Sections of code shall not be "commented out" using C-style comments

Comments should only be used to explain aspects of the code that may not be clear from the source code itself. Code that is commented-out may become out of date, which may lead to confusion when maintaining the code.

A more appropriate method of recording the history of changes in source code (e.g. a Source Control System) should be used instead of commenting-out.

/\* func(); \*/

### Rule 2-7-3: Sections of code should not be "commented out" using C++ comments

// func();

## Lexical Conventions – Identifiers

### Rule 2-10-1: Different identifiers shall be typographically unambiguous

Depending on the font used to display the character set, it is possible for certain glyphs to appear the same, even though the characters are different. This may lead to the developer confusing an identifier with another one.

To help reduce the chance of this, identifiers shall not differ by any combination of:

* Only a mixture of case
* The presence or absence of the underscore character
* The interchange of the letter "O", and the number "0"
* The interchange of the letter "I", and the number "1"
* The interchange of the letter "I", and the letter "l" (el)
* The interchange of the letter "l" (el), and the number "1"
* The interchange of the letter "S" and the number "5"
* The interchange of the letter "Z" and the number "2"
* The interchange of the letter "n" and the letter "h"
* The interchange of the letter "B" and the number "8"
* The interchange of the letter sequence "rn" ("r" followed by "n") with the letter "m"

int32\_t id1\_a\_b\_c;

int32\_t id1\_abc;        // Non-compliant

int32\_t id2\_abc;        // Non-compliant

int32\_t id2\_ABC;        // Non-compliant

int32\_t id3\_a\_bc;

int32\_t id3\_ab\_c;       // Non-compliant

int32\_t id4\_a\_bc;

int32\_t id4\_ab\_c;       // Non-compliant

int32\_t id5\_ii;

int32\_t id5\_11;         // Non-compliant

int32\_t id6\_i0;

int32\_t id6\_1O;         // Non-compliant

int32\_t id7\_in;

int32\_t id7\_1h;         // Non-compliant

int32\_t id8\_Z5;

int32\_t id8\_2S;         // Non-compliant

int32\_t id9\_ZS;

int32\_t id9\_25;         // Non-compliant

### Rule 2-10-2: Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope

void func() {

    int16\_t i;

    {

        int16\_t i;  // This is a different variable

        i = 3;      // It could be confusing as to which i this refers

    }

}

### Rule 2-10-3: A *typedef* name (including qualifications, if any) shall be a unique identifier.

* Reusing a typedef name either as another typedef name or for any other purpose may lead to developer confusion.
* The same typedef shall not be duplicated anywhere in the project, even if the declarations are identical.
* Note that where the type definition is made in a header file, and that header file is included in multiple source files, this rule is not violated.

// f1.cc

namespace NS1 {

typedef int16\_t WIDTH;

}

// f2.cc

namespace NS2 {

float32\_t WIDTH; // Compliant

// NS2::WIDTH is not the same as NS1::WIDTH

}

void f1 () {

typedef int32\_t TYPE;

}

void f2 () {

float32\_t TYPE; // Non-compliant

}

### Rule 2-10-4: A *class*, *union* or *enum* name (including qualifications, if any) shall be a unique identifier

void f1 () {

class TYPE;

}

void f2 () {

float32\_t TYPE; // Non-compliant

}

### Rule 2-10-5: The identifier name of a non-member object or function with *static* storage duration should not be reused

namespace NS1 {

static int32\_t global = 0;

}

namespace NS2 {

void fn () {

int32\_t global; // Non-compliant

}

}

### Rule 2-10-6: If an identifier refers to a type, it shall not also refer to an object or a function in the same scope

typedef struct vector { uint16\_t x ; uint16\_t y; uint16\_t z; } vector; // Non-compliant

## Lexical Conventions – Literals

### Rule 2-13-1: Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.

The defined escape sequences (ISO/IEC 14882:2003 [1] §2.13.2) are:

\n, \t, \v, \b, \r, \f, \a, \\, \?, \', \", \<Octal Number>, \x<Hexadecimal Number>

### Rule 2-13-2: Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used

Any integer constant beginning with a "0" (zero) is treated as octal. Because of this, it is possible for a zero-prefixed constant that is intended to be a decimal number to be incorrectly entered as an octal number, contrary to developer expectations.

code[1] = 109; // Compliant - decimal 109

code[2] = 100; // Compliant - decimal 100

code[3] = 052; // Non-compliant - equivalent to decimal 42

code[4] = 071; // Non-compliant - equivalent to decimal 57

### Rule 2-13-3: A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type

uint32\_t a = 10U; // Compliant

uint32\_t b = 10; // Non-compliant

### Rule 2-13-4: Literal suffixes shall be upper case

const uint32\_t a = 0U;

const uint32\_t b = 0u;          // Non-compliant

const int64\_t c = 0L;

const int64\_t d = 0l;           // Non-compliant

### Rule 2-13-5: Narrow and wide string literals shall not be concatenated

char\_t n\_array[] = "Hello" "World"; // Compliant

wchar\_t w\_array[] = L"Hello" L"World"; // Compliant

wchar\_t mixed[] = "Hello" L"World"; // Non-compliant

## Basic Concepts – Declarations and Definitions

### Rule 3-1-1: It shall be possible to include any header file in multiple translation units without violating the One Definition Rule

// a.h

void f1 (); // Compliant

void f2 () { } // Non-compliant

inline void f3 () { } // Compliant

template <typename T>

void f4 (T) {} // Compliant

int32\_t a; // Non-compliant

// a.cpp

#include "a.h"

### Rule 3-1-2: Functions shall not be declared at block scope

A function declared at block scope will refer to a member of the enclosing namespace, and so the declaration should be explicitly placed at the namespace level.

Additionally, where a declaration statement could either declare a function or an object, the compiler will choose to declare the function. To avoid potential developer confusion over the meaning of a declaration, functions should not be declared at block scope.

class A

{

};

void b1 () {

void f1(); // Non-compliant - declaring a function in block scope

A a(); // Non-compliant - appears to declare an object with no (Don’t understand: Is "A a;" better?)

// arguments to constructor, but it too declares a

// function 'a' returning type 'A' and taking no parameters.

}

### Rule 3-1-3: When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization

int32\_t array1[10];                   // Compliant

extern int32\_t array2[];              // Non-compliant

int32\_t array3[] = {0, 10, 15};     // Compliant

extern int32\_t array4[42];            // Compliant

### Rule 3-2-1: All declarations of an object or function shall have compatible types

// File a.cpp

extern int32\_t a;

extern int32\_t b[];

extern char\_t c;

int32\_t f1();

int32\_t f2(int32\_t);

// File b.cpp

extern int64\_t a; // Non-compliant – not compatible

extern int32\_t b[5]; // Compliant

int16\_t c; // Non-compliant

char\_t f1(); // Non-compliant

char\_t f2(char\_t); // Compliant – not the same function as int32\_t f2 (int32\_t)

### Rule 3-2-2: The One Definition Rule shall not be violated

// File a.cpp

struct S1

{

int64\_t i;

};

// File b.cpp

struct S1

{

int64\_t i;

int32\_t j;

}; // Non-compliant - token sequence different

### Rule 3-2-3: A type, object or function that is used in multiple translation units shall be declared in one and only one file

// header.hpp

extern int16\_t a;

// file1.cpp

#include "header.hpp"

extern int16\_t b;

// file2.cpp

#include "header.hpp"

extern int32\_t b; // Non-compliant - compiler may not detect the error

int32\_t a; // Compliant - compiler will detect the error

### Rule 3-2-4: An identifier with external linkage shall have exactly one definition

// file1.cpp

int32\_t i = 0;

// file2.cpp

int32\_t i = 1; // Non-compliant

### Rule 3-3-1: Objects or functions with external linkage shall be declared in a header file

### Rule 3-3-2: If a function has internal linkage then all re-declarations shall include the *static* storage class specifier

### Rule 3-4-1: An identifier declared to be an object or type shall be defined in a block that minimizes its visibility

void func1(int32\_t a) {

int32\_t b = func2(); // Non-compliant

if (a == 1) {

func3(b);

}

}

🡪 fix to

void func1(int32\_t a) {

if (a == 1) {

int32\_t b = func2(); // Compliant – the declaration of "b" is moved to the if block

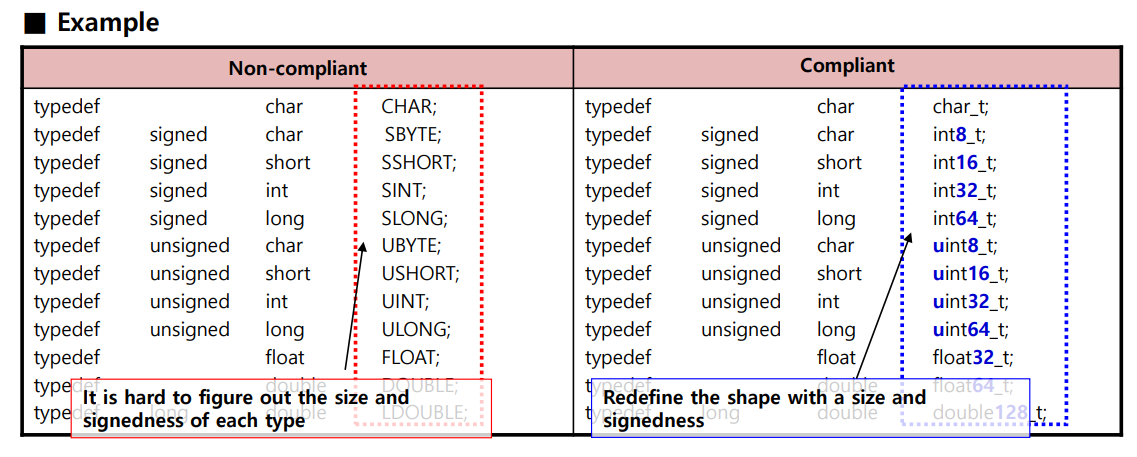
func3(b);

}

}

### Rule 3-9-1: The types used for an *object*, a *function return type*, or a *function parameter* shall be token-for-token identical in all declarations and re-declarations

### Rule 3-9-2: typedefs that indicate size and signedness should be used in place of the basic numerical types



### Rule 3-9-3: The underlying bit representations of *floating-point* values shall not be used

The storage layout used for floating-point values may vary from one compiler to another, and therefore no floating-point manipulations shall be made which rely directly on the way the values are stored. The in-built operators and functions, which hide the storage details from the developer, should be used.

float32\_t func(float32\_t f) {

uint8\_t\* pB = reinterpret\_cast< uint8\_t \* >( &f );

\*(pB + 3) &= 0x7f; // Non-compliant – generate the absolute valueof an IEEE-754 float value.

return (f);

}

## Standard Conversions – Integral Promotions

### Rule 4-5-1: Expressions with type *bool* shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator

The use of bool operands with other operators is unlikely to be meaningful (or intended). This rule allows the detection of such uses, which often occur because the logical operators (&&, || and !) can be easily confused with the bitwise operators (&, | and ~).

bool b1 = true;

bool b2 = false;

int8\_t s8a;

if (b1 & b2) // Non-compliant

if (b1 < b2) // Non-compliant

if (~b1) // Non-compliant

if (b1 ^ b2) // Non-compliant

if (b1 == fa) // Compliant

if (b1 == b2) // Compliant

if (b1 != b2) // Compliant

if (b1 && b2) // Compliant

if (!b1) // Compliant

s8a = b1 ? 3 : 7; // Compliant

### Rule 4-5-2: Expressions with type *enum* shall not be used as operands to built-in operators other than the subscript operator [ ], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=

enum {COLOUR\_0, COLOUR\_1, COLOUR\_2, COLOUR\_CO} colour;

if (COLOUR\_0 == colour ) // Compliant

if ((COLOUR\_0 + COLOUR\_1) == colour) // Non-compliant

if (colour < COLOUR\_COUNT) // Compliant

### Rule 4-5-3: Expressions with type (plain) *char* and *wchar\_t* shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator

Manipulation of character data may generate results that are contrary to developer expectations. For example, ISO/IEC 14882:2003 [1] §2.2(3) only requires that the digits "0" to "9" haveconsecutive numerical values.

**Exception**

* Exceptionally, the following operators may be used if the associated restriction is observed:
* The binary + operator may be used to add an integral value in the range 0 to 9 to '0';
* The binary – operator may be used to subtract character '0';
* The relational operators <, <=, >, >= may be used to determine if a character (or wide character) represents a digit

char\_t ch = 't'; // Compliant

uint8\_t v;

if ((ch >= 'a') && (ch <= 'z')) // Non-compliant

{

}

if ((ch >= '0') && (ch <= '9')) // Compliant by exception

{

v = ch – '0'; // Compliant by exception

v = ch – '1'; // Non-compliant

}

else

{

// ...

}

ch = '0' + v; // Compliant by exception

ch = 'A' + v; // Non-compliant

## Standard Conversions – Pointer Conversions

### Rule 4-10-1: NULL shall not be used as an integer value

#include <cstddef>

void f1(int32\_t);

void f2(int32\_t \*);

void f3() {

    f1(NULL);    // Not-compliant, NULL used as an integer

    f2(NULL);    // Compliant

}

### Rule 4-10-2: Literal zero (0) shall not be used as the null-pointer-constant

#include <cstddef>

void f1(int32\_t);

void f2(int32\_t \*);

void f3() {

    f1(0);    // Compliant

    f2(0);    // Not-compliant, NULL used as an integer

}

## Expressions – General

### Rule 5-0-1: The value of an expression shall be the same under any order of evaluation that the standard permits

TODO

### Rule 5-0-2: Limited dependence should be placed on C++ operator precedence rules in expressions

TODO

### Rule 5-0-3: A *cvalue* expression shall not be implicitly converted to a different underlying type

void f() {

int32\_t s32;

int8\_t s8;

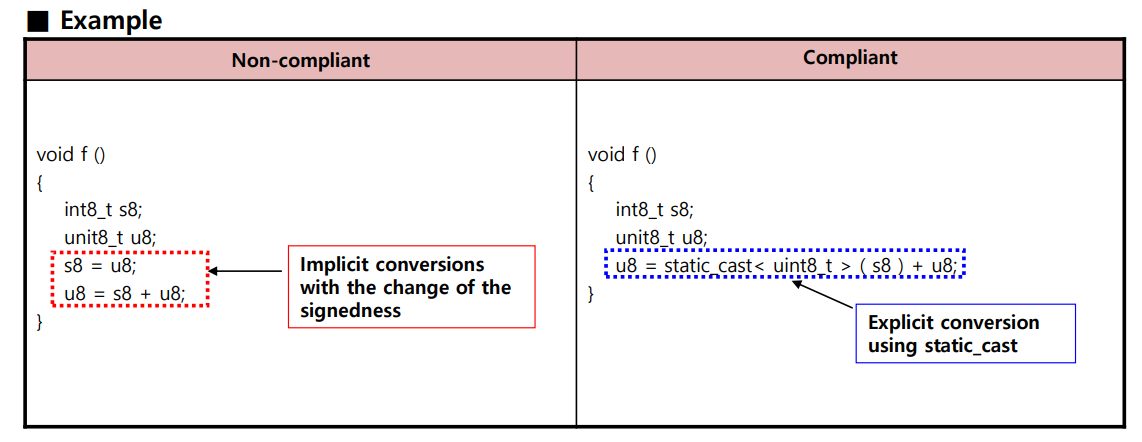
s32 = s8 + s8; // Non-compliant

s32 = static\_cast<int32\_t>(s8) + s8; // Compliant

s32 = s32 + s8; // Compliant

}

### Rule 5-0-4: An implicit integral conversion shall not change the signedness of the underlying type



### Rule 5-0-5: There shall be no implicit floating-integral conversions

void f() {

float32\_t f32;

int32\_t s32;

s32 = f32; // Non-compliant

f32 = s32; // Non-compliant

f32 = static\_cast<float32\_t>(s32); // Compliant

}

### Rule 5-0-6: An implicit integral or floating-point conversion shall not reduce the size of the underlying type

void f() {

int32\_t s32;

int16\_t s16;

s16 = s32; // Non-compliant

s16 = static\_cast<int16\_t>(s32); // Compliant

}

### Rule 5-0-7: There shall be no explicit floating-integral conversions of a *cvalue* expression

void f1() {

int16\_t s16a;

int16\_t s16b;

int16\_t s16c;

float32\_t f32a;

// The following performs integer division

f32a = static\_cast<float32\_t>(s16a / s16b); // Non-compliant

// The following also performs integer division

s16c = s16a / s16b;

f32a = static\_cast<float32\_t>(s16c); // Compliant

// The following performs floating-point division

f32a = static\_cast<float32\_t>(s16a) / s16b; // Compliant

}

### Rule 5-0-8: An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression

void f() {

int16\_t s16;

int32\_t s32;

s32 = static\_cast<int32\_t>(s16 + s16); // Non-compliant

s32 = static\_cast<int32\_t>(s16) + s16; // Compliant

}

### Rule 5-0-9: An explicit integral conversion shall not change the signedness of the underlying type of a *cvalue* expression

void f() {

int8\_t s8;

uint8\_t u8;

s8 = static\_cast<int8\_t>(u8 + u8); // Non-compliant

s8 = static\_cast<int8\_t>(u8) + static\_cast<int8\_t>(u8); // Compliant

}

### Rule 5-0-10: If the bitwise operators ~ and << are applied to an operand with an underlying type of *unsigned char* or *unsigned short*, the result shall be immediately cast to the underlying type of the operand

When the operators ~ and << are applied to small integer types (unsigned char or unsigned short), the operations are preceded by integral promotion, and the result may unexpectedly contain high order bits.

uint8\_t port = 0x5aU;

uint8\_t result\_8;

uint16\_t result\_16;

uint16\_t mode;

result\_8 = (~port) >> 4; // Non-compliant

~port is 0xffa5 on a 16-bit machine but 0xffffffa5 on a 32-bit machine. In either case the value of result is 0xfa, but 0x0a may have been expected. This danger is avoided by inclusion of the cast as shown below:

result\_8 = (static\_cast< uint8\_t > (~port)) >> 4 ; // Compliant

A similar problem exists when the << operator is used on small integer types and high order bits are retained. For example:

result\_16 = ((port << 4) & mode) >> 6; // Non-compliant

The value in result\_16 will depend on the implemented size of an int. Addition of a cast avoids any ambiguity.

result\_16 = (static\_cast<uint16\_t>(static\_cast<uint16\_t>(port) << 4) & mode) >> 6; // Compliant

Using intermediate steps would make this clearer:

uint16\_t port\_16 = static\_cast<uint16\_t> ( port );

uint16\_t port\_shifted = static\_cast<uint16\_t>(port\_16 << 4);

result\_16 = (port\_shifted & mode) >> 6; // Compliant

### Rule 5-0-11: The plain *char* type shall only be used for character values

char\_t a = 'a';     // Compliant

char\_t b = '\r';    // Compliant

char\_t c = 10;      // Non-compliant

char d = 'd'; // Compliant with this rule, but breaks Rule 3–9–2

### Rule 5-0-12: The *signed char* and *unsigned char* type shall only be used for numeric values

int8\_t a = 'a';     // Non-compliant – explicitly signed

uint8\_t b = '\r';   // Non-compliant – explicitly unsigned

int8\_t c = 10;       // Compliant

uint8\_t d = 12U;     // Compliant

signed char e = -11; // Compliant with this rule, but breaks Rule 3–9–2

unsigned char e = 11; // Compliant with this rule, but breaks Rule 3–9–2

### Rule 5-0-13: The condition of an if-statement and the condition of an iteration-statement shall have type *bool*

Exception: A condition of the form type-specifier-seq declarator is not required to have type *bool*.

extern int32\_t\* fn();

extern int32\_t fn2();

extern bool fn3();

while(true); // Compliant

while(bool flag = fn3()) // Compliant

if(u8) // Non-compliant

if(u8 && (bool\_1 <= bool\_2)) // Non-compliant

for(int32\_t x = 10; x; --x) // Non-compliant

while(int32\_t\* p = fn()) // Compliant by exception

if(int32\_t length = fn2()) // Compliant by exception

### Rule 5-0-14: The first operand of a conditional-operator shall have type *bool*

int32\_a = int16\_b ? int32\_c : int32\_d; // Non-compliant

int32\_a = bool\_b ? int32\_c : int32\_d; // Compliant

int32\_a = (int16\_b < 5) ? int32\_c : int32\_d; // Compliant

### Rule 5-0-15: Array indexing shall be the only form of pointer arithmetic

This rule bans the explicit calculation of pointer values. Array indexing shall only be applied to objects defined as an array type. Any explicitly calculated pointer value has the potential to access unintended or invalid memory addresses.

Exception: The increment/decrement operators may be used on iterators implemented by pointers to an array.

Example:

void fn1(uint8\_t\* pointer, uint8\_t array[]) {

pointer = pointer + 5; // Non-compliant – pointer increment

pointer[5] = 0; // Non-compliant – pointer was not declared as array

array[0] = 0;

array[5] = 0; // Compliant

}

uint8\_t fn2(uint8\_t\* iter, uint8\_t\* end) {

uint8\_t result = 0U;

while(iter != end) {

result += \*iter;

++iter; // Compliant by exception

}

return result;

}

### Rule 5-0-16: A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array

### Rule 5-0-17: Subtraction between pointers shall only be applied to pointers that address elements of the same array

### Rule 5-0-18: >, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array

### Rule 5-0-19: The declaration of objects shall contain no more than two levels of pointer indirection

int8\_t \* ptr1; // Compliant

int8\_t \*\* ptr2; // Compliant

int8\_t \*\*\* ptr3; // Non-compliant

### Rule 5-0-20: Non-constant operands to a binary bitwise operator shall have the same underlying type

uint8\_t mask = ~(0x10);

uint16\_t value;

value ^= mask; // Non-compliant

The intent may have been to invert all bits except for bit 5, but the top 8 bits will not have been inverted.

### Rule 5-0-21: Bitwise operators shall only be applied to operands of *unsigned* underlying type

Bitwise operations (~, <<, <<=, >>, >>=, &, &=, ^, ^=, | and |=) are not normally meaningful on signed integers or enumeration constants. Additionally, an implementation-defined result is obtained if a right shift is applied to a negative value.

if ((uint16\_a & int16\_b) == 0x1234U) // Non-compliant

if ((uint16\_a | uint16\_b) == 0x1234U) // Compliant

if (~int16\_a == 0x1234U) // Non-compliant

if (~uint16\_a == 0x1234U) // Compliant

## Expressions – Postfix Expressions

### Rule 5-2-1: Each operand of a logical && or || shall be a postfix-expression

This rule requires that operands are appropriately parenthesized.

Exception: Where an expression consists of either a sequence of only logical && or a sequence of only logical ||, extra parentheses are not required.

if (x == 0 && ishigh) // Non-compliant

if ((x == 0) && ishigh) // Compliant

if (x || y || z) // Compliant by exception, if x, y and z bool

if (x || y && z) // Non-compliant by exception

if (x && !y) // Non-compliant

if (x && (!y)) // Compliant

if (is\_odd(y) && x) // Compliant

if ( (x > c1) && (y > c2) && (z > c3) ) // Compliant by exception

if ( (x > c1) && (y > c2) || (z > c3)) // Non-compliant

if ( (x > c1) && ( (y > c2) || (z > c3) ) ) // Compliant as extra() used

### Rule 5-2-2: A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of *dynamic\_cast*

Casting from a virtual base to a derived class, using any means other than dynamic\_cast has undefined behaviour. The behaviour for dynamic\_cast is defined.

class B { ... };

class D: public virtual B { ... };

D d;

B \*pB = &d;

D \*pD = static\_cast<D\*>(pB); // Non-compliant - undefined behaviour

D \*pD2 = dynamic\_cast<D\*>(pB); // Compliant, but pD2 may be NULL

D &D3 = dynamic\_cast<D&>(\*pB); // Compliant, but may throw an exception

### Rule 5-2-3: Casts from a base class to a derived class should not be performed on polymorphic types

class Colour { /\* ... \*/ };

class Obj {

public:

virtual bool hasColour () const = 0;

virtual Colour getColour () const = 0;

};

class ObjWithColour : public Obj {

public:

virtual bool hasColour () const {

return true;

}

virtual Colour getColour () const {

return m\_colour;

}

private:

Colour m\_colour;

};

void badPrintObject(Obj const & obj) {

ObjWithColour const\* pObj = dynamic\_cast<ObjWithColour const\*>(&obj); // Non-compliant

if (0 != pObj) {

Colour color = pObj->getColour();

}

}

void goodPrintObject(Obj const & obj) { // Compliant

if (obj.hasColour()) {

Colour color = obj.getColour();

}

}

### Rule 5-2-4: C-style casts (other than *void* casts) and functional notation casts (other than explicit constructor calls) shall not be used

A\* a2 = (A\*)(&a1); // Non-compliant

A\* a3 = const\_cast<A\*>(&a1); // Compliant, but breaks Rule 5–2–5

(void)g(); // Compliant by exception

### Rule 5-2-5: A cast shall not remove any *const* or *volatile* type of a pointer or reference

### Rule 5-2-6: A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type

### Rule 5-2-7: An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly

struct S {

int32\_t i;

int32\_t j;

};

class C {

public:

int32\_t i;

public:

int32\_t j;

virtual ~C();

};

void f(S\* s1) {

C\* c1 = reinterpret\_cast<C\*>(s); // Non-compliant

int32\_t i = reinterpret\_cast<int32\_t>(s1); // Compliant, but breaks Rule 5–2–9

C\* c2 = reinterpret\_cast<C\*>(i); // Non-compliant

S\* s2 = reinterpret\_cast<S\*>(i); // Compliant, but breaks Rule 5–2–8

}

### Rule 5-2-8: An object with *integer* type or pointer to *void* type shall not be converted to an object with pointer type

### Rule 5-2-9: A cast should not convert a pointer type to an integral type

### Rule 5-2-10: The increment (++) and decrement (--) operators should not be mixed with other operators in an expression

u8a = ++u8b + u8c--; // Non-compliant

// Compliant

++u8b;

u8a = u8b + u8c;

u8c--;

### Rule 5-2-11: The comma operator, && operator and the || operator shall not be overloaded

### Rule 5-2-12: An identifier with array type passed as a function argument shall not decay to a pointer

When a variable with array type decays to a pointer, its bounds are lost.

If a design requires arrays of different lengths, then a class should be used to encapsulate the array objects and so ensure that the dimensionality is maintained

void f1(int32\_t p[10]);

void f2(int32\_t \*p);

void f3(int32\_t (&p)[10]);

void b() {

int32\_t a[10];

f1(a); // Non-compliant - Dimension "10" lost due to array to pointer conversion

f2(a); // Non-compliant - Dimension "10" lost due to array to pointer conversion

f3(a); // Compliant - Dimension preserved

}

## Expressions – Unary Expressions

### Rule 5-3-1: Each operand of the ! operator, the logical && or the logical || operators shall have type *bool*

This rule is quite similar to Rule 5-0-13.

if ((a < b) && (c < d)) // Compliant

if (1 && (c < d)) // Non-compliant

if (!0) // Non-compliant – also breaks other rules

if (!ptr) // Non-compliant

if (!false) // Compliant with this rule, but breaks others

### Rule 5-3-2: The unary minus operator shall not be applied to an expression whose underlying type is *unsigned*

uint8\_t a = -1U; // Non-compliant

### Rule 5-3-3: The unary & operator shall not be overloaded

### Rule 5-3-4: Evaluation of the operand to the *sizeof* operator shall not contain side effects

int32\_t i;

int32\_t j;

volatile int32\_t k;

j = sizeof(i = 1234); // Non-compliant - j is set to the sizeof the

// type of i which is an int32\_t.

// i is not set to 1234.

j = sizeof(k); // Compliant by exception.

## Expressions – Shift Operators

### Rule 5-8-1: The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand

It is undefined behaviour if the right hand operand is negative, or greater than or equal to the width of the left hand operand.

If, for example, the left hand operand of a left-shift or right-shift is a 16-bit integer, then it is important to ensure that this is shifted only by a number between 0 and 15 inclusive.

u8a = (uint8\_t)(u8a << 7); // Compliant

u8a = (uint8\_t)(u8a << 9); // Non-compliant

u16a = (uint16\_t)((uint16\_t)u8a << 9); // Compliant

## Expressions – Logical AND Operator

### Rule 5-14-1: The right hand operand of a logical && or || operator shall not contain side effects

## Expressions – Assignment Operators

### Rule 5-17-1: The semantic equivalence between a binary operator and its assignment operator form shall be preserved

## Expressions – Comma Operator

### Rule 5-18-1: The comma operator shall not be used

## Expressions – Constant Expressions

### Rule 5-19-1: Evaluation of constant *unsigned integer* expressions should not lead to wrap-around

## Statements – Expression Statements

### Rule 6-2-1: Assignment operators shall not be used in sub-expressions

x = y;

x = y = z; // Non-compliant

if (x != 0) { // Compliant

foo();

}

bool b1 = x != y; // Compliant

bool b2 = x != y; // Compliant

if ((x = y) != 0) { // Non-compliant

foo();

}

if (x = y) { // Non-compliant

foo();

}

if (int16\_t i = foo()) { // Compliant

}

### Rule 6-2-2: Floating-point expressions shall not be directly or indirectly tested for equality or inequality

The result of the test in the following code is unpredictable:

float32\_t x, y;

if (x == y) // Non-compliant

if (x == 0.0f) // Non-compliant

An indirect test is equally problematic and is also prohibited by this rule:

if ((x <= y) && (x >= y)) // Non-compliant

if ((x < y) || (x > y)) // Non-compliant

The following is better, but only if the magnitudes are appropriate:

if (fabs (x – y) <= std::numeric\_limits<float>::epsilon()) // Compliant

### Rule 6-2-3: Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character

## Statements – Compound Statement

### Rule 6-3-1: The statement forming the body of a *switch*, *while*, *do ... while* or for statement shall be a compound statement

// Compliant

for (i = 0; i < N\_ELEMENTS; ++i) {

buffer[i] = 0;

}

// Compliant

for (i = 0; i < N\_ELEMENTS; ++i)

buffer[i] = 0;

## Statements – Selection Statements

### Rule 6-4-1: An *if* construct shall be followed by a compound statement. The *else* keyword shall be followed by either a compound statement, or another *if* statement

### Rule 6-4-2: All *if ... else* if constructs shall be terminated with an *else* clause

if (x < 0) {

log\_error(3);

x = 0;

}

// Compliant

if (x < 0) {

log\_error(3);

x = 0;

} else if (y < 0) {

x = 3;

} else {

}

// Not-compliant

if (x < 0) {

log\_error(3);

x = 0;

} else if (y < 0) {

x = 3;

}

### Rule 6-4-3: A *switch* statement shall be a well-formed *switch* statement

// Compliant

switch (x) {

case 0:

...

break; // break is required here

case 1: // empty clause, break not required

case 2:

break; // break is required here

default: // default clause is required

break; // break is required here, in case a future modification turns this into a case clause

}

### Rule 6-4-4: A *switch-label* shall only be used when the most closely-enclosing compound statement is the body of a *switch* statement

switch (x) {

case 1: // Compliant

if (...) {

case 2: // Non-compliant

DoIt();

}

break;

default:

break;

}

### Rule 6-4-5: An unconditional *throw* or *break* statement shall terminate every non-empty *switch*-clause

switch (x) {

case 0:

break; // Compliant

case 1: // Compliant - empty drop through

case 2: // allows a group

break; // Compliant

case 3:

throw; // Compliant

case 4:

a = b;

// Non-compliant - non empty drop through

default:

// Non-compliant – default must also have "break"

}

### Rule 6-4-6: The *final* clause of a *switch* statement shall be the *default*-clause

### Rule 6-4-7: The condition of a *switch* statement shall not have *bool* type

An if statement gives a clearer representation for a Boolean choice.

### Rule 6-4-8: Every *switch* statement shall have at least one *case*-clause

## Statements – Iteration Statements

### Rule 6-5-1: A *for* loop shall contain a single *loop-counter* which shall not have *floating* type

### Rule 6-5-2: If *loop-counter* is not modified by -- or ++, then, within *condition*, the *loop-counter* shall only be used as an operand to <=, <, > or >=

for (i = 1; i != 10; i += 2) // Non-compliant

for (i = 1; i <= 10; i += 2) // Compliant

for (i = 1; i != 10; ++i) // Compliant

### Rule 6-5-3: The *loop-counter* shall not be modified within *condition* or *statement*

bool modify (int32\_t \* pX) {

\*pX++;

return (\*pX < 10);

}

for (x = 0; modify (&x); ) { // Non-compliant

}

for (x = 0; x < 10;) {

x = x \* 2; // Non-compliant

}

### Rule 6-5-4: The *loop-counter* shall be modified by one of --, ++, -=n, or +=n; where n remains constant for the duration of the loop

for (x = 0; x < 10; x += n) // Compliant if n is not modified within the body of the loop.

for (x = 0; x < 10; x += fn()) // Non-compliant

### Rule 6-5-5: A *loop-control-variable* other than the *loop-counter* shall not be modified within *condition* or *expression*

for (x = 0; (x < 10) && !bool\_a; ++x) {

if (...) {

bool\_a = true; // Compliant

}

}

bool test\_a(bool\* pB) {

\*pB = ... ? true : false;

return \*pB;

}

for (x = 0; (x < 10) && test\_a(&bool\_a); ++x) // Non-compliant

### Rule 6-5-6: A *loop-control-variable* other than the *loop-counter* which is modified in *statement* shall have type *bool*

for (x = 0; (x < 10) && (u8a != 3U); ++x) { // Non-compliant

uint8\_a = fn();

}

for (x = 0; (x < 10) && flag; ++x) { // Compliant

u8a = fn();

flag = u8a != 3U;

}

## Statements – Jump Statements

### Rule 6-6-1: Any label referenced by a *goto* statement shall be declared in the same block, or in a block enclosing the *goto* statement

### Rule 6-6-2: The *goto* statement shall jump to a label declared later in the same function body

### Rule 6-6-3: The *continue* statement shall only be used within a well-formed *for* loop

A well-formed for loop is one which satisfies Rule 6–5–1 to Rule 6–5–6.

void fn() {

for (int32\_t i = 0; i != 10; ++i) {

if ((i % 2) == 0) {

continue; // Compliant

}

// ...

}

int32\_t j = -1;

for (int32\_t i = 0; i != 10 && j != i; ++i) {

if ((i % 2) == 0) {

continue; // Non-compliant – loop is not well-formed

}

// ...

++j; // Not well-formed for loop

}

}

### Rule 6-6-4: For any iteration statement there shall be no more than one *break* or *goto* statement used for loop termination

for (int32\_t i = 0; i < 10; i++) {

if (...) {

break; // Compliant

}

}

for (int32\_t i = 0; i < 10; i++) {

if (...) {

break;

} else if (...) {

break; // Non-compliant – second jump from loop

} else {

...

}

}

### Rule 6-6-5: A function shall have a single point of exit at the end of the function

This is required by IEC 61508 [12], as part of the requirements for a modular approach.

Exception: A function implementing a function-try-block is permitted to have multiple points of exit, one for the try block and one for each catch handler. Throwing an exception that is not caught within the function is not considered a point of exit for this rule.

void fn(void) {

if (...) {

return; // Non-compliant

} try {

if (...) {

throw(1); // Compliant by exception

}

} catch (int32\_t) {

throw; // Compliant by exception

}

return; // Non-compliant

}

void fn2(void) {

try {

return; // Non-compliant

} catch (...) {

return; // Non-compliant

}

}

void fn3(void) {

try {

return; // Compliant by exception

} catch (int32\_t) {

return; // Compliant by exception

} catch (...) {

return; // Compliant by exception

}

}

## Declarations – Specifiers

### Rule 7-1-1: A variable which is not modified shall be *const* qualified

void fn2() {

int32\_t a = fn3(); // Non-compliant, because a won't be modified later

bool b = false; // Compliant, because b will be modified later

if (a == 0) {

const bool c = true; // Compliant

b = true;

}

return b;

}

### Rule 7-1-2: A pointer or reference parameter in a function shall be declared as pointer to *const* or reference to *const* if the corresponding object is not modified

void fn1(int32\_t\* p1, // Non-compliant

int32\_t\* const p2, // Compliant. You can change value pointed by p2, but cannot change p2 itself

const int32\_t\* p3) // Compliant. You can change p3 itself, but cannot change value pointed by p3

{

\*p1 = 10;

\*p2 = 10;

int32\_t\* p4 = ...;

p3 = p4;

}

## Declarations – Enumeration Declarations

### Rule 7-2-1: An expression with *enum* underlying type shall only have values corresponding to the enumerators of the enumeration

## Declarations – Namespaces

### Rule 7-3-1: The global namespace shall only contain *main*, namespace declarations and *extern "C"* declarations

void f1(int32\_t); // Non-compliant

int32\_t x1; // Non-compliant

namespace {

void f2(int32\_t); // Compliant

int32\_t x2; // Compliant

}

int32\_t main() { // Compliant

}

### Rule 7-3-2: The identifier *main* shall not be used for a function other than the global function *main*

int32\_t main () { // Compliant

}

namespace ABC {

int32\_t main () { // Non-compliant

}

}

### Rule 7-3-3: There shall be no unnamed namespaces in header files

// Header.hpp

namespace { // Non-compliant

extern int32\_t x;

}

// Header.hpp

namespace ABC { // Compliant

extern int32\_t x;

}

### Rule 7-3-4: The *using*-*directives* shall not be used. Use the *using-declaration* instead.

The *using-directives* add additional scopes to the set of scopes searched during name lookup. All identifiers in these scopes become visible, increasing the possibility that the identifier found by the compiler does not meet developer expectations.

The *using-declarations* or fully qualified names restricts the set of names considered to only the name explicitly specified, and so are safer options.

namespace NS1 {

int32\_t i1;

int32\_t j1;

int32\_t k1;

}

using namespace NS1; // Non-compliant

namespace NS2 {

int32\_t a;

int32\_t b;

}

using NS2::a; // Compliant

### Rule 7-3-5: Multiple declarations for an identifier in the same namespace shall not straddle a *using-declaration* for that identifier

### Rule 7-3-6: The *using-directives* and *using-declarations* (excluding class scope or function scope *using-declarations*) shall not be used in header files

## Declarations – The asm Declaration

### Rule 7-4-1: All usage of assembler shall be documented

### Rule 7-4-2: Assembler instructions shall only be introduced using the *asm* declaration

### Rule 7-4-3: Assembly language shall be encapsulated and isolated

## Declarations – Linkage Specifications

### Rule 7-5-1: A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function

int32\_t\* fn1(void) {

int32\_t x = 99;

return (&x); // Non-compliant

}

int32\_t\* fn2(int32\_t y) {

return (&y); // Non-compliant

}

int32\_t& fn3(void) {

int32\_t x = 99;

return (x); // Non-compliant

}

int32\_t& fn4(int32\_t y) {

return (y); // Non-compliant

}

int32\_t\* fn5(void) {

static int32\_t x = 0;

return &x; // Compliant

}

### Rule 7-5-2: The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist

void fn(void) {

int8\_t\* p1; {

int8\_t local\_auto;

p1 = &local\_auto; // Non-compliant

}

}

### Rule 7-5-3: A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference

### Rule 7-5-4: Functions should not call themselves, either directly or indirectly

In short, recursion is banned.

int32\_t fn(int32\_t x) {

if (x > 0) {

x = x \* fn(x – 1); // Non-compliant

}

return (x);

}

## Declarator – General

### Rule 8-0-1: An *init-declarator*-list or a *member-declarator-list* shall consist of a single init-declarator or member-declarator respectively

int32\_t i1; int32\_t j1; // Compliant

int32\_t i2, \*j2; // Non-compliant

## Declarator – Meaning of declarators

### Rule 8-3-1: Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments

class Base {

public:

    virtual void fn1(int32\_t a = 0);

    virtual void fn2(int32\_t a = 0);

    virtual void fn3(int32\_t a = 0);

};

class Derived : public Base {

public:

    virtual void fn1(int32\_t a = 0);     // Compliant - same default used

    virtual void fn2(int32\_t a);         // Compliant - no default specified

    virtual void fn3(int32\_t a = 10);    // Non-compliant - different value

};

void f(Derived &d) {

    Base &b = d;

    b.fn1();                             // Will use default of 0

    d.fn1();                             // Will use default of 0

    b.fn2();                             // Will use default of 0

    d.fn2(0);                            // No default value available to use

    b.fn3();                             // Will use default of 0

    d.fn3();                             // Will use default of 10

}

## Declarator – Function Definitions

### Rule 8-4-1: Functions shall not be defined using the ellipsis notation

void MyPrintf (char\_t \* pFormat, ...); // Non-compliant

### Rule 8-4-2: The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration

void fn1(int32\_t a);

void fn2(int32\_t);

void fn1(int32\_t b) // Non-compliant

### Rule 8-4-3: All exit paths from a function with non-*void* return type shall have an explicit *return* statement with an expression

int32\_t fn1(void) {

}                   // Non-compliant

int32\_t fn3(int32\_t x) {

    if (x > 100) {

        throw 42;   // Compliant by exception

    }

    return (x);     // Compliant

}

### Rule 8-4-4: A function identifier shall either be used to call the function or it shall be preceded by a &

if (0 == f) {           // Non-compliant

    // ...

}

void (\*p)(void) = f;    // Non-compliant

if (0 == &f) {          // Compliant

    (f)();              // Compliant as function is called

}

void (\*p)(void) = &f;   // Compliant

## Declarator – Initializers

### Rule 8-5-1: All variables shall have a defined value before they are used

class C {

 public:

    C() : m\_a(10), m\_b(7) {     // Compliant

    }

    C(int32\_t a) : m\_a(a) {     // Non-compliant

    }

    int32\_t GetmB(void) {

        return (m\_b);

    }

 private:

    int32\_t m\_a;

    int32\_t m\_b;

};

### Rule 8-5-2: Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures

// The following are compliant

int16\_t a1[5] = {1, 2, 3, 0, 0}; // Non-zero initialization

int16\_t a2[5] = {0};             // Zero initialization

int16\_t a3[2][2] = {};           // Zero initialization

// The following are non-compliant

int16\_t a4[5] = {1, 2, 3};       // Partial initialization

int16\_t a5[2][2] = {{}, {1, 2}}; // Zero initialization at sub-level

### Rule 8-5-3: In an *enumerator list*, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized

enum colour {

    red = 3,

    blue,

    green,

    yellow

};  // Compliant – only the first item is initialized

enum colour {

    red = 3,

    blue = 4,

    green = 5,

    yellow = 5

};  // Compliant – all items are initialized

enum colour {

    red = 3,

    blue,

    green,

    yellow = 5

};  // Non-compliant

## Class – Member Functions

### Rule 9-3-1: *const* member functions shall not return non-*const* pointers or references to *class-data*

class  {

 public:

    C(int32\_t &b\_)

        : a(new int32\_t[10]), b(b\_) {

    }

    int32\_t\* getA() const {         // Non-compliant - Returns non const pointer to data

        return a;

    }

    int32\_t\* getB() const {         // Non-compliant - Returns non const pointer to data

        return &b;

    }

    const int32\_t\* getC() const {   // Compliant - Returns const pointer to data

        return &b;

    }

 private:

    int32\_t \*a;

    int32\_t &b;

};

### Rule 9-3-2: Member functions shall not return non-*const* handles to *class-data*

class C {

public:

    int32\_t &getA() {       // Non-compliant

        return a;

    }

private:

    int32\_t a;

};

void fn(C &c) {

    int32\_t &a\_ref = c.getA();

    a\_ref = 10;             // External modification of private C::a

}

class D {

public:

    D(int32\_t \*shared)

        : m\_shared(shared) {

    }

    int32\_t \*getA() {

        return m\_shared;    // Compliant - m\_shared is not class-data

    }

private:

    int32\_t \*m\_shared;

};

### Rule 9-3-3: If a member function can be made *static* then it shall be made *static*, otherwise if it can be made *const* then it shall be made *const*

class A {

 public:

    int16\_t f1() {  // Non-compliant – can be const

        return m\_i;

    }

    int16\_t f2() {  // Non-compliant – can be static

        return m\_s;

    }

    int16\_t f3() {  // Compliant – cannot be const or static

        return ++m\_i;

    }

 private:

    int16\_t m\_i;

    static int16\_t m\_s;

};

## Class – Unions

### Rule 9-5-1: Unions shall not be used

The use of unions to access an object in different ways may result in the data being misinterpreted. Therefore, this rule prohibits the use of unions for any purpose

## Class – Bit-Fields

### Rule 9-6-1: When the absolute positioning of bits representing a bit-field is required, then the behaviour and packing of bit-fields shall be documented

### Rule 9-6-2: Bit-fields shall be either *bool* type or an explicitly *unsigned* or *signed* *int* type

struct S {

    signed int a : 2;     // Compliant

    unsigned int b : 2;   // Compliant

    char c : 2;           // Non-compliant

    signed char d : 2;    // Compliant

    unsigned char e : 2;  // Compliant

    short f : 2;          // Non-compliant

    signed short g : 2;   // Compliant

    unsigned short h : 2; // Compliant

    int i : 2;            // Non-compliant

    bool j : 2;           // Compliant

    wchar\_t k : 2;        // Non-compliant

    uint32\_t l : 2;       // Compliant

    int8\_t m : 2;         // Compliant

};

### Rule 9-6-3: Bit-fields shall not have *enum* type

### Rule 9-6-4: Named bit-fields with *signed* integer type shall have a length of more than one bit

struct S {

    signed int a : 1; // Non-compliant

    signed int : 1;   // Compliant

    signed int : 0;   // Compliant

    signed int b : 2; // Compliant

    signed int : 2;   // Compliant

};

## Derived Classes – Multiple Base Classes

### Rule 10-1-1: Classes should not be derived from *virtual* bases

class B {};

class D : public virtual B {};  // Non-compliant - B is a virtual base

### Rule 10-1-2: A base class shall only be declared *virtual* if it is used in a diamond hierarchy

class A {};

class B1: public virtual A {};  // Compliant – A is a common base for C

class B2: public virtual A {};  // Compliant – A is a common base for C

class C: public B1, B2 {};

class D: public virtual A {};   // Non-compliant

### Rule 10-1-3: An accessible base class shall not be both *virtual* and *non-virtual* in the same hierarchy

class A {};

class B1: public virtual A {};

class B2: public virtual A {};

class B3: public A {};

class C: public B1, B2, B3 {};  // Non-compliant – C has two A sub-objects

## Derived Classes – Member Name Lookup

### Rule 10-2-1: All accessible entity names within a multiple inheritance hierarchy should be unique

class B1 {

 public:

    int32\_t count;  // Non-compliant

    void foo();     // Non-compliant

};

class B2 {

 public:

    int32\_t count;  // Non-compliant

    void foo();     // Non-compliant

};

class D : public B1, public B2 {

 public:

    void Bar() {

        ++count;    // Is that B1::count or B2::count?

        foo();      // Is that B1::foo() or B2::foo()?

    }

};

## Derived Classes – Virtual Functions

### Rule 10-3-1: There shall be no more than one definition of each *virtual* function on each path through the inheritance hierarchy

TODO

### Rule 10-3-2: Each overriding *virtual* function shall be declared with the *virtual* keyword

class A {

 public:

    virtual void g();

    virtual void b();

};

class B1 : public A {

 public:

    virtual void g(); // Compliant - explicitly declared "virtual"

    void b();         // Non-compliant - implicitly virtual

};

### Rule 10-3-3: A *virtual* function shall only be overridden by a *pure virtual* function if it is itself declared as *pure virtual*

class A {

 public:

    virtual void foo() = 0;     // foo declared pure

};

class B : public A {

 public:

    virtual void foo()          // foo defined

    {

    }

};

class C : public B {

 public:

    virtual void foo() = 0;     // Non-compliant – foo re-declared pure

};

## Member Access Control – General

### Rule 11-0-1: Member data in non-*POD* class types shall be *private*

class C {

 public:

    int32\_t b; // Non-compliant

 protected:

    int32\_t c; // Non-compliant

 private:

    int32\_t d; // Compliant

};

## Special Member Functions – Constructor

### Rule 12-1-1: An object's dynamic type shall not be used from the body of its constructor or destructor

### Rule 12-1-2: All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes

class B {

 public:

    B() {

    }

    B(int32\_t i) {

    }

};

class D1 : public B {

 public:

    D1() : B(21) {  // Compliant

    }

};

class D2 : public B {

 public:

    D2() {          // Non-compliant

    }

};

### Rule 12-1-3: All constructors that are callable with a single argument of fundamental type shall be declared *explicit*

class C {

 public:

    C(int32\_t a) {            // Non-compliant

    }

};

class D {

 public:

    explicit D(int32\_t a) {   // Compliant

    }

};

## Special Member Functions – Copying Class Objects

### Rule 12-8-1: A *copy constructor* shall only initialize its base classes and the non-*static* members of the class of which it is a member

If a compiler implementation detects that a call to a copy constructor is redundant, then it is permitted to omit that call, even if the copy constructor has a side effect other than to construct a copy of the object. This is called *copy elision*.

It is therefore important to ensure that a copy constructor does not modify the program state as the number of such modifications may be indeterminate.

class A {

 public:

    A(A const &rhs)

        : m\_i(rhs.m\_i)

    {

        ++m\_static;         // Non-compliant

    }

 private:

    int32\_t m\_i;

    static int32\_t m\_static;

};

int32\_t A::m\_static = 0;

A f() {

    return A();

}

void b() {

    A a = f();

}

The value that m\_static has after the call to b() is implementation-defined.

### Rule 12-8-2: The *copy assignment operator* shall be declared *protected* or *private* in an abstract class

An abstract class represents the interface part of a hierarchy. Invoking the copy constructor from the top of such a hierarchy bypasses the underlying implementation resulting in only the base subobjects being copied.

class B1 {

 public:

    B1();

    virtual void f() = 0;

    B1 &operator=(B1 const &rhs);       // Non-compliant

    int32\_t getKind() const {

        return kind;

    }

 private:

    int32\_t kind;

};

class D1 : public B1 {

 public:

    virtual void f() {}

    D1 &operator=(D1 const &rhs);

 private:

    int32\_t member;

};

void f1(B1 &b1, B1 &b2) {

    b1 = b2;

}

As the assignment operator is public, the function f1 can call the operator and so copies the base sub-objects of b1 and b2. As the type of b1 and b2 is an abstract type, b1 and b2 must be subobjects, and so the information contained in the derived objects for both will not be copied.

Making the abstract copy assignment operator protected allows access from the derived classes but not from outside the hierarchy.

class B2 {

 public:

    B2();

    virtual void f() = 0;

    int32\_t getKind() const {

        return kind;

    }

 protected:

    B2 &operator=(B2 const &rhs);   // Compliant

 private:

    int32\_t kind;

};

class D2 : public B2 {

 public:

    virtual void f() {}

    D2 &operator=(D2 const &rhs);

};

void f2(B2 &b1, B2 &b2) {

    b1 = b2;                        // Compiler error will be reported

}

## Template – Template Declarations

### Rule 14–5–1 A non-member generic function shall only be declared in a namespace that is not an associated namespace.

### Rule 14–5–2 A *copy constructor* shall be declared when there is a template constructor with a single parameter that is a generic parameter.

### Rule 14–5–3 A *copy assignment operator* shall be declared when there is a template *assignment operator* with a parameter that is a generic parameter.

## Template – Name Resolution

### Rule 14–6–1 In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a *qualified-id* or *this->*

### Rule 14–6–2 The function chosen by overload resolution shall resolve to a function declared previously in the translation unit

## Template – Template Instantiation and Specialization

### Rule 14–7–1 All *class templates*, *function templates*, *class template member functions* and *class template static members* shall be instantiated at least once.

### Rule 14–7–2 For any given template specialization, an explicit instantiation of the template with the *template arguments* used in the specialization shall not render the program ill-formed

### Rule 14–7–3 All partial and explicit specializations for a template shall be declared in the same file as the declaration of their *primary template*

## Template – Function Template Specialization

### Rule 14–8–1 Overloaded function templates shall not be explicitly specialized.

### Rule 14–8–2 The viable *function set* for a function call should either contain no function specializations, or only contain function specializations.

## Exception Handling – General

### Rule 15-0-1: Exceptions shall only be used for error handling

### Rule 15-0-2: An exception object should not have pointer type

If an exception object of pointer type is thrown and that pointer refers to a dynamically created object, then it may be unclear which function is responsible for destroying it, and when. This ambiguity does not exist if the object is caught by value or reference.

void fn(int16\_t i) {

    static A a1;

    A \*a2 = new A;

    if (i > 10) {

        throw(&a1);     // Non-compliant – pointer type thrown

    } else {

        throw(a2);      // Non-compliant – pointer type thrown

    }

}

### Rule 15-0-3: Control shall not be transferred into a *try* or *catch* block using a *goto* or a *switch* statement

Exception Handling –

### Rule 15-1-1: The *assignment-expression* of a *throw* statement shall not itself cause an exception to be thrown

### Rule 15-1-2: *NULL* shall not be thrown explicitly

try {

    throw(NULL); // Non-compliant

}

catch (int32\_t i) {

    // ...

}

### Rule 15-1-3: An empty *throw* (*throw;*) shall only be used in the compound-statement of a *catch* handler

void f1(void) {

    try {

        throw(42);

    }

    catch (int32\_t i) {

        if (i > 0) {

            throw; // and then re-thrown - Compliant

        }

    }

}

## Exception Handling – Handling An Exception

### Rule 15-3-1: Exceptions shall be raised only after start-up and before termination of the program

### Rule 15-3-2: There should be at least one exception handler to catch all otherwise unhandled exceptions

For the majority of programs this will mean main should look like

int32\_t main() {

    try {

        // program code

    } catch (...) {

        // Handle unexpected exceptions

    }

    return 0;

}

### Rule 15-3-3: Handlers of a *function-try-block* implementation of a *class constructor* or *destructor* shall not reference *non-static* members from this class or its bases

### Rule 15-3-4: Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point

class A { };

class B { };

void f(int32\_t i) throw() {

    try     {

        if (i > 10) {

            throw A();      // Compliant

        } else {

            throw B();      // Non-compliant

        }

    } catch (A const &) {

    }

}

### Rule 15-3-5: A class type exception shall always be caught by reference

If a class type exception object is caught by value, slicing occurs. That is, if the exception object is of a derived class and is caught as the base, only the base class’s functions (including virtual functions) can be called. Also, any additional member data in the derived class cannot be accessed.

If the exception is caught by reference, slicing does not occur.

class ExpBase

{

public:

    virtual const char\_t \*who() {

        return "base";

    };

};

class ExpD1 : public ExpBase

{

public:

    virtual const char\_t \*who() {

        return "type 1 exception";

    };

};

class ExpD2 : public ExpBase

{

public:

    virtual const char\_t \*who() {

        return "type 2 exception";

    };

};

try {

    // ...

    throw ExpD1();

    // ...

    throw ExpBase();

}

catch (ExpBase &b) {    // Compliant

    // ...

    b.who();    // "base", "type 1 exception" or "type 2 exception"

                // depending upon the type of the thrown object

}

// Using the definitions above ...

catch (ExpBase b) {     // Non-compliant - derived type objects will be caught as the base type

    b.who();    // Will always be "base"

    throw b;    // The exception re-thrown is of the base class,

                // not the original exception type

}

### Rule 15-3-6: Where multiple handlers are provided in a single *try-catch* *statement* or *function-try-block* for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class

class B { };

class D : public B { };

try {

    // ...

} catch (D &d) {  // Compliant – Derived class caught before base class

    // ...

} catch (B &b) {  // Compliant – Base class caught after derived class

    // ...

}

### Rule 15-3-7: Where multiple handlers are provided in a single *try-catch statement* or *function-try-block*, any ellipsis (catch-all) handler shall occur last

void f1() {

    try {

        // ...

    }

    catch (int32\_t i) {

        // ...

    }

    catch (...) {   // catch-all handler - compliant

        // ...

    }

}

## Exception Handling – Exception Specifications

### Rule 15-4-1: If a function is declared with an *exception-specification*, then all declarations of the same function (in other translation units) shall be declared with the same set of *type-ids*

## Exception Handling – Special Functions

### Rule 15-5-1: A class destructor shall not exit with an exception

### Rule 15-5-2: Where a function's declaration includes an exception-specification. the function shall only be capable of throwing exceptions of the indicated type(s)

### Rule 15-5-3: The *terminate()* function shall not be called implicitly

## Preprocessing Directives – General

### Rule 16-0-1: The *#include* directives in a file shall only be preceded by other preprocessor directives or comments

#include <f1.h> // Compliant

int32\_t i;

#include <f2.h> // Non-compliant

### Rule 16-0-2: Macros shall only be *#define* or *#undef* in the global namespace

#ifndef MY\_HDR

#define MY\_HDR // Compliant

namespace NS {

#define FOO // Non-compliant

#undef FOO   // Non-compliant

}

#endif

### Rule 16-0-3: Macro *#undef* shall not be used

#ifndef MY\_HDR

#define MY\_HDR

#undef MY\_HDR // Non-compliant

#endif

### Rule 16-0-4: Function-like macros shall not be defined

While macros can provide a speed advantage over functions, functions provide a safer and more robust mechanism. This is particularly true with respect to the type checking of parameters, and the problem of function-like macros potentially evaluating parameters multiple times.

Inline functions should be used instead.

#define FUNC\_MACRO(X) ((X)+(X)) // Non-compliant

### Rule 16-0-5: Arguments to a *function-like macro* shall not contain tokens that look like preprocessing directives

### Rule 16-0-6: In the definition of a *function-like macro*, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##

### Rule 16-0-7: Undefined macro identifiers shall not be used in *#i*f or *#elif* preprocessor directives, except as operands to the defined operator

### Rule 16-0-8: If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token

## Preprocessing Directives – Conditional Inclusion

### Rule 16-1-1: The defined preprocessor operator shall only be used in one of the two standard forms

### Rule 16-1-2: All *#else*, *#elif* and *#endif* preprocessor directives shall reside in the same file as the *#if* or *#ifdef* directive to which they are related

## Preprocessing Directives – Source File Inclusion

### Rule 16-2-1: The pre-processor shall only be used for *file inclusion* and *include guards*

C++ provides safer ways of achieving what is often done using the pre-processor, by way of *inline functions* and *constant declarations*.

// File inclusion - Compliant

#include <iostream>

// Include guard - Compliant

#ifndef HDR         // Compliant

#define HDR         // Compliant

class A {

    ...

}

#endif

// Other uses - Non-compliant

#ifndef HDR         // Compliant

#define HDR         // Compliant

#define X(Y) (Y)    // Non-compliant

#endif

### Rule 16-2-2: C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers

C++ offers *const variable* and *function templates*, which provide a type-safe alternative to the preprocessor.

Note that the use of macros for *type qualifiers* and *storage class specifiers* will break Rule 16–2–1.

// The following are compliant, but breaks Rule 16–2–1

#define STOR extern                     // storage class specifier

// The following are non-compliant

#define CLOCK (xtal / 16)               // Constant expression

#define PLUS2(X) ((X) + 2)              // Macro expanding to expression

#define PI 3.14159F                     // use const object instead

#define int32\_t long                    // use typedef instead

#define STARTIF if(                     // language redefinition

#define INIT(value) {(value), 0, 0}     // braced initializer

#define HEADER "filename.h"             // string literal

### Rule 16-2-3: *Include guards* shall be provided

### Rule 16-2-4: The ', ", /\* or // characters shall not occur in a header file name

### Rule 16-2-5: The \ character should not occur in a header file name

### Rule 16-2-6: The *#include* directive shall be followed by either a <filename> or "filename" sequence

## Preprocessing Directives – Macro Replacement

### Rule 16-3-1: There shall be at most one occurrence of the # or ## operators in a single macro definition

### Rule 16-3-2: The # and ## operators should not be used

## Preprocessing Directives – Pragma Directive

### Rule 16-6-1: All uses of the *#pragma* directive shall be documented

## Library Introduction

### Rule 17-0-1: Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined

### Rule 17-0-2: The names of standard library macros and objects shall not be reused

### Rule 17-0-3: The names of standard library functions shall not be overridden

### Rule 17-0-4: All library code shall conform to MISRA C++

### Rule 17-0-5: The *setjmp* macro and the *longjmp* function shall not be used

## Language Support Library – General

### Rule 18-0-1: The C library shall not be used

Some C++ libraries (e.g, <cstdio>) also have corresponding C versions (e.g, <stdio.h>). This rule requires that the C++ version is used.

### Rule 18-0-2: The library functions *atof*, *atoi* and *atol* from library *<cstdlib>* shall not be used

These functions have undefined behaviour associated with them when the string cannot be converted.

#include <cstdlib>

int32\_t func(const char\_t\* numstr) {

    return atoi(numstr); // Non-compliant

}

### Rule 18-0-3: The library functions *abort*, *exit*, *getenv* and *system* from library *<cstdlib>* shall not be used

The use of these functions leads to implementation-defined behaviour.

#include <cstdlib>

void func() {

    exit(0);    // Non-compliant

}

### Rule 18-0-4: The time handling functions of library *<ctime>* shall not be used

### Rule 18-0-5: The unbounded functions of library *<cstring>* shall not be used

## Language Support Library – Implementation properties

### Rule 18-2-1: The macro *offsetof* shall not be used

### Rule 18-4-1: Dynamic heap memory allocation shall not be used

The use of dynamic memory can lead to out-of-storage run-time failures, which are undesirable.

The built-in new and delete operators, other than the placement versions, use dynamic heap memory. The functions calloc, malloc, realloc and free also use dynamic heap memory.

### Rule 18-7-1: The signal handling facilities of *<csignal>* shall not be used

## Diagnostics Library – Error Numbers

### Rule 19-3-1: The error indicator *errno* shall not be used

The errno is a facility of C++ which should in theory be useful, but which in practice is poorly defined by ISO/IEC 14882:2003 [1]. A non-zero value may or may not indicate that a problem has occurred. Therefore, errno shall not be used.

Even for those functions for which the behaviour of errno is well defined, it’s preferable to check the values of inputs before calling the function rather than relying on using errno to trap error.

#include <cstdlib>

#include <cerrno>

void f1(const char\_t \*str) {

    errno = 0;              // Non-compliant

    int32\_t i = atoi(str);

    if (0 != errno) {     // Non-compliant

        // handle error case

    }

}

## Input/Output Library – General

### Rule 27–0–1 The stream input/output library *<cstdio>* shall not be used.