MISRA C:2012

Guidelines for the use of the C language in critical systems

March 2013

`

## Rules

### A standard C environment

|  |  |
| --- | --- |
| Rule 1.1 | The program shall contain no violations of the standard C syntax and *constraints*, and shall not exceed the implementation’s translation limits |
| [MISRA Guidelines Table 3], [IEC 61508-7: Table C.1], [ISO 26262-6: Table 1] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

The program shall use only those features of the C language and its library that are specified in the chosen version of The Standard (see [Section 3.1](#_heading=h.35nkun2)).

The Standard permits implementations to provide language extensions and the use of such extensions is permitted by this rule.

Except when making use of a language extension, a program shall not:

* Contain any violations of the language syntax described in The Standard;
* Contain any violations of the constraints imposed by The Standard.

A program shall not exceed the translation limits imposed by the implementation. The minimum translation limits are specified by The Standard but an implementation may provide higher limits.

*Note:* a conforming implementation generates a diagnostic for syntax and *constraint* violations but be aware that:

* The diagnostic need not necessarily be an error but could, for example, be a warning;
* The program may be translated and an executable generated despite the presence of a syntax or constraint violation;

***Note****:* a conforming implementation does not need to generate a diagnostic when a translation limit is exceeded; an executable may be generated but it is not guaranteed to execute correctly.

**Rationale**

Problems associated with language features that are outside the supported versions of ISO/IEC 9899 have not been considered during development of these guidelines.

There is anecdotal evidence of some non-conforming implementations failing to diagnose *constraint* violations, for example in [[38]](#_heading=h.1h65qms) p135, example 2 entitled “Error of writing into the const area”.

**Example**

Some C90 compilers provide support for *inline functions* using the *inline* keyword. A C90 program that uses *inline* will be compliant with this rule provided that it is intended to be translated using such a compiler.

Many compilers for embedded targets provide additional keywords that qualify object types with attributes of the memory area in which the object is located, for example:

* \_ \_zpage — the object can be accessed using a short instruction
* \_ \_near — a pointer to the object can be held in 16 bits
* \_ \_far — a pointer to the object can be held in 24 bits

A program using these additional keywords will be compliant with this rule provided that the compiler supports those keywords as a language extension.

**See also**

[Dir 2.1](#_heading=h.1664s55), [Rule 1.2](#_heading=h.1baon6m)

|  |  |
| --- | --- |
| Rule 1.2 | Language extensions should not be used |
| Category | Advisory |
| Analysis | Undecidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

A program that relies on language extensions may be less portable than one that does not. Although The Standard requires that a conforming implementation document any extensions that it provides to the language, there is a risk that this documentation might not provide a full description of the behaviour in all circumstances.

If this rule is not applied, the decision to use each language extension should be justified in the project’s design documentation. The methods by which valid use of each extension will be assured, for example checking the compiler and its diagnostics, should also be documented.

It is recognized that it is necessary to use language extensions in embedded systems. The Standard requires that an extension does not alter the behaviour of any strictly conforming program. For example, a compiler might implement, as an extension, full evaluation of binary logical operators even though The Standard specifies that evaluation stops as soon as the result can be determined. Such an extension does not conform to The Standard because *side effects* in the right-hand operand of a logical AND operator would always occur, giving rise to a different behaviour.

**See also**

[Rule 1.1](#_heading=h.2w5ecyt)

|  |  |
| --- | --- |
| Rule 1.3 | There shall be no occurrence of undefined or critical unspecified behaviour |
| Category | Required |
| Analysis | Undecidable, System |
| Applies to | C90, C99 |

**Amplification**

Some undefined and unspecified behaviours are dealt with by specific rules. This rule prevents all other undefined and critical unspecified behaviours. [Appendix H](#_heading=h.488uthg) lists the undefined behaviours and those unspecified behaviours that are considered critical.

**Rationale**

Any program that gives rise to undefined or unspecified behaviour may not behave in the expected manner. In many cases, the effect is to make the program non-portable but it is also possible for more serious problems to occur. For example, undefined behaviour might affect the result of a computation. If correct operation of the software is dependent on this computation then system safety might be compromised. The problem is particularly difficult to detect if the undefined behaviour only manifests itself on rare occasions.

Many of the MISRA C guidelines have been designed to avoid certain undefined and unspecified behaviours. For example, compliance with all of [Rule 11.4](#_heading=h.j8sehv), [Rule 11.8](#_heading=h.1idq7dh) and [Rule 19.2](#_heading=h.44bvf6o) ensures that it is not possible in C to create a non-*const* qualified pointer to an object declared with a *const*-qualified type. This avoids C90 [Undefined 39] and C99 [Undefined 61]. However, other behaviours are not covered by specific guidelines for example because:

* It is unlikely that the behaviour will be encountered;
* There is no practical guidance that can be given other than the obvious statement that the behaviour should be avoided.

Instead of introducing a guideline for each undefined and critical unspecified behaviour, the MISRA C Guidelines directly address those that are considered most important and most likely to occur in practice. Those behaviours that do not have specific guidelines are all covered together by this single rule. [Appendix H](#_heading=h.488uthg) lists all undefined and critical unspecified behaviours, along with the MISRA C guidelines that prevent their occurrence. It therefore indicates which behaviours are expected to be prevented by this rule and which behaviours are covered by other rules.

*Note:* some implementations may provide well-defined behaviour for some of the undefined and unspecified behaviours listed in The Standard. If such well-defined behaviours are relied upon, including by means of a language extension, it will be necessary to deviate this rule in respect of those behaviours.

**See also**

[Dir 4.1](#_heading=h.25b2l0r)

### Unused code

|  |  |
| --- | --- |
| Rule 2.1 | A project shall not contain *unreachable code* |
| [IEC 61508-7 Section C.5.9], [DO-178C Section 6.4.4.3.c] | |
| Category | Required |
| Analysis | Undecidable, System |
| Applies to | C90, C99 |

**Rationale**

Provided that a program does not exhibit any undefined behaviour, *unreachable code* cannot be executed and cannot have any effect on the program’s outputs. The presence of *unreachable code* may therefore indicate an error in the program’s logic.

A compiler is permitted to remove any *unreachable code* although it does not have to do so. *Unreachable code* that is not removed by the compiler wastes resources, for example:

* It occupies space in the target machine’s memory;
* Its presence may cause a compiler to select longer, slower jump instructions when transferring control around the unreachable code;
* Within a loop, it might prevent the entire loop from residing in an instruction cache.

It is sometimes desirable to insert code that appears to be unreachable in order to handle exceptional cases. For example, in a *switch* statement in which every possible value of the controlling expression is covered by an explicit *case*, a *default* clause shall be present according to [Rule 16.4](#_heading=h.2eclud0). The purpose of the *default* clause is to trap a value that should not normally occur but that may have been generated as a result of:

* Undefined behaviour present in the program;
* A failure of the processor hardware.

If a compiler can prove that a *default* clause is unreachable, it may remove it, thereby eliminating the defensive action. On the assumption that the defensive action is important, it will be necessary either to demonstrate that the compiler does not eliminate the code despite it being unreachable, or to take steps to make the defensive code reachable. The former course of action requires a deviation against this rule, probably with a review of the object code or unit testing being used to support such a deviation. The latter course of action can usually be achieved by means of a *volatile* access. For example, a compiler might determine that the range of values held by x is covered by the *case* clauses in a *switch* statement such as:

|  |
| --- |
| uint16\_t x;  switch ( x ) |

By forcing x to be accessed by means of a *volatile* qualified *lvalue*, the compiler has to assume that the controlling expression could take any value:

|  |
| --- |
| switch ( \*( volatile uint16\_t \* ) &x ) |

*Note:* code that has been conditionally excluded by pre-processor directives is not subject to this rule as it is not presented to the later phases of translation.

**Example**

|  |
| --- |
|  |

**See also**

[Rule 14.3](#_heading=h.1nia2ey), [Rule 16.4](#_heading=h.2eclud0)

|  |  |
| --- | --- |
| Rule 2.2 | There shall be no *dead code* |
| [IEC 61508-7 Section C.5.10], [ISO 26262-6 Section 9.4.5], [DO-178C Section 6.4.4.3.c] | |
| Category | Required |
| Analysis | Undecidable, System |
| Applies to | C90, C99 |

**Amplification**

Any operation that is executed but whose removal would not affect program behaviour constitutes *dead code*. Operations that are introduced by language extensions are assumed always to have an effect on program behaviour.

*Note:* The behaviour of an embedded system is often determined not just by the nature of its actions, but also by the time at which they occur.

*Note: unreachable code* is not *dead code* as it cannot be executed.

**Rationale**

The presence of *dead code* may be indicative of an error in the program’s logic. Since *dead code* may be removed by a compiler, its presence may cause confusion.

**Exception**

A cast to *void* is assumed to indicate a value that is intentionally not being *used*. The cast is therefore not *dead code* itself. It is treated as using its operand which is therefore also not *dead code*.

**Example**

In this example, it is assumed that the object pointed to by p is used in other functions.

|  |
| --- |
|  |

In the following compliant example, the asm keyword is a language extension, not a function call operation, and is therefore not dead code:

|  |
| --- |
| asm ( "NOP" ); |

In the following example, the function g does not contain dead code, and is not itself dead code because it does not contain any operations. However, the call to the function is dead because it could be removed without affecting program behaviour.

|  |
| --- |
|  |

**See also**

[Rule 17.7](#_heading=h.4cmhg48)

|  |  |
| --- | --- |
| Rule 2.3 | A project should not contain unused type declarations |
| Category | Advisory |
| Analysis | Decidable, System |
| Applies to | C90, C99 |

**Rationale**

If a type is declared but not used, then it is unclear to a reviewer if the type is redundant or it has been left unused by mistake.

**Example**

|  |
| --- |
|  |

|  |  |
| --- | --- |
| Rule 2.4 | A project should not contain unused tag declarations |
|  | |
| Category | Advisory |
| Analysis | Decidable, System |
| Applies to | C90, C99 |

Rationale

If a tag is declared but not used, then it is unclear to a reviewer if the tag is redundant or it has been left unused by mistake.

**Example**

In the following example, the tag state is unused and the declaration could have been written without it.

|  |
| --- |
|  |

In the following example, the tag record\_t is used only in the typedef of record1\_t which is used in the rest of the translation unit whenever the type is needed. This typedef can be written in a compliant manner by omitting the tag as shown in the definition of record2\_t.

|  |
| --- |
|  |

|  |  |
| --- | --- |
| Rule 2.5 | A project should not contain unused macro declarations |
| Category | Advisory |
| Analysis | Decidable, System |
| Applies to | C90, C99 |

**Rationale**

If a macro is declared but not used, then it is unclear to a reviewer if the macro is redundant or it has been left unused by mistake.

**Example**

|  |
| --- |
|  |

|  |  |
| --- | --- |
| Rule 2.6 | A function should not contain unused label declarations |
| Category | Advisory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

If a label is declared but not used, then it is unclear to a reviewer if the label is redundant or it has been left unused by mistake.

**Example**

|  |
| --- |
|  |

|  |  |
| --- | --- |
| Rule 2.7 | There should be no unused parameters in functions |
| Category | Advisory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

Most functions will be specified as using each of their parameters. If a function parameter is unused, it is possible that the implementation of the function does not match its specification. This rule highlights such potential mismatches.

**Example**

|  |
| --- |
|  |

void withunusedpara ( uint16\_t \*para1,

int16\_t unusedpara ) /\* Non-compliant - unused \*/

{

\*para1 = 42U;

}

### Comments

|  |  |
| --- | --- |
| Rule 3.1 | The character sequences /\* and // shall not be used within a comment |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

If a comment starting sequence, /\* or //, occurs within a /\* comment, is it quite likely to be caused by a missing \*/ comment ending sequence.

If a comment starting sequence occurs within a // comment, it is probably because a region of code has been commented-out using //.

**Exception**

The sequence // is permitted within a // comment.

**Example**

Consider the following code fragment:

|  |
| --- |
|  |

In reviewing the page containing the call to the function, the assumption is that it is executed code. Because of the accidental omission of the end comment marker, the call to the safety critical function will not be executed.

In the following C99 example, the presence of // comments changes the meaning of the program:

|  |
| --- |
|  |

This gives x = y + z; but would have been x = y; in the absence of the two // comment start sequences.

**See also**

[Dir 4.4](#_heading=h.4ddeoix)

|  |  |
| --- | --- |
| Rule 3.2 | Line-splicing shall not be used in // comments |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C99 |

**Amplification**

Line-splicing occurs when the \ character is immediately followed by a new-line character. If the source file contains multibyte characters, they are converted to the source character set before any splicing occurs.

**Rationale**

If the source line containing a // comment ends with a \ character in the source character set, the next line becomes part of the comment. This may result in unintentional removal of code.

*Note:* line-splicing is described in Section 5.1.1.2(2) of both C90 and C99.

**Example**

In the following non-compliant example, the physical line containing the *if* keyword is logically part of the previous line and is therefore a comment.

|  |
| --- |
|  |

**See also**

[Dir 4.4](#_heading=h.4ddeoix)

### Character sets and lexical conventions

|  |  |
| --- | --- |
| Rule 4.1 | Octal and hexadecimal escape sequences shall be terminated |
| C90 [Implementation 11], C99 [Implementation J.3.4(7, 8)] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

An octal or hexadecimal escape sequence shall be terminated by either:

* The start of another escape sequence, or
* The end of the character constant or the end of a string literal.

**Rationale**

There is potential for confusion if an octal or hexadecimal escape sequence is followed by other characters. For example, the character constant '\x1f' consists of a single character whereas the character constant '\x1g' consists of the two characters '\x1' and 'g'. The manner in which multi-character constants are represented as integers is implementation-defined.

The potential for confusion is reduced if every octal or hexadecimal escape sequence in a character constant or string literal is terminated.

**Example**

In this example, each of the strings pointed to by s1, s2 and s3 is equivalent to “Ag”.

|  |
| --- |
|  |

**See also**

C90: Section 6.1.3.4, C99: Section 6.4.4.4

|  |  |
| --- | --- |
| Rule 4.2 | Trigraphs should not be used |
| Category | Advisory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

Trigraphs are denoted by a sequence of two 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.

*Note:* the so-called digraphs:

|  |
| --- |
| <: :> <% %> %: %:%: |

are permitted because they are tokens. Trigraphs are replaced wherever they appear in the program prior to preprocessing.

**Example**

For example the string.

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

would not behave as expected, actually being interpreted by the compiler as

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

### Identifiers

|  |  |
| --- | --- |
| Rule 5.1 | *External identifiers* shall be distinct |
| C90 [Undefined 7], C99 [Unspecified 7; Undefined 28] | |
| Category | Required |
| Analysis | Decidable, System |
| Applies to | C90, C99 |

**Amplification**

This rule requires that different *external identifiers* be distinct within the limits imposed by the implementation.

The definition of distinct depends on the implementation and on the version of the C language that is being used:

* In C90 the **minimum** requirement is that the first 6 characters of external identifiers are significant but their case is not required to be significant;
* In C99 the **minimum** requirement is that the first 31 characters of external identifiers are significant, with each universal character or corresponding extended source character occupying between 6 and 10 characters.

In practice, many implementations provide greater limits. For example it is common for *external identifiers* in C90 to be case-sensitive and for at least the first 31 characters to be significant.

**Rationale**

If two identifiers differ only in non-significant characters, the behaviour is undefined.

If portability is a concern, it would be prudent to apply this rule using the minimum limits specified in The Standard.

Long identifiers may impair the readability of code. While many automatic code generation systems produce long identifiers, there is a good argument for keeping identifier lengths well below this limit.

*Note:* In C99, if an extended source character appears in an *external identifier* and that character does not have a corresponding universal character, The Standard does not specify how many characters it occupies.

**Example**

In the following example, the definitions all occur in the same translation unit. The implementation in question supports 31 significant case-sensitive characters in *external identifiers*.

|  |
| --- |
|  |

In the following non-compliant example, the implementation supports 6 significant case-insensitive characters in *external identifiers*. The identifiers in the two translation units are different but are not distinct in their significant characters.

|  |
| --- |
|  |

**See also**

[Dir 1.1](#_heading=h.1rvwp1q), [Rule 5.](#_heading=h.3mzq4wv)2, [Rule 5.4](#_heading=h.haapch), [Rule 5.5](#_heading=h.2sioyqq)

|  |  |
| --- | --- |
| Rule 5.2 | Identifiers declared in the same *scope* and name space shall be distinct |
| C90 [Undefined 7], C99 [Undefined 28] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

This rule does not apply if both identifiers are *external identifiers* because this case is covered by [Rule 5.1](#_heading=h.1302m92).

This rule does not apply if either identifier is a *macro identifier* because this case is covered by [Rule 5.4](#_heading=h.haapch) and [Rule 5.5.](#_heading=h.2sioyqq)

The definition of distinct depends on the implementation and on the version of the C language that is being used:

* In C90 the minimum requirement is that the first 31 characters are significant;
* In C99 the minimum requirement is that the first 63 characters are significant, with each universal character or extended source character counting as a single character.

**Rationale**

If two identifiers differ only in non-significant characters, the behaviour is undefined.

If portability is a concern, it would be prudent to apply this rule using the minimum limits specified in The Standard.

Long identifiers may impair the readability of code. While many automatic code generation systems produce long identifiers, there is a good argument for keeping identifier lengths well below this limit.

**Example**

In the following example, the implementation in question supports 31 significant case-sensitive characters in identifiers that do not have external linkage.

The identifier engine\_exhaust\_gas\_temperature\_local is compliant with this rule. Although it is not distinct from the identifier engine\_exhaust\_gas\_temperature\_raw, it is in a different scope. However, it is not compliant with [Rule 5.](#_heading=h.2250f4o)3.

|  |
| --- |
|  |

**See also**

[Dir 1.1](#_heading=h.1rvwp1q), [Rule 5.1](#_heading=h.1302m92), [Rule 5.3](#_heading=h.2250f4o), [Rule 5.4](#_heading=h.haapch), [Rule 5.5](#_heading=h.2sioyqq)

|  |  |
| --- | --- |
| Rule 5.3 | An identifier declared in an inner scope shall not hide an identifier declared in an outer scope |
|  | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

An identifier declared in an inner scope shall be distinct from any identifier declared in an outer scope.

The definition of distinct depends on the implementation and the version of the C language that is being used:

* In C90 the minimum requirement is that the first 31 characters are significant;
* In C99 the minimum requirement is that the first 63 characters are significant, with each universal character or extended source character counting as a single character.

**Rationale**

If an identifier is declared in an inner scope but is not distinct from an identifier that already exists in an outer scope, then the inner-most declaration will “hide” the outer one. This may lead to developer confusion.

*Note:* An identifier declared in one name space does not hide an identifier declared in a different name space.

The terms outer and inner scope are defined as follows:

* Identifiers that have file scope can be considered as having the outermost scope;
* Identifiers that have block scope have a more inner scope;
* Successive, nested blocks, introduce more inner scopes.

**Example**

|  |
| --- |
|  |

**See also**

[Rule 5.](#_heading=h.3mzq4wv)2, [Rule 5.8](#_heading=h.319y80a)

|  |  |
| --- | --- |
| Rule 5.4 | *Macro identifiers* shall be distinct |
| C90 [Undefined 7], C99 [Unspecified 7; Undefined 28] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

This rule requires that, when a macro is being defined, its name be distinct from:

* the names of the other macros that are currently defined; and
* the names of their parameters.

It also requires that the names of the parameters of a given macro be distinct from each other but does not require that macro parameters names be distinct across two different macros.

The definition of distinct depends on the implementation and on the version of the C language that is being used:

* In C90 the minimum requirement is that the first 31 characters of macro identifiers are significant;
* In C99 the minimum requirement is that the first 63 characters of macro identifiers are significant.

In practice, implementations may provide greater limits. This rule requires that *macro identifiers* be distinct within the limits imposed by the implementation.

**Rationale**

If two *macro identifiers* differ only in non-significant characters, the behaviour is undefined. Since macro parameters are active only during the expansion of their macro, there is no issue with parameters in one macro being confused with parameters in another macro.

If portability is a concern, it would be prudent to apply this rule using the minimum limits specified in The Standard.

Long *macro identifiers* may impair the readability of code. While many automatic code generation systems produce long *macro identifiers*, there is a good argument for keeping *macro identifier* lengths well below this limit.

*Note:* In C99, if an extended source character appears in a macro name and that character does not have a corresponding universal character, The Standard does not specify how many characters it occupies.

**Example**

In the following example, the implementation in question supports 31 significant case-sensitive characters in *macro identifiers*.

|  |
| --- |
|  |

**See also**

[Rule 5.1](#_heading=h.1302m92), [Rule 5.](#_heading=h.3mzq4wv)2, [Rule 5.5](#_heading=h.2sioyqq)

|  |  |
| --- | --- |
| Rule 5.5 | Identifiers shall be distinct from macro names |
| C90 [Undefined 7], C99 [Unspecified 7; Undefined 28] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

This rule requires that the names of macros that exist prior to preprocessing be distinct from the identifiers that exist after preprocessing. It applies to identifiers, regardless of scope or name space, and to any macros that have been defined regardless of whether the definition is still in force when the identifier is declared.

The definition of distinct depends on the implementation and the version of the C language that is being used:

* In C90 the minimum requirement is that the first 31 characters are significant;
* In C99 the minimum requirement is that the first 63 characters are significant, with each universal character or extended source character counting as a single character.

**Rationale**

Keeping macro names and identifiers distinct can help to avoid developer confusion.

**Example**

In the following non-compliant example, the name of the *function-like macro* Sum is also used as an identifier. The declaration of the object Sum is not subject to macro-expansion because it is not followed by a ( character. The identifier therefore exists after preprocessing has been performed.

|  |
| --- |
|  |

The following example is compliant because there is no instance of the identifier Sum after preprocessing.

|  |
| --- |
|  |

In the following example, the implementation in question supports 31 significant case-sensitive characters in identifiers that do not have external linkage. The example is non-compliant because the macro name is not distinct from an identifier name with internal linkage in the first 31 characters.

|  |
| --- |
|  |

**See also**

[Rule 5.1](#_heading=h.1302m92), [Rule 5.](#_heading=h.3mzq4wv)2, [Rule 5.4](#_heading=h.haapch)

|  |  |
| --- | --- |
| Rule 5.6 | A *typedef* name shall be a unique identifier |
| Category | Required |
| Analysis | Decidable, System |
| Applies to | C90, C99 |

**Amplification**

A *typedef* name shall be unique across all name spaces and translation units. Multiple declarations of the same *typedef* name are only permitted by this rule if the type definition is made in a *header file* and that *header file* is included in multiple source files.

**Rationale**

Reusing a *typedef* name either as another *typedef* name or as the name of a function, object or enumeration constant, may lead to developer confusion.

**Exception**

The *typedef* name may be the same as the structure, union or enumeration tag name associated with the *typedef*.

**Example**

|  |
| --- |
|  |

**See also**

[Rule 5.7](#_heading=h.17nz8yj)

|  |  |
| --- | --- |
| Rule 5.7 | A tag name shall be a unique identifier |
| Category | Required |
| Analysis | Decidable, System |
| Applies to | C90, C99 |

**Amplification**

The tag shall be unique across all name spaces and translation units.

All declarations of the tag shall specify the same type.

Multiple complete declarations of the same tag are only permitted by this rule if the tag is declared in a *header file* and that *header file* is included in multiple source files.

**Rationale**

Reusing a tag name may lead to developer confusion.

There is also undefined behaviour associated with reuse of tag names in C90 although this is not listed in The Standard’s Annex. This undefined behaviour was recognized in C99 as a *constraint* in Section 6.7.2.3.

**Exception**

The tag name may be the same as the *typedef* name with which it is associated.

**Example**

|  |
| --- |
|  |

**See also**

[Rule 5.6](#_heading=h.3rnmrmc)

|  |  |
| --- | --- |
| Rule 5.8 | Identifiers that define objects or functions with external linkage shall be unique |
| Category | Required |
| Analysis | Decidable, System |
| Applies to | C90, C99 |

**Amplification**

An identifier used as an *external identifier* shall not be used for any other purpose in any name space or translation unit, even if it denotes an object with no linkage.

**Rationale**

Enforcing uniqueness of identifier names in this manner helps avoid confusion. Identifiers of objects that have no linkage need not be unique since there is minimal risk of such confusion.

**Example**

In the following example, file1.c and file2.c are both part of the same project.

|  |
| --- |
|  |

**See also**

[Rule 5.3](#_heading=h.2250f4o)

|  |  |
| --- | --- |
| Rule 5.9 | Identifiers that define objects or functions with internal linkage should be unique |
| Category | Advisory |
| Analysis | Decidable, System |
| Applies to | C90, C99 |

**Amplification**

The identifier name should be unique across all name spaces and translation units. Any identifier used in this way should not have the same name as any other identifier, even if that other identifier denotes an object with no linkage.

**Rationale**

Enforcing uniqueness of identifier names in this manner helps avoid confusion.

**Exception**

An *inline function* with internal linkage may be defined in more than one translation unit provided that all such definitions are made in the same *header file* that is included in each translation unit.

**Example**

In the following example, file1.c and file2.c are both part of the same project.

|  |
| --- |
|  |

**See also**

[Rule 8.10](#_heading=h.45jfvxd)

### Types

|  |  |
| --- | --- |
| Rule 6.1 | Bit-fields shall only be declared with an appropriate type |
| C90 [Undefined 38; Implementation 29], C99 [Implementation J.3.9(1, 2)] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

The appropriate bit-field types are:

* C90: either unsigned int or signed int;
* C99: one of:
* either unsigned int or signed int;
* another explicitly signed or explicitly unsigned integer type that is permitted by the implementation;
* *\_Bool*.

**Note***:* It is permitted to use *typedef*s to designate an appropriate type.

**Rationale**

Using *int* is implementation-defined because bit-fields of type *int* can be either *signed* or *unsigned*.

The use of *enum*, *short*, *char* or any other type for bit-fields is not permitted in C90 because the behaviour is undefined.

In C99, the implementation may define other integer types that are permitted in bit-field declarations.

**Example**

The following example is applicable to C90 and to C99 implementations that do not provide any additional bit-field types. It assumes that the *int* type is 16-bit.

|  |
| --- |
|  |

|  |  |
| --- | --- |
| Rule 6.2 | Single-bit named bit fields shall not be of a signed type |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

According to the C99 Standard Section 6.2.6.2, a single-bit signed bit-field has a single (one) sign bit and no (zero) value bits. In any representation of integers, 0 value bits cannot specify a meaningful value.

A single-bit signed bit-field is therefore unlikely to behave in a useful way and its presence is likely to indicate programmer confusion.

Although the C90 Standard does not provide so much detail regarding the representation of types, the same considerations apply as for C99.

**Note***:* this rule does not apply to unnamed bit fields as their values cannot be accessed.

### Literals and constants

|  |  |
| --- | --- |
| Rule 7.1 | Octal constants shall not be used |
| [Koenig 9] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

Developers writing constants that have a leading zero might expect them to be interpreted as decimal constants.

*Note:* this rule does not apply to octal escape sequences because the use of a leading \ character means that there is less scope for confusion.

**Exception**

The integer constant zero (written as a single numeric digit), is strictly speaking an octal constant, but is a permitted exception to this rule.

**Example**

|  |
| --- |
|  |

|  |  |
| --- | --- |
| Rule 7.2 | A “u” or “U” suffix shall be applied to all integer constants that are represented in an unsigned type |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

This rule applies to:

* Integer constants that appear in the controlling expressions of #if and #elif preprocessing directives;
* Any other integer constants that exist after preprocessing.

*Note:* during preprocessing, the type of an integer constant is determined in the same manner as after preprocessing except that:

* All signed integer types behave as if they were long (C90) or intmax\_t (C99);
* All unsigned integer types behave as if they were unsigned long (C90) or uintmax\_t (C99).

**Rationale**

The type of an integer constant is a potential source of confusion, because it is dependent on a complex combination of factors including:

* The magnitude of the constant;
* The implemented sizes of the integer types;
* The presence of any suffixes;
* The number base in which the value is expressed (i.e. decimal, octal or hexadecimal).

For example, the integer constant 40000 is of type *signed int* in a 32-bit environment but of type *signed long* in a 16-bit environment. The value 0x8000 is of type *unsigned int* in a 16-bit environment, but of type *signed int* in a 32-bit environment.

**Note***:*

* Any value with a “U” suffix is of unsigned type;
* An unsuffixed decimal value less than 231 is of signed type.

But:

* An unsuffixed hexadecimal value greater than or equal to 215 may be of signed or unsigned type;
* For C90, an unsuffixed decimal value greater than or equal to 231 may be of signed or unsigned type.

Signedness of constants should be explicit. If a constant is of an unsigned type, applying a “U” suffix makes it clear that the programmer understands that the constant is unsigned.

*Note:* this rule does not depend on the context in which a constant is used; promotion and other conversions that may be applied to the constant are not relevant in determining compliance with this rule.

**Example**

The following example assumes a machine with a 16-bit *int* type and a 32-bit *long* type. It shows the type of each integer constant determined in accordance with The Standard. The integer constant 0x8000 is non-compliant because it has an unsigned type but does not have a “U” suffix.

|  |  |  |
| --- | --- | --- |
| Constant | Type | Compliance |
| 32767 | *signed int* | Compliant |
| 0x7fff | *signed int* | Compliant |
| 32768 | *signed long* | Compliant |
| 32768u | *unsigned int* | Compliant |
| 0x8000 | *unsigned int* | Non-compliant |
| 0x8000u | *unsigned int* | Compliant |

|  |  |
| --- | --- |
| Rule 7.3 | The lowercase character “l” shall not be used in a literal suffix |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

Using the uppercase suffix “L” removes the potential ambiguity between “1” (digit 1) and “l” (letter “el”) when declaring literals.

**Example**

*Note:* the examples containing the *long long* suffix are applicable only to C99.

|  |
| --- |
|  |

|  |  |
| --- | --- |
| Rule 7.4 | A string literal shall not be *assigned* to an object unless the object’s type is “pointer to *const*-qualified *char*” |
| C90 [Undefined 12], C99 [Unspecified 14; Undefined 30] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

No attempt shall be made to modify a string literal or wide string literal directly.

The result of the address-of operator, &, applied to a string literal shall not be *assigned* to an object unless that object’s type is “pointer to array of *const*-qualified *char*”.

The same considerations apply to wide string literals. A wide string literal shall not be *assigned* to an object unless the object’s type is “pointer to *const*-qualified *wchar\_t*”. The result of the address-of operator, &, applied to a wide string literal shall not be *assigned* to an object unless that object’s type is “pointer to array of *const*-qualified *wchar\_t*”.

**Rationale**

Any attempt to modify a string literal results in undefined behaviour. For example, some implementations may store string literals in read-only memory in which case an attempt to modify the string literal will fail and may also result in an exception or crash.

This rule, when applied in conjunction with others, prevents a string literal from being modified.

It is explicitly unspecified in C99 whether string literals that share a common ending are stored in distinct memory locations. Therefore, even if an attempt to modify a string literal appears to succeed, it is possible that another string literal might be inadvertently altered.

**Example**

The following example shows an attempt to modify a string literal directly.

|  |
| --- |
| "0123456789"[0] = '\*'; /\* Non-compliant \*/ |

These examples show how to prevent modification of string literals indirectly.

|  |
| --- |
|  |

**See also**

[Rule 11.4](#_heading=h.j8sehv), [Rule 11.8](#_heading=h.1idq7dh)

### Declarations and definitions

|  |  |
| --- | --- |
| Rule 8.1 | Types shall be explicitly specified |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90 |

**Rationale**

The C90 standard permits types to be omitted in some circumstances, in which case the *int* type is implicitly specified. Examples of the circumstances in which an implicit *int* might be used are:

* Object declarations;
* Parameter declarations;
* Member declarations;
* typedef declarations;
* Function return types.

The omission of an explicit type might lead to confusion. For example, in the declaration:

|  |
| --- |
| extern void g ( char c, const k ); |

the type of k is *const int* whereas *const char* might have been expected.

**Example**

The following examples show compliant and non-compliant object declarations:

|  |
| --- |
|  |

The following examples show compliant and non-compliant function type declarations:

|  |
| --- |
|  |

The following examples show compliant and non-compliant type definitions:

|  |
| --- |
|  |

The following examples show compliant and non-compliant member declarations:

|  |
| --- |
|  |

See also

[Rule 8.2](#_heading=h.3s49zyc)

|  |  |
| --- | --- |
| Rule 8.2 |  |
| C90 [Undefined 22–25], C99 [Undefined 36–39, 73, 79] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

The early version of C, commonly referred to as K&R C [[30]](#_heading=h.1h65qms), did not provide a mechanism for checking the number of arguments or their types against the corresponding parameters. The type of an object or function did not have to be declared in K&R C since the default type of an object and the default return type of a function was *int*.

The C90 standard introduced function prototypes, a form of function declarator in which the parameter types were declared. This permitted argument types to be checked against parameter types. It also allowed the number of arguments to be checked except when a function prototype specified that a variable number of arguments was expected. The C90 standard did not require the

use of function prototypes for reasons of backward compatibility with existing code. For the same reason, it continued to permit types to be omitted in which case the type would default to *int*.

The C99 standard removed the default *int* type from the language but continued to allow K&R-style function types in which there was no means to supply parameter type information in a declaration and it was optional to supply parameter type information in a definition.

The mismatch between the number of arguments and parameters, their types and the expected and actual return type of a function provides potential for undefined behaviour. The purpose of this rule along with [Rule 8.1](#_heading=h.184mhaj) and [Rule 8.4](#_heading=h.meukdy) is to avoid this undefined behaviour by requiring parameter types and function return types to be specified explicitly. [Rule 17.3](#_heading=h.26sx1u5) ensures that this information is available at the time of a function call, thereby requiring the compiler to diagnose any mismatch that is detected.

This rule also requires that names be specified for all the parameters in a declaration. The parameter names can provide useful information regarding the function interface and a mismatch between a declaration and definition might be indicative of a programming error.

*Note:* An empty parameter list is not valid in a prototype. If a function type has no parameters its *prototype form* uses the keyword *void*.

**Example**

The first example shows declarations of some functions and the corresponding definitions for some of those functions.

|  |
| --- |
|  |

This example section shows the application of the rule to function types other than in function declarations and definitions.

|  |
| --- |
|  |

**See also**

[Rule 8.1](#_heading=h.184mhaj), [Rule 8.4](#_heading=h.meukdy), [Rule 17.3](#_heading=h.26sx1u5)

|  |  |
| --- | --- |
| Rule 8.3 | All declarations of an object or function shall use the same names and type qualifiers |
| C90 [Undefined 10], C99 [Undefined 14], [Koenig 59–62] | |
| Category | Required |
| Analysis | Decidable, System |
| Applies to | C90, C99 |

**Amplification**

Storage class specifiers are not included within the scope of this rule.

**Rationale**

Using types and qualifiers consistently across declarations of the same object or function encourages stronger typing.

Specifying parameter names in function prototypes allows the function definition to be checked for interface consistency with its declarations.

**Exception**

Compatible versions of the same basic type may be used interchangeably. For example, *int*, *signed* and *signed int* are all equivalent.

**Example**

|  |
| --- |
|  |

*Note:* all the above are not compliant with [Dir 4.](#_heading=h.kgcv8k)6.

|  |
| --- |
|  |

In this example the definition of area uses a different type name for the parameter h from that used in the declaration. This does not comply with the rule even though width\_t and height\_t are the same basic type.

|  |
| --- |
|  |

This rule does not require that a function pointer declaration use the same names as a function declaration. The following example is therefore compliant.

|  |
| --- |
|  |

**See also**

[Rule 8.4](#_heading=h.meukdy)

|  |  |
| --- | --- |
| Rule 8.4 | A compatible declaration shall be visible when an object or function with external linkage is defined |
| C90 [Undefined 24], C99 [Undefined 39] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

A compatible declaration is one which declares a compatible type for the object or function being defined.

**Rationale**

If a declaration for an object or function is visible when that object or function is defined, a compiler must check that the declaration and definition are compatible. In the presence of function prototypes, as required by [Rule 8.2](#_heading=h.3s49zyc), checking extends to the number and type of function parameters.

The recommended method of implementing declarations of objects and functions with external linkage is to declare them in a *header file*, and then include the *header file* in all those code files that need them, including the one that defines them (See [Rule 8.5](#_heading=h.ly7c1y)).

**Example**

In these examples there are no declarations or definitions of objects or functions other than those present in the code.

|  |
| --- |
|  |

The following non-compliant definition of func3 also violates [Rule 8.3](#_heading=h.279ka65).

|  |
| --- |
|  |

**See also**

[Rule 8.2,](#_heading=h.3s49zyc) [Rule 8.3,](#_heading=h.279ka65) [Rule 8.](#_heading=h.ly7c1y)5, [Rule 17.3](#_heading=h.26sx1u5)

|  |  |
| --- | --- |
| Rule 8.5 | An external object or function shall be declared once in one and only one file |
| [Koenig 66] | |
| Category | Required |
| Analysis | Decidable, System |
| Applies to | C90, C99 |

**Amplification**

This rule applies to non-defining declarations only.

**Rationale**

Typically, a single declaration will be made in a *header file* that will be included in any translation unit in which the identifier is defined or used. This ensures consistency between:

* The declaration and the definition;
* Declarations in different translation units.

**Note***:* there may be many *header files* in a project, but each external object or function shall only be declared in one *header file*.

**Example**

|  |
| --- |
|  |

**See also**

[Rule 8.4](#_heading=h.meukdy)

|  |  |
| --- | --- |
| Rule 8.6 | An identifier with external linkage shall have exactly one external definition |
| C90 [Undefined 44], C99 [Undefined 78], [Koenig 55, 63–65] | |
| Category | Required |
| Analysis | Decidable, System |
| Applies to | C90, C99 |

**Rationale**

The behaviour is undefined if an identifier is used for which multiple definitions exist (in different files) or no definition exists at all. Multiple definitions in different files are not permitted by this rule even if the definitions are the same. It is undefined behaviour if the declarations are different, or initialize the identifier to different values.

**Example**

In this example the object i is defined twice.

|  |
| --- |
|  |

In this example the object j has one tentative definition and one external definition.

|  |
| --- |
|  |

The following example is non-compliant because the object k has two external definitions. The tentative definition in file4.c becomes an external definition at the end of the translation unit.

|  |
| --- |
|  |

|  |  |
| --- | --- |
| Rule 8.7 | Functions and objects should not be defined with external linkage if they are referenced in only one translation unit |
| [Koenig 56, 57] | |
| Category | Advisory |
| Analysis | Decidable, System |
| Applies to | C90, C99 |

**Rationale**

Restricting the visibility of an object by giving it internal linkage or no linkage reduces the chance that it might be accessed inadvertently. Similarly, reducing the visibility of a function by giving it internal linkage reduces the chance of it being called inadvertently.

Compliance with this rule also avoids any possibility of confusion between an identifier and an identical identifier in another translation unit or a library.

|  |  |
| --- | --- |
| Rule 8.8 | The *static* storage class specifier shall be used in all declarations of objects and functions that have internal linkage |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

Since definitions are also declarations, this rule applies equally to definitions.

**Rationale**

The Standard states that if an object or function is declared with the *extern* storage class specifier and another declaration of the object or function is already visible, the linkage is that specified by the earlier declaration. This can be confusing because it might be expected that the *extern* storage class specifier creates external linkage. The *static* storage class specifier shall therefore be consistently applied to objects and functions with internal linkage.

**Example**

|  |
| --- |
|  |

|  |  |
| --- | --- |
| Rule 8.9 | An object should be defined at block scope if its identifier only appears in a single function |
| Category | Advisory |
| Analysis | Decidable, System |
| Applies to | C90, C99 |

**Rationale**

Defining an object at block scope reduces the possibility that the object might be accessed inadvertently and makes clear the intention that it should not be accessed elsewhere.

Within a function, whether objects are defined at the outermost or innermost block is largely a matter of style.

It is recognized that there are situations in which it may not be possible to comply with this rule. For example, an object with static storage duration declared at block scope cannot be accessed directly from outside the block. This makes it impossible to set up and check the results of unit test cases without using indirect accesses to the object. In this kind of situation, some projects may prefer not to apply this rule.

**Example**

In this compliant example, i is declared at block scope because it is a *loop counter*. There is no need for other functions in the same file to use the same object for any other purpose.

|  |
| --- |
|  |

In this compliant example, the function count keeps track of the number of times it has been called and returns that number. No other function needs to know the details of the implementation of count so the call counter is defined with block scope.

|  |
| --- |
|  |

|  |  |
| --- | --- |
| Rule 8.10 | An *inline function* shall be declared with the static storage class |
| C99 [Unspecified 20; Undefined 67] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C99 |

**Rationale**

If an *inline function* is declared with external linkage but not defined in the same translation unit, the behaviour is undefined.

A call to an *inline function* declared with external linkage may call the external definition of the function, or it may use the inline definition. Although this should not affect the behaviour of the called function, it might affect execution timing and therefore have an impact on a real-time program.

*Note:* an *inline function* can be made available to several translation units by placing its definition in a *header file*.

**See also**

[Rule 5.9](#_heading=h.35xuupr)

|  |  |
| --- | --- |
| Rule 8.11 | When an array with external linkage is declared, its size should be explicitly specified |
| Category | Advisory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

This rule applies to non-defining declarations only. It is possible to define an array and specify its size implicitly by means of initialization.

**Rationale**

Although it is possible to declare an array with incomplete type and access its elements, it is safer to do so when the size of the array may be explicitly determined. Providing size information for each declaration permits them to be checked for consistency. It may also permit a static checker to perform some array bounds analysis without needing to analyse more than one translation unit.

**Example**

|  |
| --- |
|  |

|  |  |
| --- | --- |
| Rule 8.12 | Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

An implicitly-specified enumeration constant has a value 1 greater than its predecessor. If the first enumeration constant is implicitly-specified then its value is 0.

An explicitly-specified enumeration constant has the value of the associated constant expression.

If implicitly-specified and explicitly-specified constants are mixed within an enumeration list, it is possible for values to be replicated. Such replication may be unintentional and may give rise to unexpected behaviour.

This rule requires that any replication of enumeration constants be made explicit, thus making the intent clear.

**Example**

In the following examples the green and yellow enumeration constants are given the same value.

|  |
| --- |
|  |

|  |  |
| --- | --- |
| Rule 8.13 | A pointer should point to a *const*-qualified type whenever possible |
| Category | Advisory |
| Analysis | Undecidable, System |
| Applies to | C90, C99 |

**Amplification**

A pointer should point to a *const*-qualified type unless either:

* It is used to modify an object, or
* It is copied to another pointer that points to a type that is not const-qualified by means of either:
* Assignment, or
* Memory move or copying functions.

For the purposes of simplicity, this rule is written in terms of pointers and the types that they point to. However, it applies equally to arrays and the types of the elements that they contain. An array should have elements with *const*-qualified type unless either:

* Any element of the array is modified, or
* It is copied to a pointer that points to a type that is not const-qualified by the means described above.

**Rationale**

This rule encourages best practice by ensuring that pointers are not inadvertently used to modify objects. Conceptually, it is equivalent to initially declaring:

* All arrays to have elements with const-qualified type, and
* All pointers to point to const-qualified types.

and then removing *const*-qualification only where it is necessary to comply with the *constraints* of the language standard.

**Example**

In the following non-compliant example, p is not used to modify an object but the type to which it points is not *const*-qualified.

|  |
| --- |
|  |

The code would be compliant if the function were defined with:

|  |
| --- |
|  |

The following example violates a *constraint* because an attempt is made to use a *const*-qualified pointer to modify an object.

|  |
| --- |
|  |

In the following example, the pointer s is *const*-qualified but the type it points to is not. Since s is not used to modify an object, this is non-compliant.

|  |
| --- |
|  |

The code would be compliant if the function were defined with:

|  |
| --- |
|  |

In this non-compliant example, none of the elements of the array a are modified but the element type is not *const*-qualified.

|  |
| --- |
|  |

The code would be compliant if the function were defined with:

|  |
| --- |
|  |

|  |  |
| --- | --- |
| Rule 8.14 | The *restrict* type qualifier shall not be used |
| C99 [Undefined 65, 66] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C99 |

**Rationale**

When used with care the *restrict* type qualifier may improve the efficiency of code generated by a compiler. It may also allow improved static analysis. However, to use the *restrict* type qualifier the programmer must be sure that the memory areas operated on by two or more pointers do not overlap.

There is a significant risk that a compiler will generate code that does not behave as expected if *restrict* is used incorrectly.

**Example**

The following example is compliant because the MISRA C Guidelines do not apply to The Standard Library functions. The programmer must ensure that the areas defined by p, q and n do not overlap.

|  |
| --- |
|  |

The following example is non-compliant because a function has been defined using *restrict*.

|  |
| --- |
|  |

### Initialization

|  |  |
| --- | --- |
| Rule 9.1 | The value of an object with automatic storage duration shall not be read before it has been set |
| C90 [Undefined 41], C99 [Undefined 10, 17] | |
| Category | Mandatory |
| Analysis | Undecidable, System |
| Applies to | C90, C99 |

**Amplification**

For the purposes of this rule, an array element or structure member shall be considered as a discrete object.

**Rationale**

According to The Standard, objects with static storage duration are automatically initialized to zero unless initialized explicitly. Objects with automatic storage duration are not automatically initialized and can therefore have indeterminate values.

*Note:* it is sometimes possible for the explicit initialization of an automatic object to be ignored. This will happen when a jump to a label using a *goto* or *switch* statement “bypasses” the declaration of the object; the object will be declared as expected but any explicit initialization will be ignored.

**Example**

|  |
| --- |
|  |

In the following non-compliant C99 example, the *goto* statement jumps past the initialization of x.

*Note:* This example is also non-compliant with [Rul](#_heading=h.11si5id)[e 15.1](#_heading=h.2koq656).

|  |
| --- |
|  |

**See also**

[Rule 15.1](#_heading=h.11si5id), [Rule 15.3](#_heading=h.3ls5o66)

|  |  |
| --- | --- |
| Rule 9.2 | The initializer for an aggregate or union shall be enclosed in braces |
| C90 [Undefined 42], C99 [Undefined 76, 77] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

This rule applies to initializers for both objects and subobjects.

An initializer of the form { 0 }, which sets all values to 0, may be used to initialize subobjects without nested braces.

*Note:* this rule does not itself require explicit initialization of objects or subobjects.

**Rationale**

Using braces to indicate initialization of subobjects improves the clarity of code and forces programmers to consider the initialization of elements in complex data structures such as multi-dimensional arrays or arrays of structures.

**Exception**

1. An array may be initialized using a string literal.
2. An automatic structure or union may be initialized using an expression with compatible structure or union type.
3. A designated initializer may be used to initialize part of a subobject.

**Example**

The following three initializations, which are permitted by The Standard, are equivalent. The first form is not permitted by this rule because it does not use braces to show subarray initialization explicitly.

|  |
| --- |
|  |

In the following example, the initialization of z1 is compliant by virtue of Exception 3 because a designated initializer is used to initialize the subobject z1[ 1 ]. The initialization of z2 is also compliant for the same reason. The initialization of z3 is non-compliant because part of the subobject z3[ 1 ] is intialized with a positional initializer but is not enclosed in braces. The initialization of z4 is compliant because a designated initializer is used to initialize the subobject z4[ 0 ] and the initializer for subobject z4[ 1 ] is brace-enclosed.

|  |
| --- |
|  |

The first line in the following example initializes 3 subarrays without using nested braces. The second and third lines show equivalent ways to write the same initializer.

|  |
| --- |
|  |

|  |  |
| --- | --- |
| Rule 9.3 | Arrays shall not be partially initialized |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

If any element of an array object or subobject is explicitly initialized, then the entire object or subobject shall be explicitly initialized.

**Rationale**

Providing an explicit initialization for each element of an array makes it clear that every element has been considered.

**Exception**

1. An initializer of the form { 0 } may be used to explicitly initialize all elements of an array object or subobject.
2. An array whose initializer consists only of designated initializers may be used, for example to perform a sparse initialization.
3. An array initialized using a string literal does not need an initializer for every element.

**Example**

|  |
| --- |
|  |

In the following compliant example, each element of the array arr is initialized:

|  |
| --- |
|  |

In the following example, array elements 6 to 9 are implicitly initialized to '\0':

|  |
| --- |
|  |

|  |  |
| --- | --- |
| Rule 9.4 | An element of an object shall not be initialized more than once |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C99 |

**Amplification**

This rule applies to initializers for both objects and subobjects.

The provision of *designated initializers* in C99 allows the naming of the components of an aggregate (structure or array) or of a union to be initialized within an initializer list and allows the object’s elements to be initialized in any order by specifying the array indices or structure member names they apply to (elements having no initialization value assume the default for uninitialized objects).

**Rationale**

Care is required when using *designated initializers* since the initialization of object elements can be inadvertently repeated leading to overwriting of previously initialized elements. The C99 Standard does not specify whether the *side effects* in an overwritten initializer occur or not although this is not listed in Annex J.

In order to allow sparse arrays and structures, it is acceptable to only initialize those which are necessary to the application.

**Example**

Array initialization:

|  |
| --- |
|  |

In the following non-compliant example, it is unspecified whether the *side effect* occurs or not:

|  |
| --- |
|  |

|  |  |
| --- | --- |
| Rule 9.5 | Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C99 |

**Amplification**

The rule applies equally to an array subobject that is a flexible array member.

**Rationale**

If the size of an array is not specified explicitly, it is determined by the highest index of any of the elements that are initialized. When using designated initializers it may not always be clear which initializer has the highest index, especially when the initializer contains a large number of elements.

To make the intent clear, the array size shall be declared explicitly. This provides some protection if, during development of the program, the indices of the initialized elements are changed as it is a *constraint* violation (C99 Section 6.7.8) to initialize an element outside the bounds of an array.

**Example**

|  |
| --- |
|  |

### The essential type model

**Rationale**

The rules in this section collectively define the *essential type model* and restrict the C type system so as to:

1. Support a stronger system of type-checking;
2. Provide a rational basis for defining rules to control the use of implicit and explicit type conversions;
3. Promote portable coding practices;
4. Address some of the type conversion anomalies found within ISO C.

The *essential type model* does this by allocating an *essential type* to those objects and expressions which ISO C considers to be of arithmetic type. For example, adding an *int* to a *char* gives a result having *essentially character* type rather than the *int* type that is actually produced by integer promotion.

The full rationale behind the *essential type model* is given in [Appendix C](#_heading=h.3sv78d1) with [Appendix D](#_heading=h.4jpj0b3) providing a comprehensive definition of the *essential type* of any arithmetic expression.

#### Essential type

The *essential type* of an object or expression is defined by its *essential type category* and size. The *essential type category* of an expression reflects its underlying behaviour and may be:

* Essentially Boolean;
* Essentially character;
* Essentially enum;
* Essentially signed;
* Essentially unsigned;
* Essentially floating.

*Note:* each enumerated type is a unique *essentially enum type* identified as *enum<i>*. This allows different enumerated types to be handled as distinct types, which supports a stronger system of type-checking. One exception is the use of an enumerated type to define a Boolean value in C90. Such types are considered to have *essentially Boolean* type. Another exception is the use of *anonymous enumerations* as defined in [Appendix D.](#_heading=h.4jpj0b3) *Anonymous enumerations* are a way of defining a set of related

constant integers and are considered to have an *essentially signed* type. 81

When comparing two types of the same type category, the terms *wider* and *narrower* are used to describe their relative sizes as measured in bytes. Two different types are sometimes implemented with the same size.

The following table shows how the standard integer types map on to *essential type categories*.

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

*Note:* C99 implementations may provide *extended integer types* each of which would be allocated a location appropriate to its rank and signedness (see C99 Section 6.3.1.1).

The restrictions enforced by the rules in this section also apply to the compound assignment operators (e.g. ^=) as these are equivalent to assigning the result obtained from the use of one of the arithmetic, bitwise or shift operators. For example:

u8a += u8b + 1U;

is equivalent to:

u8a = u8a + ( u8b + 1U );

|  |  |
| --- | --- |
| Rule 10.1 | Operands shall not be of an inappropriate *essential type* |
| C90 [Unspecified 23; Implementation 14, 17, 19, 32]  C99 [Undefined 13, 49; Implementation J3.4(2, 5), J3.5(5), J3.9(6)] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Operator | Operand | Essential type category of arithmetic operand | | | | | |
| Boolean | character | enum | signed | unsigned | floating |
| [ ] | integer | 3 | 4 |  |  |  | 1 |
| + (unary) |  | 3 | 4 | 5 |  |  |  |
| - (unary) |  | 3 | 4 | 5 |  | 8 |  |
| + - | either | 3 |  | 5 |  |  |  |
| \* / | either | 3 | 4 | 5 |  |  |  |
| % | either | 3 | 4 | 5 |  |  | 1 |
| < > <= >= | either | 3 |  |  |  |  |  |
| == != | either |  |  |  |  |  |  |
| ! && || | any |  | 2 | 2 | 2 | 2 | 2 |

In the following table a number within a cell indicates where a restriction applies to the use of an *essential type* as an operand to an operator. These numbers correspond to paragraphs in the Rationale section below and indicate why each restriction is imposed.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Operator | Operand | Essential type category of arithmetic operand | | | | | |
| Boolean | character | enum | signed | unsigned | floating |
| << >> | left | 3 | 4 | 5, 6 | 6 |  | 1 |
| << >> | right | 3 | 4 | 7 | 7 |  | 1 |
| ~ & | ^ | any | 3 | 4 | 5, 6 | 6 |  | 1 |
| ?: | 1st |  | 2 | 2 | 2 | 2 | 2 |
| ?: | 2nd and 3rd |  |  |  |  |  |  |

Under this rule the ++ and -- operators behave the same way as the binary + and - operators.

Other rules place further restrictions on the combination of *essential types* that may be used within an expression.

**Rationale**

1. The use of an expression of *essentially floating type* for these operands is a *constraint* violation.
2. An expression of *essentially Boolean type* should always be used where an operand is interpreted as a Boolean value.
3. An operand of *essentially Boolean type* should not be used where an operand is interpreted as a numeric value.
4. An operand of *essentially character type* should not be used where an operand is interpreted as a numeric value. The numeric values of character data are implementation-defined.
5. An operand of *essentially enum type* should not be used in an arithmetic operation because an *enum* object uses an implementation-defined integer type. An operation involving an *enum* object may therefore yield a result with an unexpected type. Note that an enumeration constant from an *anonymous enum* has *essentially signed type*.
6. Shift and bitwise operations should only be performed on operands of *essentially unsigned type*. The numeric value resulting from their use on *essentially signed types* is implementation- defined.
7. The right hand operand of a shift operator should be of *essentially unsigned type* to ensure that undefined behaviour does not result from a negative shift.
8. An operand of *essentially unsigned type* should not be used as the operand to the unary minus operator, as the signedness of the result is determined by the implemented size of *int*.

**Exception**

A non-negative *integer constant expression* of *essentially signed type* may be used as the right hand operand to a shift operator.

Example

|  |  |  |
| --- | --- | --- |
| enum enuma { a1, a2, a3 } ena, enb; | /\* Essentially enum<enuma> | \*/ |
| enum { K1 = 1, K2 = 2 }; | /\* Essentially signed | \*/ |

The following examples are non-compliant. The comments refer to the numbered rationale item that results in the non-compliance.

|  |  |  |
| --- | --- | --- |
| f32a & 2U | /\* Rationale 1 - constraint violation | \*/ |
| f32a << 2 | /\* Rationale 1 - constraint violation | \*/ |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| cha && bla | | | /\* | Rationale | 2 | - | char type used as a Boolean value | | | | | \*/ |
| ena ? a1 : a2 | | | /\* | Rationale | 2 | - | enum type used as a Boolean value | | | | | \*/ |
| s8a && bla | | | /\* | Rationale | 2 | - | signed type used as a Boolean value | | | | | \*/ |
| u8a ? a1 : a2 | | | /\* | Rationale | 2 | - | unsigned type used as a Boolean value | | | | | \*/ |
| f32a && bla | | | /\* | Rationale | 2 | - | floating type used as a Boolean value | | | | | \*/ |
| bla | \* blb | | /\* | Rationale | 3 | - Boolean used as a | | | | | numeric value | \*/ |
| bla | > blb | | /\* | Rationale | 3 | - Boolean used as a | | | | | numeric value | \*/ |
| cha | & chb | | /\* | Rationale | 4 | - | char | type | used | as | a numeric value | \*/ |
| cha | << 1 | | /\* | Rationale | 4 | - | char | type | used | as | a numeric value | \*/ |
| ena-- | |  | /\* | Rationale | 5 | - | enum | type | used | in | arithmetic operation | \*/ |
| ena \* | | a1 | /\* | Rationale | 5 | - | enum | type | used | in | arithmetic operation | \*/ |
| s8a & | | 2 | /\* | Rationale | 6 | - bitwise operation on signed type | | | | | | \*/ |
| 50 << | | 3U | /\* | Rationale | 6 | - shift operation on signed type | | | | | | \*/ |
| u8a << | | s8a | /\* | Rationale | 7 | - shift | | magnitude | | uses signed type | | \*/ |
| u8a << | | -1 | /\* | Rationale | 7 | - shift | | magnitude | | uses signed type | | \*/ |
| -u8a | /\* | | | Rationale | 8 | - | unary | minus on unsigned type | | | | \*/ |

The following example is non-compliant with this rule and also violates [Rule 10.3](#_heading=h.1l354xk):

ena += a1 /\* Rationale 5 - enum type used in arithmetic operation \*/

The following examples are compliant:

bla && blb

bla ? u8a : u8b

cha - chb cha > chb

ena > a1

K1 \* s8a /\* Compliant as K1 from anonymous enum \*/

s8a + s16b

-( s8a ) \* s8b s8a > 0

--s16b

u8a + u16b u8a & 2U

u8a > 0U u8a << 2U

u8a << 1 /\* Compliant by exception \*/

f32a + f32b f32a > 0.0

The following is compliant with this rule but violates [Rule 10.2](#_heading=h.452snld)

cha + chb

**See also**

[Rule 10.2](#_heading=h.452snld)

|  |  |
| --- | --- |
| Rule 10.2 | Expressions of *essentially character type* shall not be |
|  | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

The appropriate uses are:

1. For the + operator, one operand shall have *essentially character type* and the other shall have *essentially signed type* or *essentially unsigned type*. The result of the operation has *essentially character type*.
2. For the - operator, the first operand shall have *essentially character type* and the second shall have *essentially signed type*, *essentially unsigned type* or *essentially character type*. If both operands have *essentially character type* then the result has the *standard type* (usually *int* in this case) else the result has *essentially character type*.

**Rationale**

Expressions with *essentially character type* (character data) shall not be used arithmetically as the data does not represent numeric values.

The uses above are permitted as they allow potentially reasonable manipulation of character data. For example:

* + Subtraction of two operands with *essentially character type* might be used to convert between digits in the range ‘0’ to ‘9’ and the corresponding ordinal value;
  + Addition of an *essentially character type* and an *essentially unsigned type* might be used to convert an ordinal value to the corresponding digit in the range ‘0’ to ‘9’;
  + Subtraction of an *essentially unsigned type* from an *essentially character type* might be used to convert a character from lowercase to uppercase.

**Example**

The following examples are compliant:

'0' + u8a /\* Convert u8a to digit \*/ s8a + '0' /\* Convert s8a to digit \*/ cha - '0' /\* Convert cha to ordinal \*/ '0' - s8a /\* Convert -s8a to digit \*/

The following examples are non-compliant:

s16a - 'a' '0' + f32a

cha + ':' cha - ena

**See also**

[Rule 10.1](#_heading=h.2ce457m)

|  |  |
| --- | --- |
| Rule 10.3 | The value of an expression shall not be assigned to an object with a narrower *essential type* or of a different *essential type category* |
| C90 [Undefined 15; Implementation 16]  C99 [Undefined 15, 16; Implementation 3.5(4)] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

The following operations are covered by this rule:

1. *Assignment* as defined in the Glossary;
2. The conversion of the *constant expression* in a *switch* statement’s *case* label to the promoted type of the controlling expression.

**Rationale**

The C language allows the programmer considerable freedom and will permit assignments between different arithmetic types to be performed automatically. However, the use of these implicit conversions can lead to unintended results, with the potential for loss of value, sign or precision. Further details of concerns with the C type system can be found in [Appendix](#_heading=h.3sv78d1) C.

The use of stronger typing, as enforced by the MISRA *essential type model*, reduces the likelihood of these problems occurring.

**Exception**

1. A non-negative *integer constant expression* of *essentially signed type* may be *assigned* to an object of *essentially unsigned type* if its value can be represented in that type.
2. The initializer { 0 } may be used to initialize an aggregate or union type.

**Example**

enum enuma { A1, A2, A3 } ena; enum enumb { B1, B2, B3 } enb; enum { K1=1, K2=128 };

The following are compliant:

uint8\_t u8a = 0; /\* By exception \*/ bool\_t flag = ( bool\_t ) 0;

bool\_t set = true; /\* true is essentially Boolean \*/ bool\_t get = ( u8b > u8c );

ena = A1;

s8a = K1; /\* Constant value fits \*/

u8a = 2; /\* By exception \*/

u8a = 2 \* 24; /\* By exception \*/ cha += 1; /\* cha = cha + 1 assigns character to character \*/

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| pu8a | = | pu8b; |  |  | /\* | Same | essential type | \*/ |
| u8a | = | u8b + | u8c | + u8d; | /\* | Same | essential type | \*/ |

u8a = ( uint8\_t ) s8a; /\* Cast gives same essential type \*/

u32a = u16a; /\* Assignment to a wider essential type \*/ u32a = 2U + 125U; /\* Assignment to a wider essential type \*/ use\_uint16 ( u8a ); /\* Assignment to a wider essential type \*/ use\_uint16 ( u8a + u16b ); /\* Assignment to same essential type \*/

The following are non-compliant as they have different *essential type categories*:

uint8\_t u8a = 1.0f; /\* unsigned and floating \*/

bool\_t bla = 0; /\* boolean and signed \*/

cha = 7; /\* character and signed \*/

u8a = 'a'; /\* unsigned and character \*/

u8b = 1 - 2; /\* unsigned and signed \*/ u8c += 'a'; /\* u8c = u8c + 'a' assigns character to unsigned \*/ use\_uint32 ( s32a ); /\* signed and unsigned \*/

The following are non-compliant as they contain assignments to a narrower *essential type*:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| s8a = K2;  u16a = u32a; |  |  | /\*  /\* | Constant uint32\_t | value does not to uint16\_t | fit | \*/  \*/ |
| use\_uint16 ( | u32a ); |  | /\* | uint32\_t | to uint16\_t |  | \*/ |
| uint8\_t foo1  {  return x; | ( uint16\_t | x ) | /\* | uint16\_t | to uint8\_t |  | \*/ |

}

**See also**

[Rule 10.4](#_heading=h.rjefff), [Rule 10.5](#_heading=h.2k82xt6), [Rule 10.6](#_heading=h.zdd80z)

|  |  |
| --- | --- |
| Rule 10.4 | Both operands of an operator in which the *usual arithmetic conversions*  are performed shall have the same *essential type category* |
| C90 [Implementation 21], C99 [Implementation 3.6(4)] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

This rule applies to operators that are described in *usual arithmetic conversions* (see C90 Section 6.2.1.5, C99 Section 6.3.1.8). This includes all the binary operators, excluding the shift, logical &&, logical || and comma operators. In addition, the second and third operands of the ternary operator are covered by this rule.

*Note:* the increment and decrement operators are not covered by this rule.

**Rationale**

The C language allows the programmer considerable freedom and will permit conversions between different arithmetic types to be performed automatically. However, the use of these implicit conversions can lead to unintended results, with the potential for loss of value, sign or precision. Further details of concerns with the C type system can be found in [Appendix](#_heading=h.3sv78d1) C.

The use of stronger typing, as enforced by the MISRA *essential type model*, allows implicit conversions to be restricted to those that should then produce the answer expected by the developer.

**Exception**

The following are permitted to allow a common form of character manipulation to be used:

1. The binary + and += operators may have one operand with *essentially character* type and the other operand with an *essentially signed* or *essentially unsigned* type;
2. The binary - and -= operators may have a left-hand operand with *essentially character* type and a right-hand operand with an *essentially signed* or *essentially unsigned* type.

**Example**

enum enuma { A1, A2, A3 } ena; enum enumb { B1, B2, B3 } enb;

The following are compliant as they have the same *essential type category*:

ena > A1 u8a + u16b

The following is compliant by exception 1:

cha += u8a

The following is non-compliant with this rule and also violates [Rule 10.3](#_heading=h.1l354xk):

s8a += u8a /\* signed and unsigned \*/

The following are non-compliant:

u8b + 2 /\* unsigned and signed \*/ enb > A1 /\* enum<enumb> and enum<enuma> \*/ ena == enb /\* enum<enuma> and enum<enumb> \*/ u8a += cha /\* unsigned and char \*/

**See also**

[Rule 10.3](#_heading=h.1l354xk), [Rule 10.7](#_heading=h.zdd80z)

Rule 10.5

The value of an expression should not be cast to an inappropriate

*essential type*

Category Advisory

Analysis Decidable, Single Translation Unit

Applies to C90, C99

**Amplification**

The casts which should be avoided are shown in the following table, where values are cast (explicitly converted) to the *essential type category* of the first column.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Essential type category | from | | | | | |
| to | *Boolean* | *character* | *enum* | *signed* | *unsigned* | *floating* |
| *Boolean* |  | Avoid | Avoid | Avoid | Avoid | Avoid |
| *character* | Avoid |  |  |  |  | Avoid |
| *enum* | Avoid | Avoid | Avoid\* | Avoid | Avoid | Avoid |
| *signed* | Avoid |  |  |  |  |  |
| *unsigned* | Avoid |  |  |  |  |  |
| *floating* | Avoid | Avoid |  |  |  |  |

\* *Note:* an enumerated type may be cast to an enumerated type provided that the cast is to the same

*essential enumerated* type. Such casts are redundant.

Casting from *void* to any other type is not permitted as it results in undefined behaviour. This is covered by [Rule 1.3](#_heading=h.1baon6m).

**Rationale**

An explicit cast may be introduced for legitimate functional reasons, for example:

* To change the type in which a subsequent arithmetic operation is performed;
* To truncate a value deliberately;
* To make a type conversion explicit in the interests of clarity. However, some explicit casts are considered inappropriate:
* In C99, the result of a cast or assignment to *\_Bool* is always 0 or 1. This is not necessarily the case when casting to another type which is defined as *essentially Boolean*;
* A cast to an *essentially enum type* may result in a value that does not lie within the set of enumeration constants for that type;
* A cast from *essentially Boolean* to any other type is unlikely to be meaningful;
* Converting between floating and character types is not meaningful as there is no precise mapping between the two representations.

**Exception**

An *integer constant expression* with the value 0 or 1 of either signedness may be cast to a type which is

defined as *essentially Boolean*. This allows the implementation of non-C99 Boolean models.

**Example**

( bool\_t ) false /\* Compliant - C99 'false' is essentially Boolean \*/ ( int32\_t ) 3U /\* Compliant \*/

( bool\_t ) 0 /\* Compliant - by exception \*/

|  |  |  |  |
| --- | --- | --- | --- |
| ( bool\_t ) 3U | /\* | Non-compliant | \*/ |
| ( int32\_t ) ena | /\* | Compliant | \*/ |
| ( enum enuma ) 3 | /\* | Non-compliant | \*/ |
| ( char ) enc | /\* | Compliant | \*/ |

**See also**

[Rule 10.3](#_heading=h.1l354xk), [Rule 10.8](#_heading=h.4anzqyu)

##### Composite operators and expressions

Some of the concerns mentioned in [Appendix C](#_heading=h.3sv78d1) can be avoided by restricting the implicit and explicit conversions that may be applied to non-trivial expressions. These include:

* + - * The confusion about the type in which integer expressions are evaluated, as this depends on the type of the operands after any integer promotion. The type of the result of an arithmetic operation depends on the implemented size of *int*;
      * The common misconception among programmers that the type in which a calculation is conducted is influenced by the type to which the result is assigned or cast. This false expectation may lead to unintended results.

In addition to the previous rules, the *essential type model* places further restrictions on expressions whose operands are *composite expressions*, as defined below.

The following are defined as *composite operators* in this document:

* + - * Multiplicative (\*, /, %)
      * Additive (binary +, binary -)
      * Bitwise (&, |, ^)
      * Shift (<<, >>)
      * Conditional (?:) if either the second or third operand is a *composite expression*

A compound assignment is equivalent to an assignment of the result of its corresponding *composite operator*.

A *composite expression* is defined in this document as a non-*constant expression* which is the direct result of a *composite operator*.

*Note:*

* + - * The result of a compound assignment operator is not a *composite expression*;
      * A parenthesized *composite expression* is also a *composite expression*;
      * A *constant expression* is not a *composite expression*.

|  |  |
| --- | --- |
| Rule 10.6 | The value of a *composite expression* shall not be assigned to an object with wider *essential type* |
|  | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

This rule covers the assigning operations described in [Rule 10.3](#_heading=h.1l354xk).

**Rationale**

The rationale is described in the introduction on *composite operators and expressions* (see [Section 8.10.3](#_heading=h.1qoc8b1)).

**Example**

The following are compliant:

u16c = u16a + u16b; /\* Same essential type \*/ u32a = ( uint32\_t ) u16a + u16b; /\* Cast causes addition in uint32\_t \*/

The following are non-compliant:

u32a = u16a + u16b; /\* Implicit conversion on assignment \*/ use\_uint32 ( u16a + u16b ); /\* Implicit conversion of fn argument \*/

**See also**

[Rule 10.3](#_heading=h.1l354xk), [Rule 10.7](#_heading=h.zdd80z), [Section 8.10.3](#_heading=h.1qoc8b1)

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

**Rationale**

The rationale is described in the introduction on *composite operators and expressions* (see [Section 8.10.3](#_heading=h.1qoc8b1)).

Restricting implicit conversions on *composite expressions* means that sequences of arithmetic operations within an expression must be conducted in exactly the same *essential type*. This reduces possible developer confusion.

*Note:* this does not imply that all operands in an expression are of the same *essential type*.

The expression u32a + u16b + u16c is compliant as both additions will notionally be performed in type uint32\_t. In this case only *non-composite expressions* are implicitly converted.

The expression (u16a + u16b) + u32c is non-compliant as the left addition is notionally performed in type uint16\_t and the right in type uint32\_t, requiring an implicit conversion of the *composite*

*expression* u16a + u16b to uint32\_t.

**Example**

The following are compliant:

|  |  |  |  |
| --- | --- | --- | --- |
| u32a \* u16a + u16b | /\* | No composite conversion | \*/ |
| ( u32a \* u16a ) + u16b | /\* | No composite conversion | \*/ |
| u32a \* ( ( uint32\_t ) u16a + u16b ) | /\* | Both operands of \* have |  |
|  | \* | same essential type | \*/ |
| u32a += ( u32b + u16b ) | /\* | No composite conversion | \*/ |

The following are non-compliant:

u32a \* ( u16a + u16b ) /\* Implicit conversion of ( u16a + u16b ) \*/ u32a += ( u16a + u16b ) /\* Implicit conversion of ( u16a + u16b ) \*/

**See also**

[Rule 10.4](#_heading=h.rjefff), [Rule 10.6](#_heading=h.zdd80z), [Section 8.10.3](#_heading=h.1qoc8b1)

|  |  |
| --- | --- |
| Rule 10.8 | The value of a *composite expression* shall not be cast to a different  *essential type category* or a wider *essential type* |
|  | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

The rationale is described in the introduction on *composite operators and expressions* (see [Section 8.10.3](#_heading=h.1qoc8b1)).

Casting to a wider type is not permitted as the result may vary between implementations. Consider the following:

( uint32\_t ) ( u16a + u16b );

On a 16-bit machine the addition will be performed in 16 bits with the result wrapping modulo-2 before it is cast to 32 bits. However, on a 32-bit machine the addition will take place in 32 bits and would preserve high-order bits that would have been lost on a 16-bit machine.

Casting to a narrower type with the same *essential type category* is acceptable as the explicit truncation of the result always leads to the same loss of information.

**Example**

( uint16\_t ) ( u32a + u32b ) /\* Compliant \*/ ( uint16\_t ) ( s32a + s32b ) /\* Non-compliant - different essential

* type category \*/ ( uint16\_t ) s32a /\* Compliant - s32a is not composite \*/ ( uint32\_t ) ( u16a + u16b ) /\* Non-compliant - cast to wider
* essential type \*/

**See also**

[Rule 10.5](#_heading=h.2k82xt6), [Section 8.10.3](#_heading=h.1qoc8b1)

### Pointer type conversions

Pointer types can be classified as follows:

* Pointer to object;
* Pointer to function;
* Pointer to incomplete;
* Pointer to void;
* A null pointer constant, i.e. the value 0, optionally cast to void \*.

The only conversions involving pointers that are permitted by The Standard are:

* A conversion from a pointer type into void;
* A conversion from a pointer type into an arithmetic type;
* A conversion from an arithmetic type into a pointer type;
* A conversion from one pointer type into another pointer type.

Although permitted by the language *constraints*, conversion between pointers and any arithmetic types other than integer types is undefined.

The following permitted pointer conversions do not require an explicit cast:

* A conversion from a pointer type into \_Bool (C99 only);
* A conversion from a null pointer constant into a pointer type;
* A conversion from a pointer type into a compatible pointer type provided that the destination type has all the type qualifiers of the source type;
* A conversion between a pointer to an object or incomplete type and void \*, or qualified version thereof, provided that the destination type has all the type qualifiers of the source type.

In C99, any implicit conversion that does not fall into this subset of pointer conversions violates a

*constraint* (C99 Sections 6.5.4 and 6.5.16.1).

In C90, any implicit conversion that does not fall into this subset of pointer conversions leads to undefined behaviour (C90 Sections 6.3.4 and 6.3.16.1).

The conversion between pointer types and integer types is implementation-defined.

|  |  |
| --- | --- |
| Rule 11.1 | Conversions shall not be performed between a pointer to a function and any other type |
| C90 [Undefined 24, 27–29], C99 [Undefined 21, 23, 39, 41] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

A pointer to a function shall only be converted into or from a pointer to a function with a compatible

type.

**Rationale**

The conversion of a pointer to a function into or from any of:

* Pointer to object;
* Pointer to incomplete;
* void \*

results in undefined behaviour.

If a function is called by means of a pointer whose type is not compatible with the called function, the behaviour is undefined. Conversion of a pointer to a function into a pointer to a function with a different type is permitted by The Standard. Conversion of an integer into a pointer to a function is also permitted by The Standard. However, both are prohibited by this rule in order to avoid the undefined behaviour that would result from calling a function using an incompatible pointer type.

**Exception**

1. A null pointer constant may be converted into a pointer to a function;
2. A pointer to a function may be converted into void;
3. A function type may be implicitly converted into a pointer to that function type.

*Note:* exception 3 covers the implicit conversions described in C90 Section 6.2.2.1 and C99 Section 6.3.2.1. These conversions commonly occur when:

* A function is called directly, i.e. using a function identifier to denote the function to be called;
* A function is assigned to a function pointer.

**Example**

typedef void ( \*fp16 ) ( int16\_t n ); typedef void ( \*fp32 ) ( int32\_t n );

#include <stdlib.h> /\* To obtain macro NULL \*/

fp16 fp1 = NULL; /\* Compliant - exception 1 \*/

fp32 fp2 = ( fp32 ) fp1; /\* Non-compliant - function

* pointer into different
* function pointer \*/

if ( fp2 != NULL ) /\* Compliant - exception 1 \*/

{

}

fp16 fp3 = ( fp16 ) 0x8000; /\* Non-compliant - integer into

* function pointer \*/

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| fp16 | fp4 | = ( fp16 ) | 1.0e6F; | /\* | Non-compliant - float into |  |
|  |  |  |  | \* | function pointer | \*/ |

In the following example, the function call returns a pointer to a function type. Casting the return value into *void* is compliant with this rule.

typedef fp16 ( \*pfp16 ) ( void );

pfp16 pfp1;

( void ) ( \*pfp1 ( ) ); /\* Compliant - exception 2 - cast function

\* pointer into void \*/

The following examples show compliant implicit conversions from a function type into a pointer to that function type.

extern void f ( int16\_t n );

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| f ( 1 ); |  | /\* | Compliant | - exception 3 - implicit | conversion |  |
|  |  | \* | of f into | pointer to function |  | \*/ |
| fp16 fp5 = | f; | /\* | Compliant | - exception 3 |  | \*/ |
|  |  |  |  |  |  |  |

|  |  |
| --- | --- |
| Rule 11.2 | Conversions shall not be performed between a pointer to an incomplete type and any other type |
| C90 [Undefined 29], C99 [Undefined 21, 22, 41] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

A pointer to an incomplete type shall not be converted into another type. A conversion shall not be made into a pointer to incomplete type.

Although a pointer to *void* is also a pointer to an incomplete type, this rule does not apply to pointers to *void* as they are covered by [Rule 11.5](#_heading=h.338fx5o).

**Rationale**

Conversion into or from a pointer to an incomplete type may result in a pointer that is not correctly aligned, resulting in undefined behaviour.

Conversion of a pointer to an incomplete type into or from a floating type always results in undefined behaviour.

Pointers to an incomplete type are sometimes used to hide the representation of an object. Converting a pointer to an incomplete type into a pointer to object would break this encapsulation.

**Exception**

1. A null pointer constant may be converted into a pointer to an incomplete type.
2. A pointer to an incomplete type may be converted into void.

**Example**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| struct | s; |  | /\* | Incomplete type |  | \*/ |
| struct | t; |  | /\* | A different incomplete | type | \*/ |
| struct | s | \*sp; |  | | | |
| struct | t | \*tp; |
| int16\_t |  | \*ip; |

#include <stdlib.h> /\* To obtain macro NULL \*/

ip = ( int16\_t \* ) sp; /\* Non-compliant \*/

sp = ( struct s \* ) 1234; /\* Non-compliant \*/ tp = ( struct t \* ) sp; /\* Non-compliant - casting pointer into a

\* different incomplete type \*/

sp = NULL; /\* Compliant - exception 1 \*/

struct s \*f ( void );

( void ) f ( ); /\* Compliant - exception 2 \*/

**See also**

[Rule 11.5](#_heading=h.338fx5o)

|  |  |
| --- | --- |
| Rule 11.3 | A cast shall not be performed between a pointer to object type and a pointer to a different object type |
| C90 [Undefined 20], C99 [Undefined 22, 34] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

This rule applies to the unqualified types that are pointed to by the pointers.

**Rationale**

Casting a pointer to object into a pointer to a different object may result in a pointer that is not correctly aligned, resulting in undefined behaviour.

Even if conversion is known to produce a pointer that is correctly aligned, the behaviour may be undefined if that pointer is used to access an object. For example, if an object whose type is *int* is accessed as a *short* the behaviour is undefined even if *int* and *short* have the same representation and alignment requirements. See C90 Section 6.3, C99 Section 6.5, paragraph 7 for details.

**Exception**

It is permitted to convert a pointer to object type into a pointer to one of the object types *char*, *signed char* or *unsigned char*. The Standard guarantees that pointers to these types can be used to access the individual bytes of an object.

**Example**

uint8\_t \*p1; uint32\_t \*p2;

/\* Non-compliant - possible incompatible alignment \*/ p2 = ( uint32\_t \* ) p1;

extern uint32\_t read\_value ( void ); extern void print ( uint32\_t n );

void f ( void )

{

uint32\_t u = read\_value ( );

uint16\_t \*hi\_p = ( uint16\_t \* ) &u; /\* Non-compliant even though

\* probably correctly aligned \*/

\*hi\_p = 0; /\* Attempt to clear high 16-bits on big-endian machine \*/ print ( u ); /\* Line above may appear not to have been performed \*/

}

The following example is compliant because the rule applies to the unqualified pointer types. It does not prevent type qualifiers from being added to the object type.

const short \*p;

const volatile short \*q;

q = ( const volatile short \* ) p; /\* Compliant \*/

The following example is non-compliant because the unqualified pointer types are different, namely “pointer to *const-*qualified *int*” and “pointer to *int*”.

int \* const \* pcpi;

const int \* const \* pcpci;

pcpci = ( const int \* const \* ) pcpi;

**See also**

[Rule 11.4](#_heading=h.j8sehv), [Rule 11.5](#_heading=h.338fx5o), [Rule 11.8](#_heading=h.1idq7dh)

Rule 11.4

A conversion should not be performed between a pointer to object and an integer type

Category Advisory

Analysis Decidable, Single Translation Unit

Applies to C90, C99

C90 [Undefined 20; Implementation 24]

C99 [Undefined 21, 34; Implementation J.3.7(1)]

|  |  |
| --- | --- |
|  |  |
|  | |
|  |  |
|  |  |
|  |  |

**Amplification**

A pointer should not be converted into an integer. An integer should not be converted into a pointer.

**Rationale**

Conversion of an integer into a pointer to object may result in a pointer that is not correctly aligned, resulting in undefined behaviour.

Conversion of a pointer to object into an integer may produce a value that cannot be represented in the chosen integer type resulting in undefined behaviour.

*Note:* the C99 types *intptr\_t* and *uintptr\_t*, declared in <stdint.h>, are respectively signed and unsigned integer types capable of representing pointer values. Despite this, conversions between a pointer to object and these types is not permitted by this rule because their use does not avoid the undefined behaviour associated with misaligned pointers.

Casting between a pointer and an integer type should be avoided where possible, but may be necessary when addressing memory mapped registers or other hardware specific features. If casting between integers and pointers is used, care should be taken to ensure that any pointers produced do not give rise to the undefined behaviour discussed under [Rule 11.3](#_heading=h.243i4a2).

**Exception**

A *null pointer constant* that has integer type may be converted into a pointer to object.

Example

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | | | | | |
| uint8\_t \*PORTA | = | ( uint8\_t | \* ) 0x0002; | /\* | Non-compliant | \*/ |
| uint16\_t \*p; |  |  |  |  |  |  |
| int32\_t addr | = | ( int32\_t | ) &p; | /\* | Non-compliant | \*/ |
| uint8\_t \*q | = | ( uint8\_t | \* ) addr; | /\* | Non-compliant | \*/ |

bool\_t b = ( bool\_t ) p; /\* Non-compliant \*/ enum etag { A, B } e = ( enum etag ) p; /\* Non-compliant \*/

**See also**

[Rule 11.3](#_heading=h.243i4a2), [Rule 11.7](#_heading=h.3jd0qos), [Rule 11.9](#_heading=h.42ddq1a)

|  |  |
| --- | --- |
| Rule 11.5 | A conversion should not be performed from pointer to *void* into pointer to object |
| C90 [Undefined 20], C99 [Undefined 22, 34] | |
| Category | Advisory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

Conversion of a pointer to *void* into a pointer to object may result in a pointer that is not correctly aligned, resulting in undefined behaviour. It should be avoided where possible but may be necessary, for example when dealing with memory allocation functions. If conversion from a pointer to object into a pointer to *void* is used, care should be taken to ensure that any pointers produced do not give rise to the undefined behaviour discussed under [Rule 11.3](#_heading=h.243i4a2).

**Exception**

A *null pointer constant* that has type pointer to *void* may be converted into pointer to object.

**Example**

uint32\_t \*p32; void \*p; uint16\_t \*p16;

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| p | = | p32; |  | /\* | Compliant - pointer | to | uint32\_t | into |  |
|  |  |  |  | \* | pointer to void |  |  |  | \*/ |
| p16 | = | p; |  | /\* | Non-compliant |  |  |  | \*/ |
| p | = | ( void | \* ) p16; | /\* | Compliant |  |  |  | \*/ |

p32 = ( uint32\_t \* ) p; /\* Non-compliant \*/

**See also**

[Rule 11.2](#_heading=h.3oy7u29), [Rule 11.3](#_heading=h.243i4a2)

Rule 11.6

A cast shall not be performed between pointer to *void* and an arithmetic type

Category Required

Analysis Decidable, Single Translation Unit

Applies to C90, C99

C90 [Undefined 29; Implementation 24]

C99 [Undefined 21, 41; Implementation J.3.7(1)]

Section 8: Rules

|  |  |
| --- | --- |
|  |  |
|  | |
|  |  |
|  |  |
|  |  |

**Rationale**

Conversion of an integer into a pointer to *void* may result in a pointer that is not correctly aligned, resulting in undefined behaviour.

Conversion of a pointer to *void* into an integer may produce a value that cannot be represented in the chosen integer type resulting in undefined behaviour.

Conversion between any non-integer arithmetic type and pointer to *void* is undefined.

**Exception**

An *integer constant expression* with value 0 may be cast into pointer to *void*.

**Example**

void \*p; uint32\_t u;

/\* Non-compliant - implementation-defined \*/ p = ( void \* ) 0x1234u;

/\* Non-compliant - undefined \*/ p = ( void \* ) 1024.0f;

/\* Non-compliant - implementation-defined \*/ u = ( uint32\_t ) p;

Rule 11.7

A cast shall not be performed between pointer to object and a non- integer arithmetic type

Category Required

Analysis Decidable, Single Translation Unit

Applies to C90, C99

C90 [Undefined 29; Implementation 24]

C99 [Undefined 21, 41; Implementation J.3.7(1)]

**Amplification**

For the purposes of this rule a non-integer arithmetic type means one of:

* *Essentially Boolean*;
* *Essentially character*;
* *Essentially enum*;
* *Essentially floating*.

**Rationale**

Conversion of an *essentially Boolean*, *essentially character* or *essentially enum* type into a pointer to object may result in a pointer that is not correctly aligned, resulting in undefined behaviour.

Conversion of a pointer to object into an *essentially Boolean*, *essentially character* or *essentially enum* type may produce a value that cannot be represented in the chosen integer type resulting in undefined behaviour.

Conversion of a pointer to object into or from an *essentially floating* type results in undefined behaviour.

**Example**

int16\_t \*p; float32\_t f;

f = ( float32\_t ) p; /\* Non-compliant \*/ p = ( int16\_t \* ) f; /\* Non-compliant \*/

**See also**

[Rule 11.4](#_heading=h.j8sehv)

|  |  |
| --- | --- |
| Rule 11.8 | A cast shall not remove any *const* or *volatile* qualification from the type pointed to by a pointer |
| C90 [Undefined 12, 39, 40], C99 [Undefined 30, 61, 62] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

Any attempt to remove the qualification associated with the addressed type by using casting is a violation of the principle of type qualification.

*Note:* the qualification referred to here is not the same as any qualification that may be applied to the pointer itself.

Some of the problems that might arise if a qualifier is removed from the addressed object are:

* Removing a *const* qualifier might circumvent the read-only status of an object and result in it being modified;
* Removing a *const* qualifier might result in an exception when the object is accessed;
* Removing a *volatile* qualifier might result in accesses to the object being optimized away.

*Note:* removal of the C99 *restrict* type qualifier is benign.

**Example**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| uint16\_t |  | x; |  | | | |
| uint16\_t | \* const | cpi = | &x; | /\* | const pointer | \*/ |
| uint16\_t | \* const | \*pcpi; |  | /\* | pointer to const pointer | \*/ |
| uint16\_t | \* | \*ppi; |  |  |  |  |
| const uint16\_t |  | \*pci; |  | /\* | pointer to const | \*/ |
| volatile uint16\_t |  | \*pvi; |  | /\* | pointer to volatile | \*/ |
| uint16\_t |  | \*pi; |  |  |  |  |

pi = cpi; /\* Compliant - no conversion

no cast required \*/

pi = (uint16\_t \*)pci; /\* Non-compliant \*/

pi = (uint16\_t \*)pvi; /\* Non-compliant \*/

ppi = (uint16\_t \* \*)pcpi; /\* Non-compliant \*/

##### See also

[Rule 11.3](#_heading=h.243i4a2)

Rule 11.9

The macro NULL shall be the only permitted form of integer *null pointer constant*

Category Required

Analysis Decidable, Single Translation Unit

Applies to C90, C99

|  |  |
| --- | --- |
|  |  |
|  | |
|  |  |
|  |  |
|  |  |

**Amplification**

An *integer constant expression* with the value 0 shall be derived from expansion of the macro NULL if it appears in any of the following contexts:

* As the value being *assigned* to a pointer;
* As an operand of an == or != operator whose other operand is a pointer;
* As the second operand of a ?: operator whose third operand is a pointer;
* As the third operand of a ?: operator whose second operand is a pointer.

Ignoring whitespace and any surrounding parentheses, any such *integer constant expression* shall represent the entire expansion of NULL.

*Note:* a *null pointer constant* of the form (void \*)0 is permitted, whether or not it was expanded from

NULL.

**Rationale**

Using NULL rather than 0 makes it clear that a *null pointer constant* was intended.

**Example**

In the following example, the initialization of p2 is compliant because the *integer constant expression* 0 does not appear in one of the contexts prohibited by this rule.

int32\_t \*p1 = 0; /\* Non-compliant \*/

int32\_t \*p2 = ( void \* ) 0; /\* Compliant \*/

In the following example, the comparison between p2 and (void \*)0 is compliant because the *integer constant expression* 0 appears as the operand of a cast and not in one of the contexts prohibited by this rule.

#define MY\_NULL\_1 0

#define MY\_NULL\_2 ( void \* ) 0

if ( p1 == MY\_NULL\_1 ) /\* Non-compliant \*/

{

}

if ( p2 == MY\_NULL\_2 ) /\* Compliant \*/

{

}

The following example is compliant because use of the macro NULL provided by the implementation is always permitted, even if it expands to an *integer constant expression* with value 0.

/\* Could also be stdio.h, stdlib.h and others \*/ #include <stddef.h>

extern void f ( uint8\_t \*p );

/\* Compliant for any conforming definition of NULL, such as:

\* 0

* ( void \* ) 0
* ( ( ( 0 ) ) )
* ( ( ( 1 - 1 ) ) )

\*/

f ( NULL );

**See also**

[Rule 11.4](#_heading=h.j8sehv)

### Expressions

Section 8: Rules

|  |  |
| --- | --- |
| Rule 12.1 | The precedence of operators within expressions should be made explicit |
|  | |
| Category | Advisory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

The following table is used in the definition of this rule.

|  |  |  |
| --- | --- | --- |
| Description | Operator or Operand | Precedence |
| Primary | identifier, constant, string literal, ( expression ) | 16 (high) |
| Postfix | [] () (function call) . -> ++ (post-increment) -- (post-decrement) ()  {} (C99: compound literal) | 15 |
| Unary | ++ (pre-increment) -- (pre-decrement) & \* + - ~ ! *sizeof defined*  (preprocessor) | 14 |
| Cast | () | 13 |
| Multiplicative | \* / % | 12 |
| Additive | + - | 11 |
| Bitwise shift | << >> | 10 |
| Relational | < > <= >= | 9 |
| Equality | == != | 8 |
| Bitwise AND | & | 7 |
| Bitwise XOR | ^ | 6 |
| Bitwise OR | | | 5 |
| Logical AND | && | 4 |
| Logical OR | || | 3 |
| Conditional | ?: | 2 |
| Assignment | = \*= /= %= += -= <<= >>= &= ^= |= | 1 |
| Comma | , | 0 (low) |

The precedences used in this table are chosen to allow a concise description of the rule. They are not necessarily the same as those that might be encountered in other descriptions of operator precedence.

For the purposes of this rule, the precedence of an expression is the precedence of the element (operand or operator) at the root of the parse tree for that expression.

For example: the parse tree for the expression a << b + c can be represented as:

<<

/ \

a +

/ \

b c

The element at the root of this parse tree is '<<' so the expression has precedence 10.

The following advice is given:

* The operand of the *sizeof* operator should be enclosed in parentheses;
* An expression whose precedence is in the range 2 to 12 should have parentheses around any operand that has both:
  + Precedence of less than 13, and
  + Precedence greater than the precedence of the expression.

**Rationale**

The C language has a relatively large number of operators and their relative precedences are not intuitive. This can lead less experienced programmers to make mistakes. Using parentheses to make operator precedence explicit removes the possibility that the programmer’s expectations are incorrect. It also makes the original programmer’s intention clear to reviewers or maintainers of the code.

It is recognized that overuse of parentheses can clutter the code and reduce its readability. This rule aims to achieve a compromise between code that is hard to understand because it contains either too many or too few parentheses.

**Examples**

The following example shows expressions with a unary or postfix operator whose operands are either

*primary-expressions* or expressions whose top-level operators have precedence 15.

a[ i ]->n; /\* Compliant - no need to write ( a[ i ] )->n \*/

\*p++; /\* Compliant - no need to write \*( p++ ) \*/ sizeof x + y; /\* Non-compliant - write either sizeof ( x ) + y

\* or sizeof ( x + y ) \*/

The following example shows expressions containing operators at the same precedence level. All of these are compliant but, depending on the types of a, b and c, any expression with more than one operator may violate other rules.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| a a ( | +  +  a | b; b  + | +  b | c;  ) | + | c; |
| a | + | ( | b | + | c | ); |
| a | + | b | - | c | + | d; |
| ( | a | + | b | ) | - | ( c + d ); |

The following example shows a variety of mixed-operator expressions:

/\* Compliant - no need to write f ( ( a + b ), c ) \*/ x = f ( a + b, c );

/\* Non-compliant

* Operands of conditional operator (precedence 2) are:

\* == precedence 8 needs parentheses

* a precedence 16 does not need parentheses
* - precedence 11 needs parentheses

\*/

x = a == b ? a : a - b;

/\* Compliant \*/

x = ( a == b ) ? a : ( a - b );

/\* Compliant

* Operands of << operator (precedence 10) are:
* a precedence 16 does not need parentheses
* ( E ) precedence 16 already parenthesized

\*/

x = a << ( b + c );

/\* Compliant

* Operands of && operator (precedence 4) are:
* a precedence 16 does not need parentheses
* && precedence 4 does not need parentheses

\*/

if ( a && b && c )

{

}

/\* Compliant

* Operands of && operator (precedence 4) are:
* defined(X) precedence 14 does not need parentheses
* (E) precedence 16 already parenthesized

\*/

#if defined ( X ) && ( ( X + Y ) > Z )

/\* Compliant

* Operands of && operator (precedence 4) are:
* !defined ( X ) precedence 14 does not need parentheses
* defined ( Y ) precedence 14 does not need parentheses
* Operand of ! operator (precedence 14) is:
* defined ( X ) precedence 14 does not need parentheses

\*/

#if !defined ( X ) && defined ( Y )

*Note:* this rule does not require the operands of a , operator to be parenthesized. Use of the ,

operator is prohibited by [Rule 12.3](#_heading=h.1vsw3ci).

x = a, b; /\* Compliant - parsed as ( x = a ), b \*/

Rule 12.2

The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the *essential type* of the left hand operand

C90 [Undefined 32], C99 [Undefined 48]

|  |  |
| --- | --- |
|  |  |
|  | |
| Category | Required |
| Analysis | Undecidable, System |
| Applies to | C90, C99 |

**Rationale**

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

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 in the range 0 to 15.

See [Section 8.10](#_heading=h.3x8tuzt) for a description of *essential type* and the limitations on the *essential types* for the operands of shift operators.

There are various ways of ensuring this rule is followed. The simplest is for the right hand operand to be a constant (whose value can then be statically checked). Use of an unsigned integer type will ensure

that the operand is non-negative, so then only the upper limit needs to be checked (dynamically at run time or by review). Otherwise both limits will need to be checked.

**Example**

u8a = u8a << 7; /\* Compliant \*/

u8a = u8a << 8; /\* Non-compliant \*/ u16a = ( uint16\_t ) u8a << 9; /\* Compliant \*/

To assist in understanding the following examples, it should be noted that the *essential type* of 1u is

*essentially unsigned char*, whereas the *essential type* of 1UL is *essentially unsigned long*.

1u << 10u; /\* Non-compliant \*/ ( uint16\_t ) 1u << 10u; /\* Compliant \*/ 1UL << 10u; /\* Compliant \*/

|  |  |
| --- | --- |
| Rule 12.3 | The comma operator should not be used |
|  | |
| Category | Advisory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

Use of the comma operator is generally detrimental to the readability of code, and the same effect can usually be achieved by other means.

**Example**

f ( ( 1, 2 ), 3 ); /\* Non-compliant - how many parameters? \*/

The following example is non-compliant with this rule and other rules:

for ( i = 0, p = &a[ 0 ]; i < N; ++i, ++p )

{

}

|  |  |
| --- | --- |
| Rule 12.4 | Evaluation of *constant expressions* should not lead to unsigned integer wrap-around |
|  | |
| Category | Advisory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

This rule applies to expressions that satisfy the *constraints* for a *constant expression*, whether or not they appear in a context that requires a *constant expression*.

If an expression is not evaluated, for example because it appears in the right operand of a logical AND operator whose left operand is always false, then this rule does not apply.

**Rationale**

Unsigned integer expressions do not strictly overflow, but instead wrap-around. Although there may be good reasons to use modulo arithmetic at run-time, it is less likely for its use to be intentional at compile-time.

**Example**

The expression associated with a *case* label is required to be a *constant expression*. If an unsigned wrap- around occurs during evaluation of a *case* expression, it is likely to be unintentional. On a machine with a 16-bit *int* type, any value of BASE greater than or equal to 65 024 would result in wrap-around in the following example:

#define BASE 65024u

switch ( x )

{

case BASE + 0u: f ( );

break;

case BASE + 1u: g ( );

break;

case BASE + 512u: /\* Non-compliant - wraps to 0 \*/ h ( );

break;

}

The controlling expression of a *#if* or *#elif* preprocessor directive is required to be a *constant expression*.

#if 1u + ( 0u - 10u ) /\* Non-compliant as ( 0u - 10u ) wraps \*/

In this example, the expression DELAY + WIDTH has the value 70 000 but this will wrap-around to 4 464 on a machine with a 16-bit *int* type.

#define DELAY 10000u #define WIDTH 60000u

void fixed\_pulse ( void )

{

uint16\_t off\_time16 = DELAY + WIDTH; /\* Non-compliant \*/

}

This rule does not apply to the expression c + 1 in the following compliant example as it accesses an object and therefore does not satisfy the *constraints* for a *constant expression*:

const uint16\_t c = 0xffffu; void f ( void )

{

uint16\_t y = c + 1u; /\* Compliant \*/

}

In the following example, the sub-expression ( 0u - 1u ) leads to unsigned integer wrap-around. In the initialization of x, the sub-expression is not evaluated and the expression is therefore compliant. However, in the initialization of y, it may be evaluated and the expression is therefore non-compliant.

bool\_t b;

void g ( void )

{

uint16\_t x = ( 0u == 0u ) ? 0u : ( 0u - 1u ); /\* Compliant \*/ uint16\_t y = b ? 0u : ( 0u - 1u ); /\* Non-compliant \*/

}

### Side effects

|  |  |
| --- | --- |
| Rule 13.1 | *Initializer lists* shall not contain *persistent side effects* |
| C99 [Unspecified 17, 22] | |
| Category | Required |
| Analysis | Undecidable, System |
| Applies to | C99 |

**Rationale**

C90 *constrains* the initializers for automatic objects with aggregate types to contain only *constant expressions*. However, C99 permits automatic aggregate initializers to contain expressions that are evaluated at run-time. It also permits compound literals which behave as anonymous initialized objects. The order in which *side effects* occur during evaluation of the expressions in an *initializer list* is unspecified and the behaviour of the initialization is therefore unpredictable if those *side effects* are *persistent*.

**Example**

volatile uint16\_t v1; void f ( void )

{

/\* Non-compliant - volatile access is persistent side effect \*/ uint16\_t a[ 2 ] = { v1, 0 };

}

void g ( uint16\_t x, uint16\_t y )

{

/\* Compliant - no side effects \*/ uint16\_t a[ 2 ] = { x + y, x - y };

}

uint16\_t x = 0u;

extern void p ( uint16\_t a[ 2 ] ); void h ( void )

{

/\* Non-compliant - two side effects \*/ p ( ( uint16\_t[ 2 ] ) { x++, x++ } );

}

**See also**

[Rule 13.2](#_heading=h.2f3j2rp)

|  |  |
| --- | --- |
| Rule 13.2 | The value of an expression and its *persistent side effects* shall be the same under all permitted evaluation orders |
| C90 [Unspecified 7–9; Undefined 18], C99 [Unspecified 15–18; Undefined 32] | |
| Category | Required |
| Analysis | Undecidable, System |
| Applies to | C90, C99 |

Amplification

Between any two adjacent sequence points or within any full expression:

1. No object shall be modified more than once;
2. No object shall be both modified and read unless any such read of the object’s value contributes towards computing the value to be stored into the object;
3. There shall be no more than one modification access with volatile-qualified type;
4. There shall be no more than one read access with volatile-qualified type.

*Note:* An object might be accessed indirectly, by means of a pointer or a called function, as well as being accessed directly by the expression.

*Note:* This Amplification is intentionally stricter than the headline of the rule. As a result, expressions such as:

x = x = 0;

are not permitted by this rule

even though the value and the *persistent side effects*, provided that x is not *volatile*, are independent of the order of evaluation or *side effects*.

Sequence points are summarized in Annex C of both the C90 and C99 standards. The sequence points in C90 are a subset of those in C99.

Full expressions are defined in Section 6.6 of the C90 standard and Section 6.8 of the C99 standard.

**Rationale**

The Standard gives considerable flexibility to compilers when evaluating expressions. Most operators can have their operands evaluated in any order. The main exceptions are:

* Logical AND && in which the second operand is evaluated only if the first operand evaluates to non-zero;
* Logical OR || in which the second operand is evaluated only if the first operand evaluates to zero;
* The conditional operator ?: in which the first operand is always evaluated and then either the second or third operand is evaluated;
* The , operator in which the first operand is evaluated and then the second operand is evaluated.

*Note:* The presence of parentheses may alter the order in which operators are applied. However, this does not affect the order of evaluation of the lowest-level operands, which may be evaluated in any order.

Many of the common instances of the unpredictable behaviour associated with expression evaluation can be avoided by following the advice given by [Rule 13.3](#_heading=h.1yib0wl) and [Rule 13.4](#_heading=h.1a346fx).

**Examples**

When the COPY\_ELEMENT macro is invoked in this non-compliant example, i is read twice and modified twice. It is unspecified whether the order of operations on i is:

* Read, modify, read, modify, or
* Read, read, modify, modify.

#define COPY\_ELEMENT ( index ) ( a[( index )] = b[( index )] )

COPY\_ELEMENT ( i++ );

In this non-compliant example the order in which v1 and v2 are read is unspecified.

extern volatile uint16\_t v1, v2; uint16\_t t;

t = v1 + v2;

In this compliant example PORT is read and modified.

extern volatile uint8\_t PORT; PORT = PORT & 0x80u;

The order of evaluation of function arguments is unspecified as is the order in which *side effects* occur as shown in this non-compliant example.

uint16\_t i = 0;

/\*

* Unspecified whether this call is equivalent to:

\* f ( 0, 0 )

* or f ( 0, 1 )

\*/

f ( i++, i );

The relative order of evaluation of a function designator and function arguments is unspecified. In this non-compliant example, if the call to g modifies p then it is unspecified whether the function designator p->f uses the value of p prior to the call of g or after it.

p->f ( g ( &p ) );

**See also**

[Dir 4.9](#_heading=h.1jlao46), [Rule 13.1](#_heading=h.2uxtw84), [Rule 13.3](#_heading=h.1yib0wl), [Rule 13.4](#_heading=h.1a346fx)

|  |  |
| --- | --- |
| 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 |
| C90 [Unspecified 7, 8; Undefined 18], C99 [Unspecified 15; Undefined 32] | |
| Category | Advisory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

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

All sub-expressions of the full expression are treated as if they were evaluated for the purposes of this rule, even if specified as not being evaluated by The Standard.

**Rationale**

The use of increment and decrement operators in combination with other operators is not recommended because:

* It can significantly impair the readability of the code;
* It introduces additional *side effects* into a statement with the potential for undefined behaviour (covered by [Rule 13.2](#_heading=h.2f3j2rp)).

It is clearer to use these operations in isolation from any other operators.

**Example**

The expression:

u8a = u8b++

is non-compliant. The non-compliant expression statement:

u8a = ++u8b + u8c--;

is clearer when written as the following sequence:

++u8b;

u8a = u8b + u8c; u8c--;

The following are all compliant because the only *side effect* in each expression is caused by the increment or decrement operator.

x++;

a[ i ]++;

b.x++; c->x++;

++( \*p );

\*p++;

( \*p )++;

The following are all non-compliant because they contain a function call as well as an increment or decrement operator:

if ( ( f ( ) + --u8a ) == 0u )

{

}

g ( u8b++ );

The following are all non-compliant even though the sub-expression containing the increment or decrement operator or some other *side effect* is not evaluated:

u8a = ( 1u == 1u ) ? 0u : u8b++;

if ( u8a++ == ( ( 1u == 1u ) ? 0u : f ( ) ) )

{

}

**See also**

[Rule 13.2](#_heading=h.2f3j2rp)

|  |  |
| --- | --- |
| Rule 13.4 | The result of an assignment operator should not be *used* |
| C90 [Unspecified 7, 8; Undefined 18], C99 [Unspecified 15, 18; Undefined 32]  [Koenig 6] | |
| Category | Advisory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

This rule applies even if the expression containing the assignment operator is not evaluated.

**Rationale**

The use of assignment operators, simple or compound, in combination with other arithmetic operators is not recommended because:

* It can significantly impair the readability of the code;
* It introduces additional *side effects* into a statement making it more difficult to avoid the undefined behaviour covered by [Rule 13.2](#_heading=h.2f3j2rp).

**Example**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | | | |
| x = y;  a[ x ] = a[ x = y ]; | /\*  /\*  \* | Compliant  Non-compliant is used | - the value of x = y | \*/  \*/ |

/\*

* Non-compliant - value of bool\_var = false is used but
* bool\_var == false was probably intended

\*/

if ( bool\_var = false )

{

}

### 

/\* Non-compliant even though bool\_var = true isn't evaluated \*/ if ( ( 0u == 0u ) || ( bool\_var = true ) )

{

}

/\* Non-compliant - value of x = f() is used \*/ if ( ( x = f ( ) ) != 0 )

{

}

/\* Non-compliant - value of b += c is used \*/ a[ b += c ] = a[ b ];

/\* Non-compliant - values of c = 0 and b = c = 0 are used \*/ a = b = c = 0;

**See also**

[Rule 13.2](#_heading=h.2f3j2rp)

|  |  |
| --- | --- |
| Rule 13.5 | The right hand operand of a logical && or || operator shall not contain *persistent side effects* |
|  | |
| Category | Required |
| Analysis | Undecidable, System |
| Applies to | C90, C99 |

**Rationale**

The evaluation of the right-hand operand of the && and || operators is conditional on the value of the left-hand operand. If the right-hand operand contains *side effects* then those *side effects* may or may not occur which may be contrary to programmer expectations.

If evaluation of the right-hand operand would produce *side effects* which are not *persistent* at the point in the program where the expression occurs then it does not matter whether the right-hand operand is evaluated or not.

The term *persistent side effect* is defined in [Appendix](#_heading=h.gtnh0h) J.

**Example**

uint16\_t f ( uint16\_t y )

{

/\* These side effects are not persistent as seen by the caller \*/ uint16\_t temp = y;

temp = y + 0x8080U;

return temp;

}

uint16\_t h ( uint16\_t y )

{

static uint16\_t temp = 0;

/\* This side effect is persistent \*/ temp = y + temp;

return temp;

}

void g ( void )

{

/\* Compliant - f ( ) has no persistent side effects \*/ if ( ishigh && ( a == f ( x ) ) )

{

}

/\* Non-compliant - h ( ) has a persistent side effect \*/ if ( ishigh && ( a == h ( x ) ) )

{

}

}

volatile uint16\_t v;

uint16\_t x;

/\* Non-compliant - access to volatile v is persistent \*/ if ( ( x == 0u ) || ( v == 1u ) )

{

}

/\* Non-compliant if fp points to a function with persistent side effects \*/ ( fp != NULL ) && ( \*fp ) ( 0 );

Rule 13.6

The operand of the *sizeof* operator shall not contain any expression which has potential *side effects*

Category Mandatory

Analysis Decidable, Single Translation Unit

Applies to C90, C99

C99 [Unspecified 21]

**Amplification**

Any expressions appearing

in the operand of a *sizeof* operator are not normally evaluated. This rule mandates that the evaluation of any such expression shall not contain *side effects*, whether or not it is actually evaluated.

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

**Rationale**

The operand of a *sizeof* operator may be either an expression or may specify a type. If the operand contains an expression, a possible programming error is to expect that expression to be evaluated when it is actually not evaluated in most circumstances.

The C90 standard states that expressions appearing in the operand are not evaluated at run-time.

In C99, expressions appearing in the operand are usually not evaluated at run-time. However, if the operand contains a variable-length array type then the array size expression will be evaluated if

necessary. If the result can be determined without evaluating the array size expression then it is unspecified whether it is evaluated or not.

**Exception**

An expression of the form sizeof ( V ), where V is an *lvalue* with a *volatile* qualified type that is not a variable-length array, is permitted.

**Example**

volatile int32\_t i;

int32\_t j; size\_t s;

s = sizeof ( j ); /\* Compliant \*/ s = sizeof ( j++ ); /\* Non-compliant \*/ s = sizeof ( i ); /\* Compliant - exception \*/ s = sizeof ( int32\_t ); /\* Compliant \*/

In this example the final *sizeof* expression illustrates how it is possible for a variable-length array size expression to have no effect on the size of the type. The operand is the type “array of *n* pointers to function with parameter type array of v int32\_t”. Because the operand has variable-length array type, it is evaluated. However, the size of the array of *n* function pointers is unaffected by the parameter list for those function pointer types. Therefore, the *volatile*-qualified object v may or may not be evaluated and its *side effects* may or may not occur.

volatile uint32\_t v;

void f ( int32\_t n )

{

size\_t s;

s = sizeof ( int32\_t[ n ] ); /\* Compliant \*/

s = sizeof ( int32\_t[ n++ ] ); /\* Non-compliant \*/ s = sizeof ( void ( \*[ n ] ) ( int32\_t a[ v ] ) ); /\* Non-compliant \*/

}

**See also**

[Rule 18.8](#_heading=h.4ihyjke)

### Control statement expressions

The term *loop counter* is used by some of the rules in this section. A *loop counter* is defined as an object, array element or member of a structure or union which satisfies the following:

1. It has a scalar type;
2. Its value varies monotonically on each iteration of a given instance of a loop; and
3. It is involved in a decision to exit the loop.

*Note:* the second condition means that the value of the *loop counter* must change on each iteration of the loop and it must always change in the same direction for a given instance of the loop. It may, however, change in different directions on different instances, for example sometimes reading the elements of an array backwards and sometimes reading them forwards.

According to this definition, a loop need not have just one *loop counter*: it is possible for a loop to have no *loop counter* or to have more than one. See [Rule 14.2](#_heading=h.2xn8ts7) for further restrictions on *loop counters* in *for* loops.

The following code fragments show examples of loops and their corresponding *loop counters*.

In this loop, i is a *loop counter* because it has a scalar type, varies monotonically (increasing) and is involved in the loop termination condition.

for ( uint16\_t i = 0; a[ i ] < 10; ++i )

{

}

The following loop has no *loop counter*. The object count has scalar type and varies monotonically but is not involved in the termination condition.

extern volatile bool\_t b; uint16\_t count = 0;

while ( b )

{

count = count + 1U;

}

In the following code, both i and sum are scalar and monotonically varying (decreasing and increasing respectively). However sum is not a *loop counter* because it is not involved in the decision to exit the loop.

uint16\_t sum = 0;

for ( uint16\_t i = 10U; i != 0U; --i )

{

sum += i;

}

In the following loop, p is a *loop counter*. It is not involved in the loop control expression but it is involved in a decision to exit the loop by means of the *break* statement.

extern volatile bool\_t b; extern char \*p;

do

{

if ( \*p == '\0' )

{

break;

}

++p;

} while ( b );

The *loop counter* in the following example is p->count.

struct s

{

uint16\_t count; uint16\_t a[ 10 ];

};

extern struct s \*p;

for ( p->count = 0U; p->count < 10U; ++( p->count ) )

{

1. >a[ p->count ] = 0U;

}

|  |  |
| --- | --- |
| Rule 14.1 | A *loop counter* shall not have *essentially floating* type |
|  | |
| Category | Required |
| Analysis | Undecidable, System |
| Applies to | C90, C99 |

**Rationale**

When using a floating-point *loop counter*, accumulation of rounding errors may result in a mismatch between the expected and actual number of iterations. This can happen when a loop step that is not a power of the floating-point radix is rounded to a value that can be represented.

Even if a loop with a floating-point *loop counter* appears to behave correctly on one implementation, it may give a different number of iterations on another implementation.

**Example**

In the following non-compliant example, the value of counter is unlikely to be 1 000 at the end of the loop.

uint32\_t counter = 0u;

for ( float32\_t f = 0.0f; f < 1.0f; f += 0.001f )

{

++counter;

}

The following compliant example uses an integer *loop counter* to guarantee 1 000 iterations and uses it to generate f for use within the loop.

float32\_t f;

for ( uint32\_t counter = 0u; counter < 1000u; ++counter )

{

f = ( float32\_t ) counter \* 0.001f;

}

The following *while* loop is non-compliant because f is being used as a *loop counter*.

float32\_t f = 0.0f; while ( f < 1.0f )

{

f += 0.001f;

}

The following *while* loop is compliant because f is not being used as a *loop counter*.

float32\_t f; uint32\_t u32a;

f = read\_float32 ( ); do

{u32a = read\_u32 ( );

/\* f does not change in the loop so cannot be a loop counter \*/

} while ( ( ( float32\_t ) u32a - f ) > 10.0f );

**See also**

[Rule 14.2](#_heading=h.2xn8ts7)

|  |  |
| --- | --- |
| Rule 14.2 | A *for* loop shall be well-formed |
|  | |
| Category | Required |
| Analysis | Undecidable, System |
| Applies to | C90, C99 |

**Amplification**

The three clauses of a *for* statement are the:

*First clause* which

* Shall be empty, or
* Shall assign a value to the loop counter, or
* Shall define and initialize the loop counter (C99).

*Second clause* which

* Shall be an expression that has no persistent side effects, and
* Shall use the loop counter and optionally loop control flags, and
* Shall not use any other object that is modified in the for loop body.

*Third clause* which

* Shall be an expression whose only persistent side effect is to modify the value of the loop counter, and
* Shall not use objects that are modified in the for loop body.

There shall only be one *loop counter* in a *for* loop, which shall not be modified in the *for* loop body.

A *loop control flag* is defined as a single identifier denoting an object with *essentially Boolean type* that is used in the *Second clause*.

The behaviour of a *for* loop body includes the behaviour of any functions called within that statement.

**Rationale**

The *for* statement provides a general-purpose looping facility. Using a restricted form of loop makes code easier to review and to analyse.

**Exception**

All three clauses may be empty, for example for ( ; ; ), so as to allow for infinite loops.

**Example**

In the following C99 example, i is the *loop counter* and flag is a *loop control flag*.

bool\_t flag = false;

for ( int16\_t i = 0; ( i < 5 ) && !flag; i++ )

{

if ( C )

{

flag = true; /\* Compliant - allows early termination

* of loop \*/

}

i = i + 3; /\* Non-compliant - altering the loop

* counter \*/

}

**See also**

[Rule 14.1](#_heading=h.1csj400), [Rule 14.3](#_heading=h.1nia2ey), [Rule 14.4](#_heading=h.47hxl2r)

|  |  |
| --- | --- |
| Rule 14.3 | Controlling expressions shall not be invariant |
|  | |
| Category | Required |
| Analysis | Undecidable, System |
| Applies to | C90, C99 |

**Amplification**

This rule applies to:

* Controlling expressions of if, while, for, do … while and switch statements;
* The first operand of the ?: operator.

**Rationale**

If a controlling expression has an invariant value, it is possible that there is a programming error. Any code that cannot be reached due to the presence of an invariant expression may be removed by the compiler. This might have the effect of removing defensive code, for instance, from the executable.

**Exception**

1. Invariants that are used to create infinite loops are permitted.
2. A do … while loop with an essentially Boolean controlling expression that evaluates to 0 is permitted.

**Example**

s8a = ( u16a < 0u ) ? 0 : 1; /\* Non-compliant - u16a always >= 0 \*/

if ( u16a <= 0xffffu )

{

}

/\* Non-compliant - always true \*/

### 119

if ( 2 > 3 )

{

}

/\* Non-compliant - always false \*/

for ( s8a = 0; s8a < 130; ++s8a )

{

/\* Non-compliant - always true \*/

}

if ( ( s8a < 10 ) && ( s8a > 20 ) )

{

/\* Non-compliant - always false \*/

}

if ( ( s8a < 10 ) || ( s8a > 5 ) )

{

}

while ( s8a > 10 )

{

if ( s8a > 5 )

{

}

}

/\* Non-compliant - always true \*/

/\* Non-compliant - s8a not volatile \*/

while ( true )

{

/\* Compliant by exception 1 \*/

}

do

{

/\* Compliant by exception 2 \*/

} while ( 0u == 1u ); const uint8\_t numcyl = 4u;

/\*

* Non-compliant - compiler is permitted to assume that numcyl always
* has value 4

\*/

if ( numcyl == 4u )

{

}

const volatile uint8\_t numcyl\_cal = 4u;

/\*

* Compliant - compiler assumes numcyl\_cal may be changed by
* an external method, e.g. automotive calibration tool, even
* though the program cannot modify its value

\*/

if ( numcyl\_cal == 4u )

{

}

uint16\_t n; /\* 10 <= n <= 100 \*/ uint16\_t sum;

sum = 0;

for ( uint16\_t i = ( n - 6u ); i < n; ++i )

{

sum += i;

}

if ( ( sum % 2u ) == 0u )

{

/\*

* + Non-compliant - sum is the sum of 6 consecutive non-negative
  + integers so must be an odd number. The controlling expression
  + of the if statement will always be false.

\*/

}

**See also**

[Rule 2.1](#_heading=h.2afmg28), [Rule 14.2](#_heading=h.2xn8ts7)

|  |  |
| --- | --- |
| Rule 14.4 | The controlling expression of an *if* statement and the controlling expression of an *iteration-statement* shall have *essentially Boolean* type |
|  | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

The controlling expression of a *for* statement is optional. The rule does not require the expression to be present but does require it to have *essentially Boolean type* if it is present.

**Rationale**

Strong typing requires the controlling expression of an *if* statement or *iteration-statement* to have

*essentially Boolean* type.

**Example**

int32\_t \*p, \*q;

while ( p ) /\* Non-compliant - p is a pointer \*/

{

}

while ( q != NULL ) /\* Compliant \*/

{

}

while ( true ) /\* Compliant \*/

{

}

extern bool\_t flag;

while ( flag ) /\* Compliant \*/

{

}

int32\_t i;

if ( i ) /\* Non-compliant \*/

{

}

if ( i != 0 ) /\* Compliant \*/

{

}

**See also**

[Rule 14.2](#_heading=h.2xn8ts7), [Rule 20.8](#_heading=h.4hr1b5p)

### Control flow

|  |  |
| --- | --- |
| Rule 15.1 | The *goto* statement should not be used |
|  | |
| Category | Advisory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

Unconstrained use of *goto* can lead to programs that are unstructured and extremely difficult to understand.

In some cases a total ban on *goto* requires the introduction of flags to ensure correct control flow, and it is possible that these flags may themselves be less transparent than the *goto* they replace. Therefore, if this rule is not followed, the restricted use of *goto* is allowed where that use follows the guidance in [Rule 15.2](#_heading=h.11si5id) and [Rule 15.3](#_heading=h.3ls5o66).

**See also**

[Rule 9.1](#_heading=h.1fyl9w3), [Rule 15.2](#_heading=h.11si5id), [Rule 15.3](#_heading=h.3ls5o66), [Rule 15.4](#_heading=h.3ws6mnt)

|  |  |
| --- | --- |
| Rule 15.2 | The *goto* statement shall jump to a label declared later in the same function |
|  | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

Unconstrained use of *goto* can lead to programs that are unstructured and extremely difficult to understand.

Restricting the use of *goto* so that “back” jumps are prohibited ensures that iteration only occurs if the iteration statements provided by the language are used, helping to minimize visual code complexity.

**Example**

void f ( void )

{

int32\_t j = 0;

L1:

++j;

if ( 10 == j )

{

goto L2; /\* Compliant \*/

}

goto L1; /\* Non-compliant \*/ L2:

++j;

}

**See also**

[Rule 15.1](#_heading=h.11si5id), [Rule 15.3](#_heading=h.3ls5o66), [Rule 15.4](#_heading=h.3ws6mnt)

|  |  |
| --- | --- |
| Rule 15.3 | Any label referenced by a *goto* statement shall be declared in the same block, or in any block enclosing the *goto* statement |
|  | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

For the purposes of this rule, a *switch-clause* that does not consist of a compound statement is treated as if it were a block.

**Rationale**

Unconstrained use of *goto* can lead to programs that are unstructured and extremely difficult to understand.

Preventing jumps between blocks, or into nested blocks, helps to minimize visual code complexity.

*Note:* C99 is more restrictive when variably modified types are used. An attempt to make a jump from outside the scope of an identifier with a variably modified type into such a scope results in a *constraint* violation.

**Example**

void f1 ( int32\_t a )

{

if ( a <= 0 )

{

goto L2; /\* Non-compliant \*/

}

goto L1; /\* Compliant \*/ if ( a == 0 )

{

goto L1; /\* Compliant \*/

}

goto L2; /\* Non-compliant \*/ L1:

if ( a > 0 )

{

L2:

;

}

}

In the following example, the label L1 is defined in a block which encloses the block containing the *goto* statement. However, the jump crosses from one *switch-clause* to another and, since *switch-clauses* are treated like blocks for the purposes of this rule, the *goto* statement is non-compliant.

switch ( x )

{

case 0:

if ( x == y )

{

goto L1;

}

break; case 1:

y = x;L1:

++x;

break;

default: break;

}

**See also**

[Rule 9.1](#_heading=h.1fyl9w3), [Rule 15.1](#_heading=h.11si5id), [Rule 15.2](#_heading=h.11si5id), [Rule 15.4](#_heading=h.3ws6mnt), [Rule 16.1](#_heading=h.1f7o1he)

|  |  |
| --- | --- |
| Rule 15.4 | There should be no more than one *break* or *goto* statement used to terminate any iteration statement |
|  | |
| Category | Advisory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

Restricting the number of exits from a loop helps to minimize visual code complexity. The use of one *break* or *goto* statement allows a single secondary exit path to be created when early loop termination is required.

**Example**

Both of the following nested loops are compliant as each has a single *break* used for early loop termination.

for ( x = 0; x < LIMIT; ++x )

{

if ( ExitNow ( x ) )

{

break;

}

for ( y = 0; y < x; ++y )

{

if ( ExitNow ( LIMIT - y ) )

{

break;

}

}

}

The following loop is non-compliant as there are multiple *break* and *goto* statements used for early loop termination.

for ( x = 0; x < LIMIT; ++x )

{

if ( BreakNow ( x ) )

{

break;

}

else if ( GotoNow ( x ) )

{

goto EXIT;

}

else

{

KeepGoing ( x );

}

}

EXIT:

;

In the following example, the inner *while* loop is compliant because there is a single *goto* statement that can cause its early termination. However, the outer *while* loop is non-compliant because it can be terminated early either by the *break* statement or by the *goto* statement in the inner *while* loop.

while ( x != 0u )

{

x = calc\_new\_x ( );

if ( x == 1u )

{

break;

}

while ( y != 0u )

{

y = calc\_new\_y ( );

if ( y == 1u )

{

goto L1;

}

}

}

L1:

z = x + y;

**See also**

[Rule 15.1](#_heading=h.11si5id), [Rule 15.2](#_heading=h.11si5id), [Rule 15.3](#_heading=h.3ls5o66)

|  |  |
| --- | --- |
| Rule 15.5 | A function should have a single point of exit at the end |
| [IEC 61508-3 Table B.9], [ISO 26262-6 Table 8] | |
| Category | Advisory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

A function should have no more than one *return* statement.

When a *return* statement is used, it should be the final statement in the compound statement that forms the body of the function.

**Rationale**

A single point of exit is required by IEC 61508 and ISO 26262 as part of the requirements for a modular approach.

Early returns may lead to the unintentional omission of function termination code.

If a function has exit points interspersed with statements that produce *persistent side effects*, it is not easy to determine which *side effects* will occur when the function is executed.

**Example**

In the following non-compliant code example, early returns are used to validate the function parameters.

bool\_t f ( uint16\_t n, char \*p )

{

if ( n > MAX )

{

return false;

}

if ( p == NULL )

{

return false;

}

return true;

}

**See also**

[Rule 17.4](#_heading=h.26sx1u5)

Rule 15.6

The body of an *iteration-statement* or a *selection-statement* shall be a

*compound-statement*

Category Required

Analysis Decidable, Single Translation Unit

Applies to C90, C99

[Koenig 24]

**Amplification**

The body of an *iteration-statement* (*while*, *do … while* or *for*) or a *selection-statement* (*if*, *else*, *switch*) shall be a *compound-statement*.

**Rationale**

It is possible for a developer to mistakenly believe that a sequence of statements forms the body of an *iteration-statement* or *selection-statement* by virtue of their indentation. The accidental inclusion of a semi-colon after the controlling expression is a particular danger, leading to a null control statement. Using a *compound-statement* clearly defines which statements actually form the body.

Additionally, it is possible that indentation may lead a developer to associate an *else* statement with the wrong *if*.

**Exception**

An *if* statement immediately following an *else* need not be contained within a *compound-statement*.

**Example**

The layout for the *compound-statement* and its enclosing braces are style issues which are not addressed by this document; the style used in the following examples is not mandatory.

Maintenance to the following

while ( data\_available )

process\_data ( ); /\* Non-compliant \*/

could accidentally give

while ( data\_available )

process\_data ( ); /\* Non-compliant \*/ service\_watchdog ( );

where service\_watchdog() should have been added to the loop body. The use of a *compound- statement* significantly reduces the chance of this happening.

The next example appears to show that action\_2() is the *else* statement to the first *if*.

if ( flag\_1 )

if ( flag\_2 ) /\* Non-compliant \*/ action\_1 ( ); /\* Non-compliant \*/

else

action\_2 ( ); /\* Non-compliant \*/

when the actual behaviour is

if ( flag\_1 )

{

if ( flag\_2 )

{

action\_1 ( );

}

else

{

action\_2 ( );

}

}

The use of *compound-statement*s ensures that *if* and *else* associations are clearly defined.

The exception allows the use of *else if*, as shown below

if ( flag\_1 )

{

action\_1 ( );

}

else if ( flag\_2 ) /\* Compliant by exception \*/

{

action\_2 ( );

}

else

{

;

}

The following example shows how a spurious semi-colon could lead to an error

while ( flag ); /\* Non-compliant \*/

{

flag = fn ( );

}

The following example shows the compliant method of writing a loop with an empty body:

while ( !data\_available )

{

}

|  |  |
| --- | --- |
| Rule 15.7 | All *if … else if* constructs shall be terminated with an *else* statement |
|  | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

A final *else* statement shall always be provided whenever an *if* statement is followed by a sequence of one or more *else if* constructs. The *else* statement shall contain at least either one *side effect* or a comment.

**Rationale**

Terminating a sequence of *if … else if* constructs with an *else* statement is defensive programming and complements the requirement for a *default* clause in a *switch* statement (see [Rule 16.5](#_heading=h.2bxgwvm)).

The *else* statement is required to have a *side effect* or a comment to ensure that a positive indication is given of the desired behaviour, aiding the code review process.

*Note:* a final *else* statement is not required for a simple *if* statement.

**Example**

The following example is non-compliant as there is no explicit indication that no action is to be taken by the terminating *else*.

if ( flag\_1 )

{

action\_1 ( );

}

else if ( flag\_2 )

{

action\_2 ( );

}

/\* Non-compliant \*/

The following shows a compliant terminating *else*.

else

{

; /\* No action required - ; is optional \*/

}

**See also**

[Rule 16.5](#_heading=h.2bxgwvm)

### Switch statements

Rule 16.1 All *switch* statements shall be well-formed

Category Required

Analysis Decidable, Single Translation Unit

Applies to C90, C99

**Amplification**

A *switch* statement shall be considered to be well-formed if it conforms to the subset of C *switch* statements that is specified by the following syntax rules. If a syntax rule given here has the same name as one defined in The Standard then it replaces the standard version for the scope of the *switch* statement; otherwise, all syntax rules given in The Standard are unchanged.

*switch-statement*:

switch ( *switch-expression* ) { *case-label-clause-list final-default-clause-list* } switch ( *switch-expression* ) { *initial-default-clause-list case-label-clause-list* }

*case-label-clause-list*: *case-clause-list*

*case-label-clause-list case-clause-list*

*case-clause-list*:

*case-label switch-clause case-label case-clause-list*

*case-label*:

case *constant-expression*:

*final-default-clause-list*: default: *switch-clause*

*case-label final-default-clause-list*

*initial-default-clause-list*: default: *switch-clause* default: *case-clause-list*

*switch-clause*:

*statement-list*opt break;

C90: { *declaration-list*opt *statement-list*opt break; } C99: { *block-item-list*opt break; }

Except where explicitly permitted by this syntax, the *case* and *default* keywords may not appear anywhere within a *switch* statement body.

*Note:* some of the restrictions imposed on *switch* statements by this rule are expounded in the rules referenced in the “See also” section. It is therefore possible for code to violate both this rule and one of the more specific rules.

*Note:* the term *switch label* is used within the text of the specific *switch* statement rules to denote either a *case* label or a *default* label.

**Rationale**

The syntax for the *switch* statement in C is not particularly rigorous and can allow complex, unstructured behaviour. This and other rules impose a simple and consistent structure on the *switch* statement

Example

The remaining rules in this section give examples that are also relevant to this rule.

**See also**

[Rule 15.3](#_heading=h.3ls5o66), [Rule 16.2](#_heading=h.3z7bk57), [Rule 16.3](#_heading=h.3z7bk57), [Rule 16.4](#_heading=h.2eclud0), [Rule 16.5](#_heading=h.2bxgwvm), [Rule 16.6](#_heading=h.r2r73f)

|  |  |
| --- | --- |
| Rule 16.2 | A *switch label* shall only be used when the most closely-enclosing compound statement is the body of a *switch* statement |
|  | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

The Standard permits a *switch label*, *i.e.* a *case* label or *default* label, to be placed before any statement contained in the body of a *switch* statement, potentially leading to unstructured code. In order to prevent this, a *switch label* shall only appear at the outermost level of the compound statement forming the body of a *switch* statement.

**Example**

switch ( x )

{

case 1: /\* Compliant \*/ if ( flag )

{

case 2: /\* Non-compliant \*/ x = 1;

}

break; default: break;

}

**See also**

[Rule 16.1](#_heading=h.1f7o1he)

|  |  |
| --- | --- |
| Rule 16.3 | An unconditional *break* statement shall terminate every *switch-clause* |
|  | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

If a developer fails to end a *switch-clause* with a *break* statement, then control flow “falls” into the following *switch-clause* or, if there is no such clause, off the end and into the statement following the *switch* statement. Whilst falling into a following *switch-clause* is sometimes intentional, it is often an error. An unterminated *switch-clause* occurring at the end of a *switch* statement may fall into any *switch-clauses* which are added later.

To ensure that such errors can be detected, the last statement in every *switch-clause* shall be a *break* statement, or if the *switch-clause* is a compound statement, the last statement in the compound statement shall be a *break* statement.

*Note:* a *switch-clause* is defined as containing at least one statement. Two consecutive labels, *case* or

*default*, do not have any intervening statement and are therefore permitted by this rule.

**Example**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | | | | | | | |
| switch ( x | ) |
| { |  |
| case 0: |  |
| break; |  |  |  | /\* | Compliant - unconditional break |  |  |  | \*/ |
| case 1: |  |  |  | /\* | Compliant - empty fall through allows |  | a | group | \*/ |
| case 2: |  |  |  |  |  |  |  |  |  |
| break; |  |  |  | /\* | Compliant |  |  |  | \*/ |
| case 4: |  |  |  |  |  |  |  |  |  |
| a = b; |  |  |  | /\* | Non-compliant - break omitted |  |  |  | \*/ |
| case 5: |  |  |  |  |  |  |  |  |  |
| if ( a | == | b | ) |  |  |  |  |  |  |
| { |  |  |  |  |  |  |  |  |  |
| ++a; |  |  |  |  |  |  |  |  |  |

break; /\* Non-compliant - conditional break \*/

}

default:

; /\* Non-compliant - default must also have a break \*/

}

**See also**

[Rule 16.1](#_heading=h.1f7o1he)

|  |  |
| --- | --- |
| Rule 16.4 | Every *switch* statement shall have a *default* label |
|  | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

The *switch-clause* following the *default* label shall, prior to the terminating *break* statement, contain either:

* A statement, or
* A comment.

**Rationale**

The requirement for a *default* label is defensive programming. Any statements following the *default* label are intended to take some appropriate action. If no statements follow the label then the comment can be used to explain why no specific action has been taken.

**Example**

int16\_t x; switch ( x )

{

case 0:

++x;

break; case 1:

case 2: break;

}

/\* Non-compliant - default label is required \*/int16\_t x; switch ( x )

{

case 0:

++x;

break; case 1:

case 2:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| break; |  | | | |
| default: |  | /\* | Compliant - default label is present | \*/ |
| errorflag | = 1; | /\* | should be non-empty if possible | \*/ |
| break; |  |  |  |  |

}

enum Colours

{ RED, GREEN, BLUE } colour;

switch ( colour )

{case RED:

next = GREEN; break;

case GREEN: next = BLUE; break;

case BLUE: next = RED; break;

}

/\* Non-compliant - no default label.

* Even though all values of the enumeration are
* handled there is no guarantee that colour takes
* one of those values \*/**See also**

[Rule 2.1](#_heading=h.2afmg28), [Rule 16.1](#_heading=h.1f7o1he)

|  |  |
| --- | --- |
| Rule 16.5 | A *default* label shall appear as either the first or the last *switch label* of a *switch* statement |
|  | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

This rule makes it easy to locate the *default* label within a *switch* statement.

**Example**

switch ( x )

{

default: /\* Compliant - default is the first label \*/ case 0:

++x;

break; case 1:

case 2: break;

}

switch ( x )

{

case 0:

++x;

break;

default: /\* Non-compliant - default is mixed with the case labels \*/ x = 0;

break; case 1:

case 2: break;

}

switch ( x )

{

case 0:

++x;

break; case 1:

case 2: break;

default: /\* Compliant - default is the final label \*/ x = 0;

break;

}

**See also**

[Rule 15.7](#_heading=h.3b2epr8), [Rule 16.1](#_heading=h.1f7o1he)

|  |  |
| --- | --- |
| Rule 16.6 | Every *switch* statement shall have at least two *switch-clauses* |
|  | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

A *switch* statement with a single path is redundant and may be indicative of a programming error.

**Example**

switch ( x )

{

default: /\* Non-compliant - switch is redundant \*/ x = 0;

break;

}

switch ( y )

{

case 1:

default: /\* Non-compliant - switch is redundant \*/ y = 0;

break;

}

switch ( z )

{

case 1:

z = 2;

break;

default: /\* Compliant \*/

z = 0;

break;

}

**See also**

[Rule 16.1](#_heading=h.1f7o1he)

|  |  |
| --- | --- |
| Rule 16.7 | A *switch-expression* shall not have *essentially Boolean type* |
|  | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

The Standard requires the controlling expression of a *switch* statement to have an integer type. Since the type that is used to implement Boolean values is an integer, it is possible to have a *switch* statement controlled by a Boolean expression. In this instance an *if-else* construct would be more appropriate.

**Example**

switch ( x == 0 ) /\* Non-compliant - essentially Boolean \*/

{ /\* In this case an "if-else" would be more logical \*/ case false:

y = x; break; default: y = z; break;

}

### Functions

|  |  |
| --- | --- |
| Rule 17.1 | The features of <stdarg.h> shall not be used |
| C90 [Undefined 45, 70–76], C99 [Undefined 81, 128–135] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

None of *va\_list*, *va\_arg*, *va\_start*, *va\_end* and, for C99, *va\_copy* shall be used.

**Rationale**

The Standard lists many instances of undefined behaviour associated with the features of

<stdarg.h>, including:

* *va\_end* not being used prior to end of a function in which *va\_start* was used;
* *va\_arg* being used in different functions on the same *va\_list*;
* The type of an argument not being compatible with the type specified to *va\_arg*.

|  |  |  |  |
| --- | --- | --- | --- |
| Example |  | | |
| #include <stdarg.h> |
| void h ( va\_list ap )  {  double y; | /\* | Non-compliant | \*/ |
| y = va\_arg ( ap, double );  } | /\* | Non-compliant | \*/ |

void f ( uint16\_t n, ... )

{

uint32\_t x;

va\_list ap; /\* Non-compliant \*/

va\_start ( ap, n ); /\* Non-compliant \*/

x = va\_arg ( ap, uint32\_t ); /\* Non-compliant \*/ h ( ap );

/\* undefined - ap is indeterminate because va\_arg used in h ( ) \*/ x = va\_arg ( ap, uint32\_t ); /\* Non-compliant \*/

/\* undefined - returns without using va\_end ( ) \*/

}

void g ( void )

{

/\* undefined - uint32\_t:double type mismatch when f uses va\_arg ( ) \*/ f ( 1, 2.0, 3.0 );

}

Rule 17.2 Functions shall not call themselves, either directly or indirectly

Category Required

Analysis Undecidable, System

Applies to C90, C99

**Rationale**

Recursion carries with it the danger of exceeding available stack space, which can lead to a serious failure. Unless recursion is very tightly controlled, it is not possible to determine before execution what the worst-case stack usage could be.

Rule 17.3 A function shall not be declared implicitly

C90 [Undefined 6, 22, 23]

Category Mandatory

Analysis Decidable, Single Translation Unit

Applies to C90

**Rationale**

Provided that a function call is made in the presence of a prototype, a *constraint* ensures that the number of arguments matches the number of parameters and that each argument can be assigned to its corresponding parameter.

If a function is declared implicitly, a C90 compiler will assume that the function has a return type of *int*. Since an implicit function declaration does not provide a prototype, a compiler will have no information about the number of function parameters and their types. Inappropriate type conversions may result in passing the arguments and assigning the return value, as well as other undefined behaviour.

**Example**

If the function power is declared as:

extern double power ( double d, int n );

but the declaration is not visible in the following code then undefined behaviour will occur.

void func ( void )

{

/\* Non-compliant - return type and both argument types incorrect \*/ double sq1 = power ( 1, 2.0 );

}

**See also**

[Rule 8.2,](#_heading=h.3s49zyc) [Rule 8.4](#_heading=h.meukdy)

Rule 17.4

All exit paths from a function with non-*void* return type shall have an explicit *return* statement with an expression

Category Mandatory

Analysis Decidable, Single Translation Unit

Applies to C90, C99

|  |  |
| --- | --- |
|  |  |
|  | |
|  |  |
|  |  |
|  |  |

C90 [Undefined 43], C99 [Undefined 82]

Rationale

The expression given to the *return* statement provides the value that the function returns. If a non- *void* function does not return a value but the calling function uses the returned value, the behaviour is undefined. This can be avoided by ensuring that, in a non-*void* function:

* Every *return* statement has an expression, and
* Control cannot reach the end of the function without encountering a *return* statement.

*Note:* C99 *constrains* every *return* statement in a non-*void* function to return a value.

**Example**

int32\_t absolute ( int32\_t v )

{

if ( v < 0 )

{

return v;

}

/\*

* Non-compliant - control can reach this point without
* returning a value

\*/

}

uint16\_t lookup ( uint16\_t v )

{

if ( ( v < V\_MIN ) || ( v > V\_MAX ) )

{

/\* Non-compliant - no value returned. Constraint in C99 \*/ return;

}

return table[ v ];

}

**See also**

[Rule 15.5](#_heading=h.20xfydz)

Rule 17.5

The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements

Category Advisory

Analysis Undecidable, System

Applies to C90, C99

|  |  |
| --- | --- |
|  |  |
|  | |
|  |  |
|  |  |
|  |  |

**Amplification**

If a parameter is declared as an array with a specified size, the corresponding argument in each function call should point into an object that has at least as many elements as the array.

**Rationale**

The use of an array declarator for a function parameter specifies the function interface more clearly than using a pointer. The minimum number of elements expected by the function is explicitly stated, whereas this is not possible with a pointer.

A function parameter array declarator which does not specify a size is assumed to indicate that the function can handle an array of any size. In such cases, it is expected that the array size will be communicated by some other means, for example by being passed as another parameter, or by terminating the array with a sentinel value.

The use of an array bound is recommended as it allows out-of-bounds checking to be implemented within the function body and extra checks on parameter passing. It is legal in C to pass an array of the incorrect size to a parameter with a specified size, which can lead to unexpected behaviour.

**Example**

/\*

* Intent is that function does not access outside the range
* array1[ 0 ] .. array1[ 3 ]

\*/

void fn1 ( int32\_t array1[ 4 ] );

/\* Intent is that function handles arrays of any size \*/ void fn2 ( int32\_t array2[ ] );

void fn ( int32\_t \*ptr )

{

int32\_t arr3[ 3 ] = { 1, 2, 3 };

int32\_t arr4[ 4 ] = { 0, 1, 2, 3 };

/\* Compliant - size of array matches the prototype \*/ fn1 ( arr4 );

/\* Non-compliant - size of array does not match prototype \*/ fn1 ( arr3 );

/\* Compliant only if ptr points to at least 4 elements \*/ fn1 ( ptr );

/\* Compliant \*/

fn2 ( arr4 );

/\* Compliant \*/

fn2 ( ptr );

}

**See also**

[Rule 17.6](#_heading=h.1smtxgf)

Rule 17.6

The declaration of an array parameter shall not contain the *static*

keyword between the [ ]

Category Mandatory

Analysis Decidable, Single Translation Unit

Applies to C99

C99 [Undefined 71]

|  |  |
| --- | --- |
|  |  |
|  | |
|  |  |
|  |  |
|  |  |

**Rationale**

The C99 language standard provides a mechanism for the programmer to inform the compiler that an array parameter contains a specified minimum number of elements. Some compilers are able to take advantage of this information to generate more efficient code for some types of processor.

If the guarantee made by the programmer is not honoured, and the number of elements is less than the minimum specified, the behaviour is undefined.

The processors used in typical embedded applications are unlikely to provide the facilities required to take advantage of the additional information provided by the programmer. The risk of the program failing to meet the guaranteed minimum number of elements outweighs any potential performance increase.

Example

|  |
| --- |
|  |

There is no use of this C99 language feature that is compliant with this rule. The examples show some of the undefined behaviour that can arise from its use.

/\* Non-compliant - uses static in array declarator \*/ uint16\_t total ( uint16\_t n, uint16\_t a[ static 20 ] )

{

uint16\_t i; uint16\_t sum = 0U;

/\* Undefined behaviour if a has fewer than 20 elements \*/ for ( i = 0U; i < n; ++i )

{

sum = sum + a[ i ];

}

return sum;

}

extern uint16\_t v1[ 10 ]; extern uint16\_t v2[ 20 ];

void g ( void )

{

uint16\_t x;

x = total ( 10U, v1 ); /\* Undefined - v1 has 10 elements but needs

\* at least 20 \*/ x = total ( 20U, v2 ); /\* Defined but non-compliant \*/

}

**See also**

[Rule 17.5](#_heading=h.3dhjn8m)

|  |  |
| --- | --- |
| Rule 17.7 | The value returned by a function having non-*void* return type shall be  *used* |
|  | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

It is possible to call a function without using the return value, which may be an error. If the return value of a function is intended not to be *used* explicitly, it should be cast to the *void* type. This has the effect of using the value without violating [Rule 2.](#_heading=h.pkwqa1)2.

**Example**

|  |
| --- |
|  |

uint16\_t func ( uint16\_t para1 )

{

return para1;

}

uint16\_t x;

void discarded ( uint16\_t para2 )

{

func ( para2 ); /\* Non-compliant - value discarded \*/ ( void ) func ( para2 ); /\* Compliant \*/

x = func ( para2 ); /\* Compliant \*/

}

**See also**

[Dir 4.7](#_heading=h.34g0dwd), [Rule 2.2](#_heading=h.pkwqa1)

|  |  |
| --- | --- |
| Rule 17.8 | A function parameter should not be modified |
|  | |
| Category | Advisory |
| Analysis | Undecidable, System |
| Applies to | C90, C99 |

**Rationale**

A function parameter behaves in the same manner as an object that has automatic storage duration. While the C language permits parameters to be modified, such use can be confusing and conflict with programmer expectations. It may be less confusing to copy the parameter to an automatic object and modify that copy. With a modern compiler, this will not usually result in any storage or execution time penalty.

Programmers who are unfamiliar with C, but who are used to other languages, may modify a parameter believing that the effects of the modification will be felt in the calling function.

**Example**

int16\_t glob = 0;

void proc ( int16\_t para )

{

para = glob; /\* Non-compliant \*/

}

void f ( char \*p, char \*q )

{

p = q; /\* Non-compliant \*/

\*p = \*q; /\* Compliant \*/

}

### Pointers and arrays

|  |  |
| --- | --- |
| Rule 18.1 | A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand |
| C90 [Undefined 30], C99 [Undefined 43, 44, 46, 59] | |
| Category | Required |
| Analysis | Undecidable, System |
| Applies to | C90, C99 |

**Amplification**

Creation of a pointer to one beyond the end of the array is well-defined by The Standard and is permitted by this rule. Dereferencing a pointer to one beyond the end of an array results in undefined behaviour and is forbidden by this rule.

This rule applies to all forms of array indexing:

integer\_expression + pointer\_expression pointer\_expression + integer\_expression pointer\_expression - integer\_expression pointer\_expression += integer\_expression pointer\_expression -= integer\_expression

++pointer\_expression pointer\_expression++

--pointer\_expression pointer\_expression--

pointer\_expression [ integer\_expression ] integer\_expression [ pointer\_expression ]

*Note:* a subarray is also an array.

*Note:* for purposes of pointer arithmetic, The Standard treats an object that is not a member of an array as if it were an array with a single element (C90 Section 6.3.6, C99 Section 6.5.6).

**Rationale**

Although some compilers may be able to determine at compile time that an array boundary has been exceeded, no checks are generally made at run-time for invalid array subscripts. Using an invalid array subscript can lead to erroneous behaviour of the program.

Run-time derived array subscript values are of the most concern since they cannot easily be checked by static analysis or manual review. Code of a defensive programming nature should, where possible and practicable, be provided to check such subscript values against valid ones and, if required, appropriate action be taken.

It is undefined behaviour if the result obtained from one of the above expressions is not a pointer to an element of the array pointed to by pointer\_expression or an element one beyond the end of that array. See C90 Section 6.3.6 and C99 Section 6.5.6 for further information.

Multi-dimensional arrays are “arrays of arrays”. This rule does not allow pointer arithmetic that results in the pointer addressing a different subarray. Array subscripting over “internal” boundaries shall not be used, as such behaviour is undefined.

**Example**

|  |
| --- |
|  |

The use of the + operator will also violate [Rule 18.4](#_heading=h.1q7ozz1).

int32\_t f1 ( int32\_t \* const a1, int32\_t a2[ 10 ] )

{

int32\_t \*p = &a1[ 3 ]; /\* Compliant/non-compliant depending on

\* the value of a1 \*/

return \*( a2 + 9 ); /\* Compliant \*/

}

void f2 ( void )

{

int32\_t data = 0; int32\_t b = 0;

int32\_t c[ 10 ] = { 0 };

int32\_t d[ 5 ][ 2 ] = { 0 }; /\* 5-element array of 2-element arrays

* of int32\_t \*/

int32\_t \*p1 = &c[ 0 ]; /\* Compliant \*/ int32\_t \*p2 = &c[ 10 ]; /\* Compliant - points to one beyond \*/ int32\_t \*p3 = &c[ 11 ]; /\* Non-compliant - undefined, points to

* two beyond \*/

data = \*p2; /\* Non-compliant - undefined, dereference

* one beyond \*/

data = f1 ( &b, c ); data = f1 ( c, c );

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| p1++; |  | /\* | Compliant |  |  | \*/ |
| c[ -1 ] = | 0; | /\* | Non-compliant | - undefined, | array |  |

* bounds exceeded \*/

data = c[ 10 ]; /\* Non-compliant - undefined, dereference

* of address one beyond \*/ data = \*( &data + 0 ); /\* Compliant - C treats data as an
* array of size 1 \*/

d[ 3 ][ 1 ] = 0; /\* Compliant \*/

data = \*( \*( d + 3 ) + 1 ); /\* Compliant \*/ data = d[ 2 ][ 3 ]; /\* Non-compliant - undefined, internal

* boundary exceeded \*/

p1 = d[ 1 ]; /\* Compliant \*/

data = p1[ 1 ]; /\* Compliant - p1 addresses an array

* of size 2 \*/

}

The following example illustrates pointer arithmetic applied to members of a structure. Because each member is an object in its own right, this rule prevents the use of pointer arithmetic to move from one member to the next. However, it does not prevent arithmetic on a pointer to a member provided that the resulting pointer remains within the bounds of the member object.

struct

{

uint16\_t x; uint16\_t y; uint16\_t z; uint16\_t a[ 10 ];

} s; uint16\_t \*p;

void f3 ( void )

{

p = &s.x;

++p; /\* Compliant - p points one beyond s.x \*/ p[ 0 ] = 1; /\* Non-compliant - undefined, dereference of address one

* beyond s.x which is not necessarily
* the same as s.y \*/ p[ 1 ] = 2; /\* Non-compliant - undefined \*/

p = &s.a[ 0 ]; /\* Compliant - p points into s.a \*/

p = p + 8; /\* Compliant - p still points into s.a \*/ p = p + 3; /\* Non-compliant - undefined, p points more than one

* beyond s.a \*/

}

**See also**

[Dir 4.1](#_heading=h.25b2l0r), [Rule 18.4](#_heading=h.1q7ozz1)

|  |  |
| --- | --- |
| Rule 18.2 | Subtraction between pointers shall only be applied to pointers that address elements of the same array |
| C90 [Undefined 31], C99 [Undefined 45] | |
| Category | Required |
| Analysis | Undecidable, System |
| Applies to | C90, C99 |

**Rationale**

This rule applies to expressions of the form:

pointer\_expression\_1 - pointer\_expression\_2

It is undefined behaviour if pointer\_expression\_1 and pointer\_expression\_2 do not point to elements of the same array or the element one beyond the end of that array.

**Example**

|  |
| --- |
|  |

#include <stddef.h>

void f1 ( int32\_t \*ptr )

{

int32\_t a1[ 10 ]; int32\_t a2[ 10 ]; int32\_t \*p1 = &a1[ 1 ]; int32\_t \*p2 = &a2[ 10 ]; ptrdiff\_t diff;

diff = p1 - a1; /\* Compliant \*/

diff = p2 - a2; /\* Compliant \*/

diff = p1 - p2; /\* Non-compliant \*/

diff = ptr - p1; /\* Non-compliant \*/

}

**See also**

[Dir 4.1](#_heading=h.25b2l0r), [Rule 18.4](#_heading=h.1q7ozz1)

Rule 18.3

The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object

C90 [Undefined 33], C99 [Undefined 50]

Category Required

Analysis Undecidable, System

Applies to C90, C99

**Rationale**

Attempting to make comparisons between pointers will produce undefined behaviour if the two pointers do not point to the same object.

Note: it is permissible to address the next element beyond the end of an array, but accessing this element is not allowed.

**Example**

void f1 ( void )

{

int32\_t a1[ 10 ]; int32\_t a2[ 10 ]; int32\_t \*p1 = a1;

if ( p1 < a1 ) /\* Compliant \*/

{

}

if ( p1 < a2 ) /\* Non-compliant \*/

{

}

}

struct limits

{

int32\_t lwb; int32\_t upb;

};

void f2 ( void )

{

struct limits limits\_1 = { 2, 5 }; struct limits limits\_2 = { 10, 5 };

if ( &limits\_1.lwb <= &limits\_1.upb ) /\* Compliant \*/

{

}

if ( &limits\_1.lwb > &limits\_2.upb ) /\* Non-Compliant \*/

{

}

}

**See also**

[Dir 4.1](#_heading=h.25b2l0r)

|  |  |
| --- | --- |
| Rule 18.4 | The +, -, += and -= operators should not be applied to an expression of pointer type |
|  | |
| Category | Advisory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

Array indexing using the array subscript syntax, ptr[expr], is the preferred form of pointer arithmetic because it is often clearer and hence less error prone than pointer manipulation. Any explicitly calculated pointer value has the potential to access unintended or invalid memory addresses. Such behaviour is also possible with array indexing, but the subscript syntax may ease the task of manual review.

Pointer arithmetic in C can be confusing to the novice. The expression ptr+1 may be mistakenly interpreted as the addition of 1 to the address held in ptr. In fact the new memory address depends on the size in bytes of the pointer’s target. This misunderstanding can lead to unexpected behaviour if *sizeof* is applied incorrectly.

When used with caution however, pointer manipulation using ++ can in some cases be considered more natural; e.g. sequentially accessing locations during a memory test where it is more convenient to treat the memory space as a contiguous set of locations and the address bounds can be determined at compilation time.

**Exception**

Subject to [Rule 18.2](#_heading=h.261ztfg), pointer subtraction between two pointers is allowed.

**Example**

|  |
| --- |
|  |

void fn1 ( void )

{

uint8\_t a[ 10 ]; uint8\_t \*ptr; uint8\_t index = 0U;

index = index + 1U; /\* Compliant - rule only applies to pointers \*/

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| a[ index ] = 0U; ptr = &a[ 5 ]; | /\*  /\* | Compliant Compliant |  |  |  |  | \*/  \*/ |
| ptr = a; ptr++; | /\* | Compliant | - increment | operator | not | + | \*/ |

\*( ptr + 5 ) = 0U; /\* Non-compliant \*/

ptr[ 5 ] = 0U; /\* Compliant \*/

}

void fn2 ( void )

{

uint8\_t array\_2\_2[ 2 ][ 2 ] = { { 1U, 2U }, { 4U, 5U } }; uint8\_t i = 0U;

uint8\_t j = 0U; uint8\_t sum = 0U;

for ( i = 0U; i < 2U; i++ )

{

uint8\_t \*row = array\_2\_2[ i ];

for ( j = 0U; j < 2U; j++ )

{

sum += row[ j ]; /\* Compliant \*/

}

}

}

In the following example, [Rule 18.1](#_heading=h.3qwpj7n) may also be violated if p1 does not point to an array with at least six elements and p2 does not point to an array with at least 4 elements.

void fn3 ( uint8\_t \*p1, uint8\_t p2[ ] )

{

p1++; /\* Compliant \*/

p1 = p1 + 5; /\* Non-compliant \*/

p1[ 5 ] = 0U; /\* Compliant \*/

p2++; /\* Compliant \*/

p2 = p2 + 3; /\* Non-compliant \*/

p2[ 3 ] = 0U; /\* Compliant \*/

}

uint8\_t a1[ 16 ]; uint8\_t a2[ 16 ]; uint8\_t data = 0U;

void fn4 ( void )

{

fn3 ( a1, a2 );

fn3 ( &data, &a2[ 4 ] );

}

**See also**

[Rule 18.1](#_heading=h.3qwpj7n), [Rule 18.2](#_heading=h.261ztfg)

|  |  |
| --- | --- |
| Rule 18.5 | Declarations should contain no more than two levels of pointer nesting |
|  | |
| Category | Advisory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

No more than two *pointer declarators* should be applied consecutively to a type. Any *typedef-name*

appearing in a declaration is treated as if it were replaced by the type that it denotes.

**Rationale**

The use of more than two levels of pointer nesting can seriously impair the ability to understand the behaviour of the code, and should therefore be avoided.

**Example**

|  |
| --- |
|  |

typedef int8\_t \* INTPTR;

void function ( int8\_t \*\* arrPar[ ] ) /\* Non-compliant \*/

{

int8\_t \*\* obj2; /\* Compliant \*/

int8\_t \*\*\* obj3; /\* Non-compliant \*/

INTPTR \* obj4; /\* Compliant \*/ INTPTR \* const \* const obj5; /\* Non-compliant \*/ int8\_t \*\* arr[ 10 ]; /\* Compliant \*/ int8\_t \*\* ( \*parr )[ 10 ]; /\* Compliant \*/ int8\_t \* ( \*\*pparr )[ 10 ]; /\* Compliant \*/

}

struct s

{

int8\_t \* s1; /\* Compliant \*/

int8\_t \*\* s2; /\* Compliant \*/

int8\_t \*\*\* s3; /\* Non-compliant \*/

};

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| struct struct struct | s \* ps1; s \*\* ps2; s \*\*\* ps3; |  |  |  | /\*  /\*  /\* | Compliant Compliant  Non-compliant | \*/  \*/  \*/ |
| int8\_t | \*\* ( \*pfunc1 | )( | void | ); | /\* | Compliant | \*/ |
| int8\_t | \*\* ( \*\*pfunc2 | )( | void | ); | /\* | Compliant | \*/ |
| int8\_t | \*\* ( \*\*\*pfunc3 | )( | void | ); | /\* | Non-compliant | \*/ |
| int8\_t | \*\*\* ( \*\*pfunc4 | )( | void | ); | /\* | Non-compliant | \*/ |
| *Note:* |  |  |  |  |  |  |  |

* arrPar is of type pointer to pointer to pointer to int8\_t because parameters declared with array type are converted to a pointer to the initial element of the array — this is three levels and is non-compliant;
* arr is of type array of pointer to pointer to int8\_t — this is compliant;
* parr is of type pointer to array of pointer to pointer to int8\_t — this is compliant;
* pparr is of type pointer to pointer to array of pointer to int8\_t — this is compliant.

|  |  |
| --- | --- |
| 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 |
| C90 [Undefined 9, 26], C99 [Undefined 8, 9, 40] | |
| Category | Required |
| Analysis | Undecidable, System |
| Applies to | C90, C99 |

**Amplification**

The address of an object might be copied by means of:

* *Assignment*;
* Memory move or copying functions.

**Rationale**

The address of an object becomes indeterminate when the lifetime of that object expires. Any use of an indeterminate address results in undefined behaviour.

**Example**

int8\_t \*func ( void )

{

int8\_t local\_auto;

return &local\_auto; /\* Non-compliant - &local\_auto is indeterminate

\* when func returns \*/

}

In the following example, the function g stores a copy of its pointer parameter p. If p always points to an object with static storage duration then the code is compliant with this rule. However, in the example given, p does point to an object with automatic storage duration. In such a case, copying the parameter p is non-compliant.

uint16\_t \*sp;

void g ( uint16\_t \*p )

{

sp = p; /\* Non-compliant - address of f's parameter u

\* copied to static sp \*/

}

void f ( uint16\_t u )

{

g ( &u );

}

void h ( void )

{

static uint16\_t \*q; uint16\_t x = 0u;

q = &x; /\* Non-compliant - &x stored in object with

\* greater lifetime \*/

}

|  |  |
| --- | --- |
| Rule 18.7 | Flexible array members shall not be declared |
| C99 [Undefined 59] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

Flexible array members are most likely to be used in conjunction with dynamic memory allocation which is banned by [Dir 4.12](#_heading=h.xvir7l) and [Rule 21.3](#_heading=h.1pgrrkc).

The presence of flexible array members modifies the behaviour of the *sizeof* operator in ways that might not be expected by a programmer. The assignment of a structure that contains a flexible array member to another structure of the same type may not behave in the expected manner as it copies only those elements up to but not including the start of the flexible array member.

**Example**

#include <stdlib.h> struct s

{

uint16\_t len;

uint32\_t data[ ]; /\* Non-compliant - flexible array member \*/

} str;

struct s \*copy ( struct s \*s1 )

{

struct s \*s2;

/\* Omit malloc ( ) return check for brevity \*/

s2 = malloc ( sizeof ( struct s ) + ( s1->len \* sizeof ( uint32\_t ) ) );

\*s2 = \*s1; /\* Only copies s1->len \*/

return s2;

}

**See also**

[Dir 4.12](#_heading=h.xvir7l), [Rule 21.3](#_heading=h.1pgrrkc)

|  |  |
| --- | --- |
| Rule 18.8 | Variable-length array types shall not be used |
| C99 [Unspecified 21; Undefined 69, 70] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C99 |

**Rationale**

Variable-length array types are specified when the size of an array declared in a block or a function prototype is not an *integer constant expression*. They are typically implemented as a variable size object stored on the stack. Their use can therefore make it impossible to determine statically the amount of memory that must be reserved for a stack.

If the size of a variable-length array is negative or zero, the behaviour is undefined.

If a variable-length array is used in a context in which it is required to be compatible with another array type, possibly itself variable-length, then the size of the array types shall be identical. Further, all sizes shall evaluate to positive integers. If these requirements are not met, the behaviour is undefined.

If a variable-length array type is used in the operand of a *sizeof* operator, under some circumstances it is unspecified whether the array size expression is evaluated or not.

Each instance of a variable-length array type has its size fixed at the start of its lifetime. This gives rise to behaviour that might be confusing, for example:

void f ( void )

{

uint16\_t n = 5;

typedef uint16\_t Vector[ n ]; /\* An array type with 5 elements \*/ n = 7;

Vector a1; /\* An array type with 5 elements \*/

uint16\_t a2[ n ]; /\* An array type with 7 elements \*/

}

**Example**

There is no use of variable-length arrays that is compliant with this rule. The examples show some of the undefined behaviour that can arise from their use.

void f ( int16\_t n )

{

uint16\_t vla[ n ]; /\* Non-compliant - Undefined if n <= 0 \*/

}

void g ( void )

{

f ( 0 ); /\* Undefined \*/

f ( -1 ); /\* Undefined \*/

f ( 10 ); /\* Defined \*/

}

void h ( uint16\_t n,

uint16\_t a[ 10 ][ n ] ) /\* Non-compliant \*/

{

uint16\_t ( \*p )[ 20 ];

/\* Undefined unless n == 20: incompatible types otherwise \*/ p = a;

}

**See also**

[Rule 13.6](#_heading=h.2981zbj)

### Overlapping storage

|  |  |
| --- | --- |
| Rule 19.1 | An object shall not be assigned or copied to an overlapping object |
| C90 [Undefined 34, 55], C99 [Undefined 51, 94] | |
| Category | Mandatory |
| Analysis | Undecidable, System |
| Applies to | C90, C99 |

**Rationale**

The behaviour is undefined when two objects are created which have some overlap in memory and one is assigned or copied to the other.

**Exception**

The following are permitted because the behaviour is well-defined:

1. Assignment between two objects that overlap exactly and have compatible types (ignoring their type qualifiers)
2. Copying between objects that overlap partially or completely using The Standard Library function memmove

**Example**

This example also violates [Rule 19.2](#_heading=h.44bvf6o) because it uses unions.

void fn ( void )

{

union

{

int16\_t i; int32\_t j;

} a = { 0 }, b = { 1 };

a.j = a.i; /\* Non-compliant \*/

a = b; /\* Compliant - exception 1 \*/

}

#include <string.h> int16\_t a[ 20 ];

void f ( void )

{

memcpy ( &a[ 5 ], &a[ 4 ], 2u \* sizeof ( a[ 0 ] ) ); /\* Non-compliant \*/

}

void g ( void )

{

int16\_t \*p = &a[ 0 ]; int16\_t \*q = &a[ 0 ];

\*p = \*q; /\* Compliant - exception 1 \*/

}

**See also**

[Rule 19.2](#_heading=h.44bvf6o)

Rule 19.2 The *union* keyword should not be used

C90 [Undefined 39, 40; Implementation 27], C99 [Unspecified 10; Undefined 61, 62]

Category Advisory

Analysis Decidable, Single Translation Unit

Applies to C90, C99

|  |  |
| --- | --- |
|  |  |
|  | |
|  |  |
|  |  |
|  |  |

##### Rationale

A union member can be written and the same member can then be read back in a well-defined manner.

However, if a union member is written and then a different union member is read back, the behaviour depends on the relative sizes of the members:

* If the member read is wider than the member written then the value is unspecified;
* Otherwise, the value is implementation-defined.

The Standard permits the bytes of a union member to be accessed by means of another member whose type is array of *unsigned char*. However, since it is possible to access bytes with unspecified values, unions should not be used.

If this rule is not followed, the kinds of behaviour that need to be determined are:

* Padding — how much padding is inserted at the end of the union;
* Alignment — how are members of any structures within the union aligned;
* Endianness — is the most significant byte of a word stored at the lowest or highest memory address;
* Bit-order — how are bits numbered within bytes and how are bits allocated to bit fields.

##### Example

In this non-compliant example, a 16-bit value is stored into a union but a 32-bit value is read back resulting in an unspecified value being returned.

uint32\_t zext ( uint16\_t s )

{

union

{

uint32\_t ul; uint16\_t us;

} tmp;

tmp.us = s;

return tmp.ul; /\* unspecified value \*/

}

**See also**

[Rule 19.1](#_heading=h.1kc7wiv)

### Preprocessing directives

Rule 20.1

*#include* directives should only be preceded by preprocessor directives or comments

Category Advisory

Analysis Decidable, Single Translation Unit

Applies to C90, C99

|  |  |
| --- | --- |
|  |  |
|  | |
|  |  |
|  |  |
|  |  |

C90 [Undefined 56], C99 [Undefined 96, 97]

**Amplification**

The rule shall be applied to the contents of a file before preprocessing occurs.

**Rationale**

To aid code readability, all the *#include* directives in a particular code file should be grouped together near the top of the file.

Additionally, using *#include* to include a standard *header file* within a declaration or definition, or using part of The Standard Library before the inclusion of the related standard *header file* leads to undefined behaviour.

**Example**

|  |
| --- |
|  |

|  |  |  |  |
| --- | --- | --- | --- |
| /\* f.c \*/ |  | | |
| int16\_t #include "f.h" | /\* | Non-compliant | \*/ |
| /\* f1.c \*/ #define F1\_MACRO |  |  |  |
| #include "f1.h" | /\* | Compliant | \*/ |
| #include "f2.h" | /\* | Compliant | \*/ |
| int32\_t i = 0; |  |  |  |
| #include "f3.h" | /\* | Non-compliant | \*/ |

/\* f.h \*/ xyz = 0;

|  |  |
| --- | --- |
| Rule 20.2 | The ', " or \ characters and the /\* or // character sequences shall not occur in a *header file* name |
| C90 [Undefined 14], C99 [Undefined 31] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

The behaviour is undefined if:

* The ', " or \ characters, or the /\* or // character sequences are used between < and >

delimiters in a header name preprocessing token;

* The ' or \ characters, or the /\* or // character sequences are used between the " delimiters in a header name preprocessing token.

*Note:* although use of the \ character results in undefined behaviour, many implementations will accept the / character in its place.

**Example**

#include "fi'le.h" /\* Non-compliant \*/

|  |  |
| --- | --- |
| Rule 20.3 | The *#include* directive shall be followed by either a <filename> or  "filename" sequence |
| C90 [Undefined 48], C99 [Undefined 85] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

This rule applies after macro replacement has been performed.

**Rationale**

The behaviour is undefined if a *#include* directive does not use one of the following forms:

* #include <filename>
* #include "filename"

**Example**

|  |
| --- |
|  |

#include "filename.h" /\* Compliant \*/ #include <filename.h> /\* Compliant \*/ #include another.h /\* Non-compliant \*/

#define HEADER "filename.h"

#include HEADER /\* Compliant \*/ #define FILENAME file2.h

#include FILENAME /\* Non-compliant \*/

#define BASE "base" #define EXT ".ext"

#include BASE EXT /\* Non-compliant - strings are concatenated

\* after preprocessing \*/ #include "./include/cpu.h" /\* Compliant - filename may include a path \*/

|  |  |
| --- | --- |
| Rule 20.4 | A macro shall not be defined with the same name as a keyword |
| C90 [Undefined 56], C99 [Undefined 98] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

This rule applies to all keywords, including those that implement language extensions.

**Rationale**

Using macros to change the meaning of keywords can be confusing. The behaviour is undefined if a standard header is included while a macro is defined with the same name as a keyword.

**Example**

The following example is non-compliant because it alters the behaviour of the *int* keyword. Including a standard header in the presence of this macro results in undefined behaviour.

#define int some\_other\_type #include <stdlib.h>

The following example shows that it is non-compliant to redefine the keyword *while* but it is compliant to define a macro that expands to statements.

#define while( E ) for ( ; ( E ) ; ) /\* Non-compliant - redefined while \*/ #define unless( E ) if ( ! ( E ) ) /\* Compliant \*/

#define seq( S1, S2 ) do { \

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| S1; S2; } while ( false | ) | /\* | Compliant | \*/ |
| #define compound( S ) { S; | } | /\* | Compliant | \*/ |

The following example is compliant in C90, but not C99, because *inline* is not a keyword in C90.

/\* Remove inline if compiling for C90 \*/ #define inline

**See also**

[Rule 21.1](#_heading=h.qbtyoq)

|  |  |
| --- | --- |
| Rule 20.5 | *#undef* should not be used |
|  | |
| Category | Advisory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

The use of *#undef* can make it unclear which macros exist at a particular point within a translation unit.

**Example**

#define QUALIFIER volatile

#undef QUALIFIER /\* Non-compliant \*/ void f ( QUALIFIER int32\_t p )

{

while ( p != 0 )

{

; /\* Wait... \*/

}

}

|  |  |
| --- | --- |
| Rule 20.6 | Tokens that look like a preprocessing directive shall not occur within a macro argument |
| C90 [Undefined 50], C99 [Undefined 87] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

An argument containing sequences of tokens that would otherwise act as preprocessing directives leads to undefined behaviour.

**Example**

#define M( A ) printf ( #A ) #include <stdio.h>

void main ( void )

{

M (

#ifdef SW /\* Non-compliant \*/ "Message 1"

#else /\* Non-compliant \*/ "Message 2"

#endif /\* Non-compliant \*/

);

}

The above may print

#ifdef SW "Message 1" #else "Message 2" #endif

or

"Message 2"

or exhibit some other behaviour.

|  |  |
| --- | --- |
| Rule 20.7 | Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses. |
| [Koenig 78–81] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

If the expansion of any macro parameter produces a token, or sequence of tokens, that form an expression then that expression, in the fully-expanded macro, shall either:

* Be a parenthesized expression itself; or
* Be enclosed in parentheses.

*Note:* this does not necessarily require that all macro parameters are parenthesized; it is acceptable for parentheses to be provided in macro arguments.

**Rationale**

If parentheses are not used, then operator precedence may not give the desired results when macro substitution occurs.

If a macro parameter is not being used as an expression then the parentheses are not necessary because no operators are involved.

**Example**

In the following non-compliant example,

#define M1( x, y ) ( x \* y ) r = M1 ( 1 + 2, 3 + 4 );

the macro expands to give:

r = ( 1 + 2 \* 3 + 4 );

The expressions 1 + 2 and 3 + 4 are derived from expansion of parameters x and y respectively, but neither is enclosed in parentheses. The value of the resulting expression is 11, whereas the result 21 might have been expected.

The code could be written in a compliant manner either by parenthesizing the macro arguments, or by writing an alternative version of the macro that inserts parentheses during expansion, for example:

r = M1 ( ( 1 + 2 ), ( 3 + 4 ) ); /\* Compliant \*/

#define M2( x, y ) ( ( x ) \* ( y ) )

r = M2 ( 1 + 2, 3 + 4 ); /\* Compliant \*/

The following example is compliant because the first expansion of x is as the operand of the ## operator, which does not produce an expression. The second expansion of x is as an expression which is parenthesized as required.

#define M3( x ) a ## x = ( x ) int16\_t M3 ( 0 );

The following example is compliant because expansion of the parameter M as a member name does not produce an expression. Expansion of the parameter S produces an expression, with structure or union type, which does require parentheses.

#define GET\_MEMBER( S, M ) ( S ).M

v = GET\_MEMBER ( s1, minval );

The following compliant example shows that it is not always necessary to parenthesize every instance of a parameter, although this is often the easiest method of complying with this rule.

#define F( X ) G( X ) #define G( Y ) ( ( Y ) + 1 )

int16\_t x = F ( 2 );

The fully-expanded macro is ( ( 2 ) + 1 ). Tracing back through macro expansion, the value 2 arises from expansion of parameter Y in macro G which in turn arises from parameter X in macro F. Since 2 is parenthesized in the fully-expanded macro, the code is compliant.

**See also**

[Dir 4.9](#_heading=h.1jlao46)

|  |  |
| --- | --- |
| Rule 20.8 | The controlling expression of a *#if* or *#elif* preprocessing directive shall evaluate to 0 or 1 |
|  | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

This rule does not apply to controlling expressions in preprocessing directives which are not evaluated. Controlling expressions are not evaluated if they are within code that is being excluded and cannot have an effect on whether code is excluded or not.

**Rationale**

Strong typing requires the controlling expression of conditional inclusion preprocessing directives to have a Boolean value.

Example

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Example |  |  | |
| #define FALSE | 0 |
| #define TRUE | 1 |
| #if FALSE |  | /\* Compliant | \*/ |
|  | #endif  #if 10 |  | /\* Non-compliant | \*/ |
|  | #endif |  |  |  |

#if ! defined ( X ) /\* Compliant \*/ #endif

#if A > B /\* Compliant assuming A and B are numeric \*/ #endif

**See also**

[Rule 14.4](#_heading=h.47hxl2r)

|  |  |
| --- | --- |
| Rule 20.9 | All identifiers used in the controlling expression of *#if* or *#elif*  preprocessing directives shall be *#define*’d before evaluation |
|  | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

As well as using a *#define* preprocessor directive, identifiers may effectively be *#define*’d in other, implementation-defined, ways. For example some implementations support:

* Using a compiler command-line option, such as -D to allow identifiers to be defined prior to translation;
* Using environment variables to achieve the same effect;
* Pre-defined identifiers provided by the compiler.

**Rationale**

If an attempt is made to use a macro identifier in a preprocessor directive, and that identifier has not been defined, then the preprocessor will assume that it has a value of zero. This may not meet developer expectations.

**Example**

|  |
| --- |
|  |

The following examples assume that the macro M is undefined.

#if M == 0 /\* Non-compliant \*/

/\* Does 'M' expand to zero or is it undefined? \*/

#endif

#if defined ( M ) /\* Compliant - M is not evaluated \*/ #if M == 0 /\* Compliant - M is known to be defined \*/

/\* 'M' must expand to zero. \*/

#endif #endif

/\* Compliant - B is only evaluated in ( B == 0 ) if it is defined \*/ #if defined ( B ) && ( B == 0 )

#endif

|  |  |
| --- | --- |
| Rule 20.10 | The # and ## preprocessor operators should not be used |
| C90 [Unspecified 12; Undefined 51, 52], C99 [Unspecified 25; Undefined 3, 88, 89] | |
| Category | Advisory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

The order of evaluation associated with multiple #, multiple ## or a mix of # and ## preprocessor operators is unspecified. In some cases it is therefore not possible to predict the result of macro expansion.

The use of the ## operator can result in code that is obscure.

*Note:* [Rule 1.3](#_heading=h.1baon6m) covers the undefined behaviour that arises if either:

* The result of a # operator is not a valid string literal; or
* The result of a ## operator is not a valid preprocessing token.

**See also**

[Rule 20.11](#_heading=h.1c1lvlb)

|  |  |
| --- | --- |
| Rule 20.11 | A macro parameter immediately following a # operator shall not immediately be followed by a ## operator |
| C90 [Unspecified 12], C99 [Unspecified 25] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

The order of evaluation associated with multiple #, multiple ## or a mix of # and ## preprocessor operators is unspecified. The use of # and ## is discouraged by [Rule 20.1](#_heading=h.1c1lvlb)0. In particular, the result of a # operator is a string literal and it is extremely unlikely that pasting this to any other preprocessing token will result in a valid token.

**Example**

#define A( x ) #x /\* Compliant \*/ #define B( x, y ) x ## y /\* Compliant \*/ #define C( x, y ) #x ## y /\* Non-compliant \*/

**See also**

[Rule 20.10](#_heading=h.1c1lvlb)

|  |  |
| --- | --- |
| Rule 20.12 | A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators |
|  | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Rationale**

A macro parameter that is used as an operand of a # or ## operator is not expanded prior to being used. The same parameter appearing elsewhere in the replacement text is expanded. If the macro parameter is itself subject to macro replacement, its use in mixed contexts within a macro replacement may not meet developer expectations.

**Example**

In the following non-compliant example, the macro parameter x is replaced with AA which is subject to further macro replacement when not used as the operand of ##.

#define AA 0xffff

#define BB( x ) ( x ) + wow ## x /\* Non-compliant \*/

void f ( void )

{

int32\_t wowAA = 0;

/\* Expands as wowAA = ( 0xffff ) + wowAA; \*/ wowAA = BB ( AA );

}

In the following compliant example, the macro parameter X is not subject to further macro replacement.

int32\_t speed; int32\_t speed\_scale; int32\_t scaled\_speed;

#define SCALE( X ) ( ( X ) \* X ## \_scale )

/\* expands to scaled\_speed = ( ( speed ) \* speed\_scale ); \*/ scaled\_speed = SCALE ( speed );

|  |  |
| --- | --- |
| Rule 20.13 | A line whose first token is # shall be a valid preprocessing directive |
|  | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

White-space is permitted between the # and preprocessing tokens.

**Rationale**

A preprocessor directive may be used to conditionally exclude source code until a corresponding

*#else*, *#elif* or *#endif* directive is encountered. A malformed or invalid preprocessing directive contained

within the excluded source code may not be detected by the compiler, possibly leading to the exclusion of more code than was intended.

Requiring all preprocessor directives to be syntactically valid, even when they occur within an excluded block of code, ensures that this cannot happen.

**Example**

In the following example all the code between the *#ifndef* and *#endif* directives may be excluded if AAA is defined. The developer intended that AAA be assigned to x, but the *#else* directive was entered incorrectly and not diagnosed by the compiler.

#define AAA 2 int32\_t foo ( void )

{

int32\_t x = 0;

#ifndef AAA x = 1;

#else1 /\* Non-compliant \*/ x = AAA;

#endif

return x;

}

The following example is compliant because the text #start appearing in a comment is not a token.

/\*

#start is not a token in a comment

\*/

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

**Rationale**

Confusion can arise when blocks of code are included or excluded by the use of conditional compilation directives which are spread over multiple files. Requiring that a *#if* directive be terminated within the same file reduces the visual complexity of the code and the chance that errors will be made during maintenance.

*Note: #if* directives may be used within included files provided they are terminated within the same

file.

**Example**

/\* file1.c \*/

#ifdef A /\* Compliant \*/ #include "file1.h"

#endif

/\* End of file1.c \*/

/\* file2.c \*/

#if 1 /\* Non-compliant \*/ #include "file2.h"

/\* End of file2.c\*/

/\* file1.h \*/

#if 1 /\* Compliant \*/ #endif

/\* End of file1.h \*/

/\* file2.h \*/ #endif

/\* End of file1.h \*/

### Standard libraries

|  |  |
| --- | --- |
| Rule 21.1 | *#define* and *#undef* shall not be used on a reserved identifier or reserved macro name |
| C90 [Undefined 54, 57, 58, 62, 71]  C99 [Undefined 93, 100, 101, 104, 108, 116, 118, 130] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

This rule applies to the following:

* Identifiers or macro names beginning with an underscore;
* Identifiers in file scope described in Section 7, “Library”, of The Standard;
* Macro names described in Section 7, “Library”, of The Standard as being defined in a standard header.

This rule also prohibits the use of *#define* or *#undef* on the identifier *defined* as this results in explicitly undefined behaviour.

This rule does not include those identifiers or macro names that are described in the section of the applicable C standard entitled “Future Library Directions”.

The Standard states that defining a macro with the same name as:

* A macro defined in a standard header, or
* An identifier with file scope declared in a standard header

is well-defined provided that the header is not included. This rule does not permit such definitions on the grounds that they are likely to cause confusion.

*Note:* the macro NDEBUG is not defined in a standard header and may therefore be *#define*’d.

**Rationale**

Reserved identifiers and reserved macro names are intended for use by the implementation. Removing or changing the meaning of a reserved macro may result in undefined behaviour.

**Example**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| #undef | LINE | /\* | Non-compliant | - begins with \_ | \*/ |
| #define | \_GUARD\_H 1 | /\* | Non-compliant | - begins with \_ | \*/ |
| #undef | \_BUILTIN\_sqrt | /\* | Non-compliant | - the implementation |  |

* may use \_BUILTIN\_sqrt for other
* purposes, e.g. generating a sqrt
* instruction \*/

#define defined /\* Non-compliant - reserved identifier \*/ #define errno my\_errno /\* Non-compliant - library identifier \*/ #define isneg( x ) ( ( x ) < 0 ) /\* Compliant - rule doesn't include

* future library
* directions \*/

**See also**

[Rule 20.4](#_heading=h.3im3ia3)

|  |  |
| --- | --- |
| Rule 21.2 | A reserved identifier or macro name shall not be declared |
| C90 [Undefined 57, 58, 64, 71], C99 [Undefined 93, 100, 101, 104, 108, 116, 118, 130] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

See the Amplification for [Rule 21.1](#_heading=h.qbtyoq) for a description of the relevant identifiers and macro names.

**Rationale**

The implementation is permitted to rely on reserved identifiers behaving as described in The Standard and may treat them specially. If reserved identifiers are reused, the program may exhibit undefined behaviour.

**Example**

In the following non-compliant example, the function *memcpy* is declared explicitly. The compliant method of declaring this function is to include <string.h>.

/\*

\* Include <stddef.h> to define size\_t

\*/

#include <stddef.h>

extern void \*memcpy ( void \*restrict s1, const void \*restrict s2, size\_t n );

An implementation is permitted to provide a *function-like macro* definition for each Standard Library function in addition to the library function itself. This feature is often used by compiler writers to generate efficient inline operations in place of the call to a library function. Using a *function-like macro*, the call to a library function can be replaced with a call to a reserved function that is detected by the compiler’s code generation phase and replaced with the inline operation. For example, the fragment of <math.h> that declares *sqrt* might be written using a *function-like macro* that generates a call to

\_BUILTIN\_sqrt which is replaced with an inline SQRT instruction on processors that support it:

extern double sqrt ( double x );

#define sqrt( x ) ( \_BUILTIN\_sqrt ( x ) )The following non-compliant code might interfere with the compiler’s built-in mechanism for handling

sqrt and therefore produce undefined behaviour:

#define \_BUILTIN\_sqrt( x ) ( x ) /\* Non-compliant \*/ #include <math.h>

float64\_t x = sqrt ( ( float64\_t ) 2.0 ); /\* sqrt may not behave as

\* defined in The Standard \*/

|  |  |
| --- | --- |
| Rule 21.3 | The memory allocation and deallocation functions of <stdlib.h>  shall not be used |
| C90 [Unspecified 19; Undefined 9, 91, 92; Implementation 69]  C99 [Unspecified 39, 40; Undefined 8, 9, 168–171; Implementation J.3.12(35)] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

The identifiers *calloc*, *malloc*, *realloc* and *free* shall not be used and no macro with one of these names shall be expanded.

**Rationale**

Use of dynamic memory allocation and deallocation routines provided by The Standard Library can lead to undefined behaviour, for example:

* Memory that was not dynamically allocated is subsequently freed;
* A pointer to freed memory is used in any way;
* Accessing allocated memory before storing a value into it.

*Note:* this rule is a specific instance of [Dir 4.12](#_heading=h.xvir7l).

**See also**

[Dir 4.12](#_heading=h.xvir7l), [Rule 18.7](#_heading=h.3e8gvnb), [Rule 22.1](#_heading=h.3nqndbk), [Rule 22.2](#_heading=h.1tdr5v4)

|  |  |
| --- | --- |
|  |  |
|  | |
|  |  |
|  |  |
|  |  |

**Amplification**

None of the facilities that are specified as being provided by <setjmp.h> shall be used.

**Rationale**

*setjmp* and *longjmp* allow the normal function call mechanisms to be bypassed. Their use may lead to undefined and unspecified behaviour.

|  |  |
| --- | --- |
|  |  |
|  | |
|  |  |
|  |  |
|  |  |

**Amplification**

None of the facilities that are specified as being provided by <signal.h> shall be used.

**Rationale**

Signal handling contains implementation-defined and undefined behaviour.

|  |  |
| --- | --- |
| Rule 21.6 | The Standard Library input/output functions shall not be used |
| C90 [Unspecified 2–5, 16–18; Undefined 77–89; Implementation 53–68]  C99 [Unspecified 3–6, 34–37; Undefined 138–166, 186; Implementation J.3.12(14–32)] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

This rule applies to the functions that are specified as being provided by <stdio.h> and, in C99, their wide-character equivalents specified in Sections 7.24.2 and 7.24.3 of the C99 Standard as being provided by <wchar.h>.

None of these identifiers shall be used and no macro with one of these names shall be expanded.

**Rationale**

Streams and file I/O have unspecified, undefined and implementation-defined behaviours associated with them.

**See also**

[Rule 22.1](#_heading=h.3nqndbk), [Rule 22.3,](#_heading=h.4a7cimu) [Rule 22.4](#_heading=h.2pcmsun), [Rule 22.5,](#_heading=h.2pcmsun) [Rule 22.6](#_heading=h.22vxnjd)

|  |  |
| --- | --- |
| Rule 21.7 | The *atof*, *atoi*, *atol* and *atoll* functions of <stdlib.h> shall not be used |
| C90 [Undefined 90], C99 [Undefined 113] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

The identifiers *atof*, *atoi*, *atol* and, for C99 only *atoll*, shall not be used and no macro with one of these names shall be expanded.

**Rationale**

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

|  |  |
| --- | --- |
| Rule 21.8 | The library functions *abort*, *exit*, *getenv* and *system* of <stdlib.h>  shall not be used |
| Implementation J.3.12(36–38)]  C90 [Undefined 93; Implementation 70–73]  C99 [Undefined 172, 174, 175; | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

The identifiers *abort*, *exit*, *getenv* and *system* shall not be used and no macro with one of these names shall be expanded.

**Rationale**

These functions have undefined and implementation-defined behaviours associated with them.

|  |  |
| --- | --- |
| Rule 21.9 | The library functions *bsearch* and *qsort* of <stdlib.h> shall not be used |
| C90 [Unspecified 20, 21], C99 [Unspecified 41, 42; Undefined 176–178] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

The identifiers *bsearch* and *qsort* shall not be used and no macro with one of these names shall be expanded.

**Rationale**

If the comparison function does not behave consistently when comparing elements, or it modifies any of the elements, the behaviour is undefined.

*Note:* the unspecified behaviour, which relates to the treatment of elements that compare as equal, can be avoided by ensuring that the comparison function never returns 0. When two elements are otherwise equal, the comparison function could return a value that indicates their relative order in the initial array.

The implementation of *qsort* is likely to be recursive and will therefore place unknown demands on stack resource. This is of concern in embedded systems as the stack is likely to be a fixed, often small, size.

|  |  |
| --- | --- |
| Rule 21.10 | The Standard Library time and date functions shall not be used |
| C90 [Unspecified 22; Undefined 80, 97; Implementation 75, 76]  C99 [Unspecified 43, 44; Undefined 146, 154, 182; Implementation J.3.12(39–42)] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C90, C99 |

**Amplification**

None of the facilities that are specified as being provided by <time.h> shall be used.

In C99, the identifier *wcsftime* shall not be used and no macro with this name shall be expanded.

**Rationale**

The time and date functions have unspecified, undefined and implementation-defined behaviours associated with them.

|  |  |
| --- | --- |
| Rule 21.11 | The standard *header file* <tgmath.h> shall not be used |
| C99 [Undefined 184, 185] | |
| Category | Required |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C99 |

**Amplification**

None of the facilities that are specified as being provided by <tgmath.h> shall be used.

**Rationale**

Using the facilities of <tgmath.h> may result in undefined behaviour.

**Example**

|  |
| --- |
|  |

#include <tgmath.h> float f1, f2;

void f ( void )

{

f1 = sqrt ( f2 ); /\* Non-compliant - generic sqrt used \*/

}

#include <math.h> float f1, f2; void f ( void )

{

f1 = sqrtf ( f2 ); /\* Compliant - float version of sqrt used \*/

}

|  |  |
| --- | --- |
| Rule 21.12 | The exception handling features of <fenv.h> should not be used |
| C99 [Unspecified 27, 28; Undefined 109–111; Implementation J.3.6(8)] | |
| Category | Advisory |
| Analysis | Decidable, Single Translation Unit |
| Applies to | C99 |

**Amplification**

The identifiers *feclearexcept*, *fegetexceptflag*, *feraiseexcept*, *fesetexceptflag* and *fetestexcept* shall not be used and no macro with one of these names shall be expanded.

The macros *FE\_INEXACT*, *FE\_DIVBYZERO*, *FE\_UNDERFLOW*, *FE\_OVERFLOW*, *FE\_INVALID* and *FE\_ALL\_EXCEPT*,

along with any implementation-defined floating-point exception macros, shall not be used.

**Rationale**

In some circumstances, the values of the floating-point status flags are unspecified and attempts to access them may lead to undefined behaviour.

The order in which exceptions are raised by the *feraiseexcept* function is unspecified and could therefore result in a program that has been designed for a certain order not operating correctly.

**Example**

|  |
| --- |
|  |

#include <fenv.h>

void f ( float32\_t x, float32\_t y )

{

float32\_t z;

feclearexcept ( FE\_DIVBYZERO ); /\* Non-compliant \*/ z = x / y;

if ( fetestexcept ( FE\_DIVBYZERO ) ) /\* Non-compliant \*/

{

}

else

{

#pragma STDC FENV\_ACCESS ON

z = x \* y;

}

if ( z > x )

{

#pragma STDC FENV\_ACCESS OFF

if ( fetestexcept ( FE\_OVERFLOW ) ) /\* Non-compliant \*/

{

}

}

}

### Resources

Many of the rules in this section are applicable only when rules in other sections have been deviated.

|  |  |
| --- | --- |
| Rule 22.1 | All resources obtained dynamically by means of Standard Library functions shall be explicitly released |
|  | |
| Category | Required |
| Analysis | Undecidable, System |
| Applies to | C90, C99 |

**Amplification**

The Standard Library functions that allocate resources are *malloc*, *calloc*, *realloc* and *fopen*.

**Rationale**

If resources are not explicitly released then it is possible for a failure to occur due to exhaustion of those resources. Releasing resources as soon as possible reduces the possibility that exhaustion will occur.

**Example**

#include <stdlib.h> int main ( void )

{

void \*b = malloc ( 40 );

/\* Non-compliant - dynamic memory not released \*/ return 1;

}

#include <stdio.h> int main ( void )

{

FILE \*fp = fopen ( "tmp", "r" );

/\* Non-compliant - file not closed \*/ return 1;

}

In the following non-compliant example, the handle on “tmp-1” is lost when “tmp-2” is opened.

#include <stdio.h> int main ( void )

{

FILE \*fp;

fp = fopen ( "tmp-1", "w" ); fprintf ( fp, "\*" );

/\* File "tmp-1" should be closed here, but stream 'leaks'. \*/

fp = fopen ( "tmp-2", "w" );

fprintf ( fp, "!" ); fclose ( fp );

return ( 0 );

}

**See also**

[Dir 4.12](#_heading=h.xvir7l), [Dir 4.13](#_heading=h.3hv69ve), [Rule 21.3](#_heading=h.1pgrrkc), [Rule 21.6](#_heading=h.14hx32g)

Rule 22.2

A block of memory shall only be freed if it was allocated by means of a Standard Library function

C90 [Undefined 92], C99 [Undefined 169]

Category Mandatory

Analysis Undecidable, System

Applies to C90, C99

|  |  |
| --- | --- |
|  |  |
|  | |
|  |  |
|  |  |
|  |  |

**Amplification**

The Standard Library functions that allocate memory are *malloc*, *calloc* and *realloc*.

A block of memory is freed when its address is passed to *free* and potentially freed when its address is passed to *realloc*. Once freed, a block of memory is no longer considered to be allocated and therefore cannot subsequently be freed again.

**Rationale**

Freeing non-allocated memory, or freeing the same allocated memory more than once leads to undefined behaviour.

**Example**

#include <stdlib.h> void fn ( void )

{

int32\_t a;

/\* Non-compliant - a does not point to allocated storage \*/ free ( &a );

}

void g ( void )

{

char \*p = ( char \* ) malloc ( 512 ); char \*q = p;

free ( p );

/\* Non-compliant - allocated block freed a second time \*/ free ( q );

/\* Non-compliant - allocated block may be freed a third time \*/ p = ( char \* ) realloc ( p, 1024 );

}

**See also**

[Dir 4.12](#_heading=h.xvir7l), [Dir 4.13](#_heading=h.3hv69ve), [Rule 21.3](#_heading=h.1pgrrkc)

Rule 22.3

The same file shall not be open for read and write access at the same time on different streams

C90 [Implementation 61], C99 [Implementation J.3.12(22)]

Category Required

Analysis Undecidable, System

Applies to C90, C99

|  |  |
| --- | --- |
|  |  |
|  | |
|  |  |
|  |  |
|  |  |

**Amplification**

This rule applies to files opened with The Standard Library functions. It may also apply to similar features provided by the execution environment.

**Rationale**

The Standard does not specify the behaviour if a file is both written and read via different streams.

*Note:* it is acceptable to open a file multiple times for read-only access.

**Example**

#include <stdio.h> void fn ( void )

{

FILE \*fw = fopen ( "tmp", "r+" ); /\* "r+" opens for read/write \*/ FILE \*fr = fopen ( "tmp", "r" ); /\* Non-compliant \*/

}

**See also**

[Rule 21.6](#_heading=h.14hx32g)

Rule 22.4

There shall be no attempt to write to a stream which has been opened as read-only

Category Mandatory

Analysis Undecidable, System

Applies to C90, C99

|  |  |
| --- | --- |
|  |  |
|  | |
|  |  |
|  |  |
|  |  |

**Rationale**

The Standard does not specify the behaviour if an attempt is made to write to a read-only stream. For this reason it is considered unsafe to write to a read-only stream.

**Example**

#include <stdio.h> void fn ( void )

{

FILE \*fp = fopen ( "tmp", "r" );

( void ) fprintf ( fp, "What happens now?" ); /\* Non-compliant \*/

( void ) fclose ( fp );

}

**See also**

[Rule 21.6](#_heading=h.14hx32g)

Rule 22.5 A pointer to a FILE object shall not be dereferenced

Category Mandatory

Analysis Undecidable, System

Applies to C90, C99

|  |  |
| --- | --- |
|  |  |
|  | |
|  |  |
|  |  |
|  |  |

**Amplification**

A pointer to a FILE object shall not be dereferenced directly or indirectly (e.g. by a call to memcpy or

memcmp).

**Rationale**

The Standard (C90 Section 7.9.3(6), C99 Section 7.19.3(6)) states that the address of a FILE object used to control a stream may be significant and a copy of the object may not give the same behaviour. This rule ensures that such a copy cannot be made.

The direct manipulation of a FILE object is prohibited as this may be incompatible with its use as a stream designator.

**Example**

#include <stdio.h>

FILE \*pf1; FILE \*pf2; FILE f3;

pf2 = pf1; /\* Compliant \*/ f3 = \*pf2; /\* Non-compliant \*/

The following example assumes that FILE \* specifies a complete type with a member named pos:

pf1->pos = 0; /\* Non-compliant \*/

**See also**

[Rule 21.6](#_heading=h.14hx32g)

|  |  |
| --- | --- |
| Rule 22.6 | The value of a pointer to a FILE shall not be used after the associated stream has been closed |
| C99 [Undefined 140] | |
| Category | Mandatory |
| Analysis | Undecidable, System |
| Applies to | C90, C99 |

**Rationale**

The Standard states that the value of a FILE pointer is indeterminate after a close operation on a stream.

**Example**

#include <stdio.h> void fn ( void )

{

FILE \*fp; void \*p;

fp = fopen ( "tmp", "w" ); if ( fp == NULL )

{

error\_action ( );

}

fclose ( fp );

fprintf ( fp, "?" ); /\* Non-compliant \*/ p = fp; /\* Non-compliant \*/

}

**See also**

[Dir 4.13](#_heading=h.3hv69ve), [Rule 21.6](#_heading=h.14hx32g)