MISRA C:2012 Coding Standard Manual - Version 1.0

Introduction

This document details the Programming Research enforcement of the rules and directives of the *MISRA C:2012* Guidelines for the use of the C language in critical systems.

The MISRA C3 Compliance Module (M3CM) is a configuration of the Programming Research C static analyser tool: QAC. Each rule and directive is listed with additional details provided in the MISRA C:2012 document.

For further information on implementation of the guidelines, detailed descriptions of terms used throughout the guidelines, and additional code examples, please refer to the official MISRA C:2012 Guidelines document.

Conventions

Throughout this document, a guideline is formatted using the following structure.

Directives	This paragraph describes a guideline for which it is not possible to provide the full description necessary to perform a check for compliance. Adherence is advisory or required.
Rules	This paragraph describes a guideline for which a complete description of the requirement has been provided. Adherence is advisory, required or mandatory.
Amplification	This is text that provides a more precise description of the guideline.
Exception	This is text that describes situations in which the rule does not apply.
Implemented by QAC messages	This is a list of QAC messages, as html links, that enforce the given rule or directive.

Base Standard

This document is the Programming Research, MISRA C:2012 Compliance Module (M3CM), enforcement of MISRA C:2012, described in the "MISRA C:2012 - Guidelines for the use of the C language in critical systems" document, published in March 2013 by The Motor Industry Software Reliability Association (MISRA).

"MISRA C:2012" is based on ISO C Standard 9899:1990 - referred to as C90 and ISO/IEC 9899:1999 - referred to as C99.

Coding Rules

The following sections comprise the directives and rules of the MISRA C:2012 coding guidelines. Details of the support for these guidelines by the M3CM Compliance Module provided by Programming Research for the QAC static analyser are given below.

7 Directives

7.1 The implementation

7.1 Required Dir-1.1 **Any implementation-defined behaviour on which the output of the program depends shall be documented and understood**

Amplification

Appendix G of MISRA C:2012 lists, for both C90 and C99, those implementation-defined behaviours that:

- Are considered to have the potential to cause unexpected program operation, and
- May be present in a program even if it complies with all the other MISRA C guidelines.

All of these implementation-defined behaviours on which the program output depends must be:

- Documented, and
- Understood by developers.

Note: a conforming implementation is required to document its treatment of all implementation-defined behaviour. The developer of an implementation should be consulted if any documentation is missing.

Reference

C90 [Annex G.3], C99 [Annex J.3]

Implemented by QAC messages:

0180 [C99] Use of ll for conversion specifier.
0202 [I] '-' character in '[]' conversion specification is implementation defined.
0284 [I] Multiple character constants have implementation defined values.
0285 [I] Character constant contains character which is not a member of the basic source character set.
0286 [I] String literal contains character which is not a member of the basic source character set.
0287 [I] Header name contains character which is not a member of the basic source character set.
0288 [I] Source file '%s' has comments containing characters which are not members of the basic source character set.
0289 [I] Source file '%s' has preprocessing tokens containing characters which are not members of the basic source character set.
0292 [I] Source file '%s' has comments containing one of the characters '$', '@' or '`'.
0299 [I] Source file '%s' includes #pragma directives containing characters which are not members of the basic source character set.
0320 [C99] Declaration within 'for' statement.
0371 [L] Nesting levels of blocks exceeds 127 - program does not conform strictly to ISO:C99.
0372 [L] More than 63 levels of nested conditional inclusion - program does not conform strictly to ISO:C99.
0375 [L] Nesting of parenthesized expressions exceeds 63 - program does not conform strictly to ISO:C99.
0380 [L] Number of macro definitions exceeds 4095 - program does not conform strictly to ISO:C99.
0388 [L] '#include "%s"' causes nesting to exceed 15 levels - program does not conform strictly to ISO:C99.
0390 [L] Number of members in 'struct' or 'union' exceeds 1023 - program does not conform strictly to ISO:C99.
0391 [L] Number of enumeration constants exceeds 1023 - program does not conform strictly to ISO:C99.
0392 [L] Nesting of 'struct' or 'union' types exceeds 63 - program does not conform strictly to ISO:C99.
0410 [L] Nesting of parentheses exceeds 32 - program does not conform strictly to ISO:C90.
0581 [I] Floating-point constant may be too small to be representable.
0604 [C99] Declaration appears after statements in a compound statement.
0609 [L] More than 12 pointer, array or function declarators modifying a declaration - program does not conform strictly to ISO:C90.
0611 [L] Nesting of 'struct' or 'union' types exceeds 15 - program does not conform strictly to ISO:C90.
0612 [L] Size of object '%s' exceeds 32767 bytes - program does not conform strictly to ISO:C90.
0614 [L] More than 127 block scope identifiers defined within a block - program does not conform strictly to ISO:C90.
0617 [C99] 'const' qualifier has been duplicated.
0618 [C99] 'volatile' qualifier has been duplicated.
0634 [I] Bit-fields in this struct/union have not been declared explicitly as unsigned or signed.
0639 [L] Number of members in 'struct' or 'union' exceeds 127 - program does not conform strictly to ISO:C90.
0647 [L] Number of enumeration constants exceeds 127 - program does not conform strictly to ISO:C90.
0715 [L] Nesting of control structures (statements) exceeds 15 - program does not conform strictly to ISO:C90.
0739 [L] Number of 'case' labels exceeds 257 - program does not conform strictly to ISO:C90.
0810 [L] '#include "%s"' causes nesting to exceed 8 levels - program does not conform strictly to ISO:C90.
0828 [L] More than 8 levels of nested conditional inclusion - program does not conform strictly to ISO:C90.
0850 [C99] Macro argument is empty.
0857 [L] Number of macro definitions exceeds 1024 - program does not conform strictly to ISO:C90.
0858 [L] Number of macro parameters exceeds 31 - program does not conform strictly to ISO:C90.
0859 [L] Number of arguments in macro call exceeds 31 - program does not conform strictly to ISO:C90.
0875 [L] String literal exceeds 509 characters - program does not conform strictly to ISO:C90.
0930 [C99] Trailing comma at the end of an enumerator-list.
1011 [C99] Use of '//' comment.
1018 [C99] Use of LL suffix.
1027 [C99] Use of type 'long long'.
1030 [C99] Macro defined with variable argument list.
1031 [C99] Initializer for 'struct', 'union' or array type is not a constant expression.
1053 [C99] Designators have been used in this initialization list.
1054 [C99] A compound literal has been used.
1055 [C99] The keyword 'inline' has been used.
1056 [C99] The keyword '_Bool' has been used.
2855 Constant: Casting to a signed integer type of insufficient size.
2856 Definite: Casting to a signed integer type of insufficient size.
2857 Apparent: Casting to a signed integer type of insufficient size.
2860 Constant: Implementation-defined value resulting from left shift operation on expression of signed type.

2861 Definite: Implementation-defined value resulting from left shift operation on expression of signed type.
2862 Apparent: Implementation-defined value resulting from left shift operation on expression of signed type.
2890 Constant: Negative value implicitly converted to an unsigned type.
2891 Definite: Negative value implicitly converted to an unsigned type.
2892 Apparent: Negative value implicitly converted to an unsigned type.
2895 Constant: Negative value cast to an unsigned type.
2896 Definite: Negative value cast to an unsigned type.
2897 Apparent: Negative value cast to an unsigned type.
3116 Unrecognized #pragma arguments '%s' This #pragma directive has been ignored.

7.2 Compilation and build

7.2 Required Dir-2.1 All source files shall compile without any compilation errors

7.3 Requirements traceability

7.3 Required Dir-3.1 All code shall be traceable to documented requirements

Reference [DO-178C Section 6.4.4.3.d]

7.4 Code design

7.4 Required Dir-4.1 Run-time failures shall be minimized

Reference C90 [Undefined 15,19,26,30,31,32,33,94], C99 [Undefined 15,16,33,40,43,44,45,48,49,113]

Implemented by QAC messages:

2791 Definite: Right hand operand of shift operator is negative or too large.
2792 Apparent: Right hand operand of shift operator is negative or too large.
2801 Definite: Overflow in signed arithmetic operation.
2802 Apparent: Overflow in signed arithmetic operation.
2811 Definite: Dereference of NULL pointer.
2812 Apparent: Dereference of NULL pointer.
2821 Definite: Arithmetic operation on NULL pointer.
2822 Apparent: Arithmetic operation on NULL pointer.
2831 Definite: Division by zero.
2832 Apparent: Division by zero.
2841 Definite: Dereference of an invalid pointer value.
2842 Apparent: Dereference of an invalid pointer value.
2845 Constant: Maximum number of characters to be written is larger than the target buffer size.
2846 Definite: Maximum number of characters to be written is larger than the target buffer size.
2847 Apparent: Maximum number of characters to be written is larger than the target buffer size.
2871 Infinite loop identified.
2872 This loop, if entered, will never terminate.
2877 This loop will never be executed more than once.

7.4 Advisory Dir-4.2 All usage of assembly language should be documented

Amplification

The rationale for the use of the assembly language and the mechanism for interfacing between C and the assembly language shall be documented.

Implemented by QAC messages:

1003 [E] '#%s' is a language extension for in-line assembler. All statements located between #asm and #endasm will be ignored.
1006 [E] This in-line assembler construct is a language extension. The code has been ignored.

7.4 Required Dir-4.3 Assembly language shall be encapsulated and isolated

Amplification

Where assembly language instructions are used they shall be encapsulated and isolated in:

- Assembly language functions;
- C functions (inline functions preferred for C99);
- C macros.

Implemented by QAC messages:

3006 This function contains a mixture of in-line assembler statements and C statements.

7.4 Advisory Dir-4.4 **Sections of code should not be "commented out"**

Amplification

This rule applies to both // and /* ... */ styles of comment.

7.4 Advisory Dir-4.5 **Identifiers in the same name space with overlapping visibility should be typographically unambiguous**

Amplification

The definition of the term "unambiguous" should be determined for a project taking into account the alphabet and language in which the source code is being written.

For the Latin alphabet as used in English words, it is advised as a minimum that identifiers should not differ by any combination of:

- The interchange of a lowercase character with its uppercase equivalent;
- The presence or absence of the underscore character;
- The interchange of the letter "O", and the digit "0";
- The interchange of the letter "l", and the digit "1";
- The interchange of the letter "l", and the letter "l" (el);
- The interchange of the letter "l" (el), and the digit "1";
- The interchange of the letter "S", and the digit "5";
- The interchange of the letter "Z", and the digit "2";
- The interchange of the letter "n", and the letter "h";
- The interchange of the letter "B", and the digit "8";
- The interchange of the letter sequence "rn" ("r" followed by "n"), and the letter "m";

7.4 Advisory Dir-4.6 **typedefs that indicate size and signedness should be used in place of the basic numerical types**

Amplification

The basic numerical types of char, short, int, long, long long (C99), float, double and long double (C99) should not be used, but specific-length typedefs should be used.

For C99, the types provided by <stdint.h> should be used. For C90, equivalent types should be defined and used.

A type must not be defined with a specific length unless the implemented type is actually of that length.

It is not necessary to use typedefs in the declaration of bit-fields.

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

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

Exception

1. The basic numerical types may be used in a typedef to define a specific-length type.
2. For function "main" an int may be used rather than the typedefs as a return type. Therefore int main (void) is permitted.

3. For function "main" an int may be used rather than the typedefs for the input parameter argc.
4. For function "main" a char may be used rather than the typedefs for the input parameter argv.

Therefore int main(int argc, char *argv[]) is permitted (C99 Section 5.1.2.2.1).

Implemented by QAC messages:

5209 Use of basic type '%s'.

7.4 Required Dir-4.7 — If a function returns error information, then that error information shall be tested

Amplification

The list of functions that are deemed to return error information shall be determined by the project.

The error information returned by a function shall be tested in a meaningful manner.

Exception

If it can be shown, for example by checking arguments, that a function cannot return an error indication then there is no need to perform a check.

7.4 Advisory Dir-4.8 — If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden

Amplification

The implementation of an object should be hidden by means of a pointer to an incomplete type.

7.4 Advisory Dir-4.9 — A function should be used in preference to a function-like macro where they are interchangeable

Amplification

This guideline applies only where a function is permitted by the syntax and constraints of the language standard.

Reference [Koenig 78-81]

Implemented by QAC messages:

3453 A function could probably be used instead of this function-like macro.

7.4 Required Dir-4.10 — Precautions shall be taken in order to prevent the contents of a header file being included more then once

Implemented by QAC messages:

0883 Include file code is not protected against repeated inclusion

---

7.4 Required Dir-4.11 — The validity of values passed to library functions shall be checked

Amplification

The nature and organisation of the project will determine which libraries, and functions within those libraries, should be subject to this directive.

Reference C90 [Undefined 60,63; Implementation 45,47], C99 [Unspecified 30-34,46,52-54]

7.4 Required Dir-4.12 — Dynamic memory allocation shall not be used

Amplification

This rule applies to all dynamic memory allocation packages including:

- Those provided by The Standard Library;
- 3rd party packages.

7.4 Advisory Dir-4.13 — Functions which are designed to provide operations on a resource should be called in an appropriate sequence

Amplification

A set of functions providing operations on a resource typically has three kinds of operation:

1. allocation of the resource, e.g. opening a file
2. deallocation of the resource, e.g. closing a file
3. other operations, e.g. reading from a file

For each such set of functions, all uses of its operations should occur in an appropriate sequence.

8 Rules

8.1 A standard C environment

8.1 Required 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

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).

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.

Reference [MISRA Guidelines Table 3], [IEC 61508-7: Table C.1], [ISO 26262-6: Table 1]

Implemented by QAC messages:

0232	[C] Value of hex escape sequence is not representable in type 'unsigned char'.
0233	[C] Value of octal escape sequence is not representable in type 'unsigned char'.
0244	[C] Value of character constant is not representable in type 'int'.
0261	[C] Comment still open at end of included file.
0321	[C] Declaration within 'for' statement defines an identifier '%s' which is not an object.
0322	[C] Illegal storage class specifier used in 'for' statement declaration.
0338	[C] Octal or hex escape sequence value is too large for 'unsigned char' or 'wchar_t' type.
0422	[C] Function call contains fewer arguments than prototype specifies.
0423	[C] Function call contains more arguments than prototype specifies.
0426	[C] Called function has incomplete return type.
0427	[C] Object identifier used as if it were a function or a function pointer identifier.
0429	[C] Function argument is not of arithmetic type.
0430	[C] Function argument is not of compatible 'struct'/'union' type.
0431	[C] Function argument points to a more heavily qualified type.
0432	[C] Function argument is not of compatible pointer type.
0435	[C] The 'struct'/'union' member '%s' does not exist.
0436	[C] Left operand of '.' must be a 'struct' or 'union' object.
0437	[C] Left operand of '->' must be a pointer to a 'struct' or 'union' object.
0446	[C] Operand of ++/-- must have scalar (arithmetic or pointer) type.
0447	[C] Operand of ++/-- must be a modifiable object.
0448	[C] Operand of ++/-- must not be a pointer to an object of unknown size.
0449	[C] Operand of ++/-- must not be a pointer to a function.
0450	[C] An expression of array type cannot be cast.

0451 [C] Subscripting requires a pointer (or array lvalue).
0452 [C] Cannot subscript a pointer to an object of unknown size.
0453 [C] An array subscript must have integral type.
0454 [C] The address-of operator '&' cannot be applied to an object declared with 'register'.
0456 [C] This expression does not have an address - '&' may only be applied to an lvalue or a function designator.
0457 [C] The address-of operator '&' cannot be applied to a bit-field.
0458 [C] Indirection operator '*' requires operand of pointer type.
0466 [C] Unary '+' requires arithmetic operand.
0467 [C] Operand of '!' must have scalar (arithmetic or pointer) type.
0468 [C] Unary '-' requires arithmetic operand.
0469 [C] Bitwise not '~' requires integral operand.
0476 [C] 'sizeof' cannot be applied to a bit-field.
0477 [C] 'sizeof' cannot be applied to a function.
0478 [C] 'sizeof' cannot be applied to an object of unknown size.
0481 [C] Only scalar expressions may be cast to other types.
0482 [C] Expressions may only be cast to 'void' or scalar types.
0483 [C] A pointer to an object of unknown size cannot be the operand of an addition operator.
0484 [C] A pointer to an object of unknown size cannot be the operand of a subtraction operator.
0485 [C] Only integral expressions may be added to pointers.
0486 [C] Only integral expressions and compatible pointers may be subtracted from pointers.
0487 [C] If two pointers are subtracted, they must be pointers that address compatible types.
0493 [C] Type of left operand is not compatible with this operator.
0494 [C] Type of right operand is not compatible with this operator.
0495 [C] Left operand of '%', '<<', '>>', '&', '^' or '|' must have integral type.
0496 [C] Right operand of '%', '<<', '>>', '&', '^' or '|' must have integral type.
0513 [C] Relational operator used to compare pointers to incompatible types.
0514 [C] Relational operator used to compare a pointer with an incompatible operand.
0515 [C] Equality operator used to compare a pointer with an incompatible operand.
0536 [C] First operand of '&&', '||' or '?' must have scalar (arithmetic or pointer) type.
0537 [C] Second operand of '&&' or '||' must have scalar (arithmetic or pointer) type.
0540 [C] 2nd and 3rd operands of conditional operator '?' must have compatible types.
0541 [C] Argument no. %s does not have object type.
0542 [C] Controlling expression must have scalar (arithmetic or pointer) type.
0546 [C] 'enum %s' has unknown content. Use of an enum tag with undefined content is not permitted.
0547 [C] This declaration of tag '%s' conflicts with a previous declaration.
0550 [C] Left operand of '+=' or '-=' is a pointer to an object of unknown size.
0554 [C] 'static %s()' has been declared and called but no definition has been given.
0555 [C] Invalid assignment to object of void type or array type.
0556 [C] Left operand of assignment must be a modifiable object.
0557 [C] Right operand of assignment is not of arithmetic type.
0558 [C] Right operand of '+=' or '-=' must have integral type when left operand is a pointer.
0559 [C] Right operand of '<<=', '>>=', '&=', '|=', '^=' or '%=' must have integral type.
0560 [C] Left operand of '<<=', '>>=', '&=', '|=', '^=' or '%=' must have integral type.
0561 [C] Right operand of assignment is not of compatible 'struct'/'union' type.
0562 [C] Right operand of assignment points to a more heavily qualified type.
0563 [C] Right operand of assignment is not of compatible pointer type.
0564 [C] Left operand of assignment must be an lvalue (it must designate an object).
0565 [C] Left operand of '+=' or '-=' must be of arithmetic or pointer to object type.
0580 [C] Constant is too large to be representable.
0588 [C] Width of bit-field must be an integral constant expression.
0589 [C] Enumeration constant must be an integral constant expression.
0590 [C] Array bound must be an integral constant expression.
0591 [C] A 'case' label must be an integral constant expression.
0605 [C] A declaration must declare a tag or an identifier.
0616 [C] Illegal combination of type specifiers or storage class specifiers.
0619 [C] The identifier '%s' has already been defined in the current scope within the ordinary identifier namespace.
0620 [C] Cannot initialize '%s' because it has unknown size.
0621 [C] The struct/union '%s' cannot be initialized because it has unknown size.
0622 [C] The identifier '%s' has been declared both with and without linkage in the same scope.

0627 [C] '%s' has different type to previous declaration in the same scope.
0628 [C] '%s' has different type to previous declaration at wider scope.
0629 [C] More than one definition of '%s' (with internal linkage).
0631 [C] More than one declaration of '%s' (with no linkage).
0638 [C] Duplicate member name '%s' in 'struct' or 'union'.
0640 [C] '%s' in 'struct' or 'union' type may not have 'void' type.
0641 [C] '%s' in 'struct' or 'union' type may not have function type.
0642 [C] '%s' in 'struct' or 'union' type may not be an array of unknown size.
0643 [C] '%s' in 'struct' or 'union' type may not be a 'struct' or 'union' with unknown content.
0644 [C] Width of bit-field must be no bigger than the width of an 'int'.
0645 [C] A zero width bit-field cannot be given a name.
0646 [C] Enumeration constants must have values representable as 'int's.
0649 [C] K&R style declaration of parameters is not legal after a function header that includes a parameter list.
0650 [C] Illegal storage class specifier on named function parameter.
0651 [C] Missing type specifiers in function declaration.
0653 [C] Duplicate definition of 'struct', 'union' or 'enum' tag '%s'.
0655 [C] Illegal storage class specifier on unnamed function parameter.
0656 [C] Function return type cannot be function or array type, or an incomplete struct/union (for function definition).
0657 [C] Unnamed parameter specified in function definition.
0659 [C] The identifier '%s' was not given in the parameter list.
0664 [C] Parameter specified with type 'void'.
0665 [C] Two parameters have been declared with the same name '%s'.
0671 [C] Initializer for object of arithmetic type is not of arithmetic type.
0673 [C] Initializer points to a more heavily qualified type.
0674 [C] Initializer for pointer is of incompatible type.
0675 [C] Initializer is not of compatible 'struct'/'union' type.
0677 [C] Array size is negative, or unrepresentable.
0682 [C] Initializer for object of a character type is a string literal.
0683 [C] Initializer for object of a character type is a wide string literal.
0684 [C] Too many initializers.
0685 [C] Initializer for any object with static storage duration must be a constant expression.
0690 [C] String literal contains too many characters to initialize object.
0698 [C] String literal used to initialize an object of incompatible type.
0699 [C] String literal used to initialize a pointer of incompatible type.
0708 [C] No definition found for the label '%s' in this function.
0709 [C] Initialization of locally declared 'extern %s' is illegal.
0736 [C] 'case' label does not have unique value within this 'switch' statement.
0737 [C] More than one 'default' label found in 'switch' statement.
0738 [C] Controlling expression in a 'switch' statement must have integral type.
0746 [C] 'return exp;' found in '%s()' whose return type is 'void'.
0747 [C] 'return exp;' found in '%s()' whose return type is qualified 'void'.
0755 [C] 'return' expression is not of arithmetic type.
0756 [C] 'return' expression is not of compatible 'struct'/'union' type.
0757 [C] 'return' expression points to a more heavily qualified type.
0758 [C] 'return' expression is not of compatible pointer type.
0766 [C] 'continue' statement found outside an iteration statement.
0767 [C] 'break' statement found outside a 'switch' or iteration statement.
0768 [C] 'case' or 'default' found outside a 'switch' statement.
0774 [C] 'auto' may not be specified on global declaration of '%s'.
0775 [C] 'register' may not be specified on global declaration of '%s'.
0801 [C] The '##' operator may not be the first token in a macro replacement list.
0802 [C] The '##' operator may not be the last token in a macro replacement list.
0803 [C] The '#' operator may only appear before a macro parameter.
0804 [C] Macro parameter '%s' is not unique.
0811 [C] The glue operator '##' may only appear in a '#define' preprocessing directive.
0812 [C] Header name token '<text>' found outside '#include' preprocessing directive.
0821 [C] '#include %s' does not identify a header or source file that can be processed.
0834 [C] Function-like macro '%s()' is being redefined as an object-like macro.
0835 [C] Macro '%s' is being redefined with different parameter names.

0844 [C] Macro '%s' is being redefined with a different replacement list.
0845 [C] Object-like macro '%s' is being redefined as a function-like macro.
0851 [C] More arguments in macro call than specified in definition.
0852 [C] Unable to find the ')' that marks the end of the macro call.
0856 [C] Fewer arguments in macro call than specified in definition.
0866 [C] The string literal in a '#line' directive cannot be a 'wide string literal'.
0873 [C] Preprocessing token cannot be converted to an actual token.
0877 [C] '#if' and '#elif' expressions may contain only integral constants.
0940 [C] Illegal usage of a variably modified type.
0941 [C] A variable length array may not be initialized.
0943 [C] Jump to label '%s' is a jump into the scope of an identifier with variably modified type.
0944 [C] The label '%s' is inside the scope of an identifier with variably modified type.
1023 [C] Using '__alignof__' on function types is illegal.
1024 [C] Using '__alignof__' on incomplete types is illegal.
1025 [C] Using '__alignof__' on bit-fields is illegal.
1033 [C] The identifier __VA_ARGS__ may only be used in the replacement list of a variadic macro.
1047 [C] Function is being declared with default argument syntax after a previous call to the function. This is not allowed.
1048 [C] Default argument values are missing for some parameters in this function declaration. This is not allowed.
3236 [C] 'inline' may not be applied to function 'main'.
3237 [C] inline function '%1s' has external linkage and is defining an object, '%2s', with static storage duration.
3238 [C] inline function '%1s' has external linkage and is referring to an object, '%2s', with internal linkage.
3244 [C] 'inline' may only be used in the declaration of a function identifier.

8.1 Advisory Rule-1.2 **Language extensions should not be used**

Implemented by QAC messages:

0240 [E] This file contains the control-M character at the end of a line.
0241 [E] This file contains the control-Z character - was this transferred from a PC?
0246 [E] Binary integer constants are a language extension.
0551 [E] Cast may not operate on the left operand of the assignment operator.
0601 [E] Function 'main()' is not of type 'int (void)' or 'int (int, char *[])'.
0633 [E] Empty structures and unions are a language extension.
0635 [E] Bit-fields in this struct/union have been declared with types other than int, signed int or unsigned int.
0660 [E] Defining an unnamed member in a struct or union. This is a language extension.
0662 [E] Accessing a member of an unnamed struct or union member in this way is a language extension.
0830 [E] Unrecognized text encountered after a preprocessing directive.
0831 [E] Use of '\\' in this '#include' line is a PC extension - this usage is non-portable.
0899 [E] Unrecognized preprocessing directive has been ignored - assumed to be a language extension.
1001 [E] '#include %s' is a VMS extension.
1002 [E] '%s' is not a legal identifier in ISO C.
1003 [E] '#%s' is a language extension for in-line assembler. All statements located between #asm and #endasm will be ignored.
1006 [E] This in-line assembler construct is a language extension. The code has been ignored.
1008 [E] '#%s' is not a legal ISO C preprocessing directive.
1012 [E] Use of a C++ reference type ('type &') will be treated as a language extension.
1014 [E] Non-standard type specifier - this will be treated as a language extension.
1015 [E] '%s' is not a legal keyword in ISO C - this will be treated as a language extension.
1019 [E] '@ address' is not supported in ISO C - this will be treated as a language extension.
1020 [E] '__typeof__' is not supported in ISO C, and is treated as a language extension.
1021 [E] A statement expression is not supported in ISO C, and is treated as a language extension.
1022 [E] '__alignof__' is not supported in ISO C, and is treated as a language extension.
1026 [E] The indicated @word construct has been ignored.
1028 [E] Use of the sizeof operator in a preprocessing directive is a language extension.
1029 [E] Whitespace encountered between backslash and new-line has been ignored.
1034 [E] Macro defined with named variable argument list. This is a language extension.
1035 [E] No macro arguments supplied for variable argument list. This is a language extension.
1036 [E] Comma before ## ignored in expansion of variadic macro. This is a language extension.
1037 [E] Arrays of length zero are a language extension.
1038 [E] The sequence ", ##__VA_ARGS__" is a language extension.

1041 [E] Empty aggregate initializers are a language extension.
1042 [E] Using I64 or UI64 as an integer constant suffix. This is a language extension.
1043 [E] Defining an anonymous union object. This is a language extension.
1044 [E] Defining an anonymous struct object. This is a language extension.
1045 [E] Use of the #include_next preprocessing directive is a language extension.
1046 [E] Function is being declared with default argument syntax. This is a language extension.
3664 [E] Using a dot operator to access an individual bit is a language extension.

8.1 Required Rule-1.3 **There shall be no occurrence of undefined or critical unspecified behaviour**

Amplification

Some undefined and unspecified behaviours are dealt with by specific rules. This rule prevents all other undefined and critical unspecified behaviours. Appendix H lists the undefined behaviours and those unspecified behaviours that are considered critical.

Implemented by QAC messages:

0160 [U] Using unsupported conversion specifier number %s.
0161 [U] Unknown length modifier used with 'i' or 'd' conversion specifier, number %s.
0162 [U] Unknown length modifier used with 'o' conversion specifier, number %s.
0163 [U] Unknown length modifier used with 'u' conversion specifier, number %s.
0164 [U] Unknown length modifier used with 'x' conversion specifier, number %s.
0165 [U] Unknown length modifier used with 'X' conversion specifier, number %s.
0166 [U] Unknown length modifier used with 'f' conversion specifier, number %s.
0167 [U] Unknown length modifier used with 'e' conversion specifier, number %s.
0168 [U] Unknown length modifier used with 'E' conversion specifier, number %s.
0169 [U] Unknown length modifier used with 'g' conversion specifier, number %s.
0170 [U] Unknown length modifier used with 'G' conversion specifier, number %s.
0171 [U] Unknown length modifier used with 'c' conversion specifier, number %s.
0172 [U] Unknown length modifier used with '%%' conversion specifier, number %s.
0173 [U] Unknown length modifier used with 's' conversion specifier, number %s.
0174 [U] Unknown length modifier used with 'n' conversion specifier, number %s.
0175 [U] Unknown length modifier used with 'p' conversion specifier, number %s.
0176 [U] Incomplete conversion specifier, number %s.
0177 [U] Field width of format conversion specifier exceeds 509 characters.
0178 [U] Precision of format conversion specifier exceeds 509 characters.
0179 [U] Argument type does not match conversion specifier number %s.
0184 [U] Insufficient arguments to satisfy conversion specifier, number %s.
0185 [U] Call contains more arguments than conversion specifiers.
0186 [U] A call to this function must include at least one argument.
0190 [U] Using unsupported conversion specifier number %s.
0191 [U] Unknown length modifier used with 'd/i/n' conversion specifier, number %s.
0192 [U] Unknown length modifier used with 'o' conversion specifier, number %s.
0193 [U] Unknown length modifier used with 'u' conversion specifier, number %s.
0194 [U] Unknown length modifier used with 'x/X' conversion specifier, number %s.
0195 [U] Unknown length modifier used with 'e/E/f/F/g/G' conversion specifier, number %s.
0196 [U] Unknown length modifier used with 's' conversion specifier, number %s.
0197 [U] Unknown length modifier used with 'p' conversion specifier, number %s.
0198 [U] Unknown length modifier used with '%%' conversion specifier, number %s.
0199 [U] Unknown length modifier used with '[' conversion specifier, number %s.
0200 [U] Unknown length modifier used with 'c' conversion specifier, number %s.
0201 [U] Incomplete conversion specifier, number %s.
0203 [U] Value of character prior to '-' in '[]' is greater than following character.
0204 [U] Field width of format conversion specifier exceeds 509 characters.
0206 [U] Argument type does not match conversion specifier number %s.
0207 [U] 'scanf' expects address of objects being stored into.
0208 [U] Same character occurs in scanset more than once.
0235 [U] Unknown escape sequence.
0275 [U] Floating value is out of range for conversion to destination type.
0301 [u] Cast between a pointer to object and a floating type.
0302 [u] Cast between a pointer to function and a floating type.

0304	[U] The address of an array declared 'register' may not be computed.
0307	[u] Cast between a pointer to object and a pointer to function.
0309	[U] Integral type is not large enough to hold a pointer value.
0337	[U] String literal has undefined value. This may be a result of using '#' on \\.
0400	[U] '%s' is modified more than once between sequence points - evaluation order unspecified.
0401	[U] '%s' may be modified more than once between sequence points - evaluation order unspecified.
0402	[U] '%s' is modified and accessed between sequence points - evaluation order unspecified.
0403	[U] '%s' may be modified and accessed between sequence points - evaluation order unspecified.
0475	[u] Operand of 'sizeof' is an expression designating a bit-field.
0543	[U] 'void' expressions have no value and may not be used in expressions.
0544	[U] The value of an incomplete 'union' may not be used.
0545	[U] The value of an incomplete 'struct' may not be used.
0602	[U] The identifier '%s' is reserved for use by the library.
0623	[U] '%s' has incomplete type and no linkage - this is undefined.
0625	[U] '%s' has been declared with both internal and external linkage - the behaviour is undefined.
0626	[U] '%s' has different type to previous declaration (which is no longer in scope).
0630	[U] More than one definition of '%s' (with external linkage).
0632	[U] Tentative definition of '%s' with internal linkage cannot have unknown size.
0636	[U] There are no named members in this 'struct' or 'union'.
0654	[U] Using 'const' or 'volatile' in a function return type is undefined.
0658	[U] Parameter cannot have 'void' type.
0661	[U] '%s()' may not have a storage class specifier of 'static' when declared at block scope.
0667	[U] '%s' is declared as a typedef and may not be redeclared as an object at an inner scope without an explicit type specifier.
0668	[U] '%s' is declared as a typedef and may not be redeclared as a member of a 'struct' or 'union' without an explicit type specifier.
0672	[U] The initializer for a 'struct', 'union' or array is not enclosed in braces.
0676	[u] Array element is of function type. Arrays cannot be constructed from function types.
0678	[u] Array element is array of unknown size. Arrays cannot be constructed from incomplete types.
0680	[u] Array element is 'void' or an incomplete 'struct' or 'union'. Arrays cannot be constructed from incomplete types.
0706	[U] Label '%s' is not unique within this function.
0745	[U] 'return;' found in '%s()', which has been defined with a non-'void' return type.
0777	[U] External identifier does not differ from other identifier(s) (e.g. '%s') within the specified number of significant characters.
0779	[U] Identifier does not differ from other identifier(s) (e.g. '%s') within the specified number of significant characters.
0809	[U] The '#include' preprocessing directive has not been followed by <h-char-sequence> or "s-char-sequence".
0813	[U] Using any of the characters ' " or /* in '#include <%s>' gives undefined behaviour.
0814	[U] Using the characters ' or /* in '#include "%s"' gives undefined behaviour.
0836	[U] Definition of macro named 'defined'.
0837	[U] Use of '#undef' to remove the operator 'defined'.
0848	[U] Attempting to #undef '%s', which is a predefined macro name.
0853	[U] Macro arguments contain a sequence of tokens that has the form of a preprocessing directive.
0854	[U] Attempting to #define '%s', which is a predefined macro name.
0864	[U] '#line' directive specifies line number which is not in the range 1 to 32767.
0865	[U] '#line' directive is badly formed.
0867	[U] '#line' has not been followed by a line number.
0872	[U] Result of '##' operator is not a legal preprocessing token.
0874	[U] Character string literal and wide character string literal are adjacent.
0885	[U] The token 'defined' is generated in the expansion of this macro.
0887	[U] Use of 'defined' must match either 'defined(identifier)' or 'defined identifier'.
0888	[U] 'defined' requires an identifier as an argument.
0914	[U] Source file does not end with a newline character.
0915	[U] Source file ends with a backslash character followed by a newline.
0942	[U] A * can only be used to specify array size within function prototype scope.
1331	Type or number of arguments doesn't match previous use of the function.
1332	Type or number of arguments doesn't match prototype found later.
1333	Type or number of arguments doesn't match function definition found later.
1509	'%1s' has external linkage and has multiple definitions.
1510	'%1s' has external linkage and has incompatible declarations.
2800	Constant: Overflow in signed arithmetic operation.

2810 Constant: Dereference of NULL pointer.
2820 Constant: Arithmetic operation on NULL pointer.
2830 Constant: Division by zero.
2840 Constant: Dereference of an invalid pointer value.
3113 [U] 'return' statement includes no expression but function '%s()' is implicitly of type 'int'.
3114 [U] Function '%s()' is implicitly of type 'int' but ends without returning a value.
3239 [U] inline function '%1s' has external linkage, but is not defined within this translation unit.
3311 [u] An earlier jump to this statement will bypass the initialization of local variables.
3312 [u] This goto statement will jump into a previous block and bypass the initialization of local variables.
3319 [U] Function called with number of arguments which differs from number of parameters in definition.
3320 Type of argument no. %s differs from its type in definition of function.
3437 [u] The assert macro has been suppressed to call a function of that name.
3438 [U] #undef'ing the assert macro to call a function of that name causes undefined behaviour.

8.2 Unused code

8.2 Required Rule-2.1 A project shall not contain unreachable code

Reference

[IEC 61508-7 Section C.5.9], [DO-178C Section 6.4.4.03.c]

Implemented by QAC messages:

0594 Negative 'case' label expression is incompatible with unsigned controlling expression in 'switch' statement.
1460 'Switch' label value, %s, not contained in enum type.
1503 The function '%1s' is defined but is not used within this project.
2742 This 'if' controlling expression is a constant expression and its value is 'false'.
2744 This 'while' or 'for' loop controlling expression is a constant expression and its value is 'false'. The loop will not be entered.
2880 This code is unreachable.
2882 This 'switch' statement will bypass the initialization of local variables.
3219 Static function '%s()' is not used within this translation unit.

8.2 Required Rule-2.2 There shall be no dead code

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.

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.

Reference

[IEC 61508-7 Section C.5.10], [ISO 26262-6 Section 9.4.5], [DO-178C Section 6.4.4.03.c]

Implemented by QAC messages:

2980 The value of this function parameter is never used before being modified.
2981 This initialization is redundant. The value of this object is never used before being modified.
2982 This assignment is redundant. The value of this object is never used before being modified.
2983 This assignment is redundant. The value of this object is never subsequently used.
2984 This operation is redundant. The value of the result is always '%1s'.
2985 This operation is redundant. The value of the result is always that of the left-hand operand.
2986 This operation is redundant. The value of the result is always that of the right-hand operand.
2995 The result of this logical operation is always 'true'.
2996 The result of this logical operation is always 'false'.
3110 The left-hand operand of this ',' has no side effects.
3112 This statement has no side-effect - it can be removed.
3425 One branch of this conditional operation is a redundant expression.

3426 Right hand side of comma expression has no side effect and its value is not used.
3427 Right hand side of logical operator has no side effect and its value is not used.

8.2 Advisory Rule-2.3 A project should not contain unused type declarations

8.2 Advisory Rule-2.4 A project should not contain unused tag declarations

8.2 Advisory Rule-2.5 A project should not contain unused macro declarations

8.2 Advisory Rule-2.6 A function should not contain unused label declarations

Implemented by QAC messages:

3202 The label '%s:' is not used in this function and could be removed.

8.2 Advisory Rule-2.7 There should be no unused parameters in functions

Implemented by QAC messages:

3206 The parameter '%s' is not used in this function.

8.3 Comments

8.3 Required Rule-3.1 The character sequences /* and // shall not be used within a comment.

Exception

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

Implemented by QAC messages:

3108 Nested comments are not recognized in the ISO standard.
5133 Comment delimiter /* or // found within comment.

8.3 Required Rule-3.2 Line-splicing shall not be used in // comments.

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.

Implemented by QAC messages:

5134 C++ style comment uses line splicing.

8.4 Character sets and lexical conventions

8.4 Required Rule-4.1 Octal and hexadecimal escape sequences shall be terminated

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.

Reference C90 [Implementation 11], C99 [Implementation J.3.4(7-9)]

8.4 Advisory Rule-4.2 Trigraphs should not be used

Implemented by QAC messages:

3601 Trigraphs (??x) are an ISO feature.

8.5 Identifiers

8.5 Required Rule-5.1 External identifiers shall be distinct

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.

Reference

C90 [Undefined 7], C99 [Unspecified 7; Undefined 28]

Implemented by QAC messages:

0777 [U] External identifier does not differ from other identifier(s) (e.g. '%s') within the specified number of significant characters.

8.5 Required Rule-5.2 Identifiers declared in the same scope and name space shall be distinct

Amplification

This rule does not apply if both identifers are external identifers because this case is covered by Rule 5.1.

This rule does not apply if either identifer is a macro identifer because this case is covered by Rule 5.4 and Rule 5.5.

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;

Reference

C90 [Undefined 7], C99 [Undefined 28]

Implemented by QAC messages:

0779 [U] Identifier does not differ from other identifier(s) (e.g. '%s') within the specified number of significant characters.

8.5 Required Rule-5.3 An identifier declared in an inner scope shall not hide an identifier declared in an outer scope

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.

Implemented by QAC messages:

2547 This declaration of tag '%s' hides a more global declaration.

3334 This declaration of '%s' hides a more global declaration.

8.5 Required Rule-5.4 Macro identifiers shall be distinct

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.

Reference C90 [Undefined 7], C99 [Unspecified 7; Undefined 28]

8.5 Required Rule-5.5 **Identifiers shall be distinct from macro names**

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.

Reference C90 [Undefined 7], C99 [Unspecified 7; Undefined 28]

8.5 Required Rule-5.6 **A typedef name shall be a unique identifier**

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.

Exception

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

Implemented by QAC messages:

1506	The identifier '%1s' is declared as a typedef and is used elsewhere for a different kind of declaration.
1507	'%1s' is used as a typedef for different types.
1508	The typedef '%1s' is declared in more than one location.
3448	Declaration of typedef '%s' is not in a header file although it is used in a definition or declaration with external linkage.

8.5 Required Rule-5.7 **A tag name shall be a unique identifier**

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.

Exception

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

8.5 Required Rule-5.8 **Identifiers that define objects or functions with external linkage shall be unique**

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.

Implemented by QAC messages:

1525 Object/function with external linkage has same identifier as another object/function with internal linkage.
1526 Object with no linkage has same identifier as another object/function with external linkage.

8.5 Advisory Rule-5.9 **Identifiers that define objects or functions with internal linkage should be unique**

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.

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.

Implemented by QAC messages:

1525 Object/function with external linkage has same identifier as another object/function with internal linkage.
1527 Object/function with internal linkage has same identifier as another object/function with internal linkage.
1528 Object with no linkage has same identifier as another object/function with internal linkage.

8.6 Types

8.6 Required Rule-6.1 **Bit-fields shall only be declared with an appropriate type**

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 typedefs to designate an appropriate type.

Reference C90 [Undefined 38; Implementation 29], C99 [Implementation J.3.9(1,2)]

Implemented by QAC messages:

0634 [I] Bit-fields in this struct/union have not been declared explicitly as unsigned or signed.
0635 [E] Bit-fields in this struct/union have been declared with types other than int, signed int or unsigned int.

8.6 Required Rule-6.2 **Single-bit named bit fields shall not be of a signed type**

Implemented by QAC messages:

3660 Named bit-field consisting of a single bit declared with a signed type.
3665 Unnamed bit-field consisting of a single bit declared with a signed type.

8.7 Literals and constants

8.7 Required Rule-7.1 **Octal constants shall not be used**

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.

Reference [Koenig 9]

Implemented by QAC messages:

0336 Macro defined as an octal constant.
0339 Octal constant used.
3628 Octal escape sequences used in a character constant or string literal.

8.7 Required Rule-7.2 — A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type

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).

Implemented by QAC messages:

1281 Integer literal constant is of an unsigned type but does not include a "U" suffix.

8.7 Required Rule-7.3 — The lowercase character "l" shall not be used in a literal suffix

Implemented by QAC messages:

1280 A lowercase letter L (l) has been used in an integer or floating suffix.

8.7 Required Rule-7.4 — A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char"

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-qualifed 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-qualifed 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-qualifed wchar_t".

Reference

C90 [Undefined 12], C99 [Unspecified 14; Undefined 30]

Implemented by QAC messages:

0752 String literal passed as argument to function whose parameter is not a 'pointer to const'.
0753 String literal assigned to pointer which is not a 'pointer to const'.

8.8 Declarations and definitions

8.8 Required Rule-8.1 — Types shall be explicitly specified

Implemented by QAC messages:

2050 The 'int' type specifier has been omitted from a function declaration.
2051 The 'int' type specifier has been omitted from an object declaration.

8.8 Required Rule-8.2 — Function types shall be in prototype form with named parameters

Reference

C90 [Undefined 22,23,25], C99 [Undefined 36,37,38]

Implemented by QAC messages:

1335 Parameter identifiers missing in function prototype declaration.
1336 Parameter identifiers missing in declaration of a function type.
3001 Function has been declared with an empty parameter list.
3002 Defining '%s()' with an identifier list and separate parameter declarations is an obsolescent feature.
3007 "void" has been omitted when defining a function with no parameters.

8.8 Required Rule-8.3 — All declarations of an object or function shall use the same names and type qualifiers

Amplification

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

Exception

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

Reference

[Koenig 59-62]

Implemented by QAC messages:

0624	Function '%s' is declared using typedefs which are different to those in a previous declaration.
1330	The parameter identifiers in this function declaration differ from those in a previous declaration.
3675	Function parameter declared with type qualification which differs from previous declaration.

8.8 Required Rule-8.4 — A compatible declaration shall be visible when an object or function with external linkage is defined

Amplification

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

Implemented by QAC messages:

3408	'%s' has external linkage and is being defined without any previous declaration.

8.8 Required Rule-8.5 — An external object or function shall be declared once in one and only one file

Amplification

This rule applies to non-defining declarations only.

Reference

[Koenig 66]

Implemented by QAC messages:

1513	Identifier '%1s' with external linkage has separate non-defining declarations in more than one location.
3221	Function with external linkage declared at block scope.
3222	Object with external linkage declared at block scope.
3447	'%s' is being declared with external linkage but this declaration is not in a header file.
3451	The global identifier '%s' has been declared in more than one file.

8.8 Required Rule-8.6 — An identifier with external linkage shall have exactly one external definition

Reference

C90 [Undefined 44], C99 [Undefined 78], [Koenig 55,63-65]

Implemented by QAC messages:

0630	[U] More than one definition of '%s' (with external linkage).
1509	'%1s' has external linkage and has multiple definitions.
3406	Object/function '%s', with external linkage, has been defined in a header file.

8.8 Advisory Rule-8.7 — Functions and objects should not be defined with external linkage if they are referenced in only one translation unit

Reference

[Koenig 56-57]

Implemented by QAC messages:

1504	The object '%1s' is only referenced in the translation unit where it is defined.
1505	The function '%1s' is only referenced in the translation unit where it is defined.

8.8 Required Rule-8.8 — The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage

Amplification

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

Implemented by QAC messages:

3224 This identifier has previously been declared with internal linkage but is not declared here with the static storage class specifier.

8.8 Advisory Rule-8.9 **An object should be defined at block scope if its identifier only appears in a single function**

Implemented by QAC messages:

1514 The object '%1s' is only referenced by function '%2s', in the translation unit where it is defined

3218 File scope static, '%s', is only accessed in one function.

8.8 Required Rule-8.10 **An inline function shall be declared with the static storage class**

Reference

C99 [Unspecified 20; Undefined 67]

Implemented by QAC messages:

3240 inline function '%s' is being defined with external linkage.

3243 inline function '%s' is also an 'external definition'.

8.8 Advisory Rule-8.11 **When an array with external linkage is declared, its size should be explicitly specified**

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.

Implemented by QAC messages:

3684 Array declared with unknown size.

8.8 Required Rule-8.12 **Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique**

8.8 Advisory Rule-8.13 **A pointer should point to a const-qualified type whenever possible**

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.

Implemented by QAC messages:

3673 The object addressed by the pointer parameter '%s' is not modified and so the pointer could be of type 'pointer to const'.

8.8 Required Rule-8.14 **The restrict type qualifier shall not be used**

Reference

C99 [Undefined 65,66]

Implemented by QAC messages:

5137 Use of 'restrict' type qualifier.

8.9 Initialization

8.9 Mandatory Rule-9.1 **The value of an object with automatic storage duration shall not be read before it has been set**

Amplification

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

Reference C90 [Undefined 41], C99 [Undefined 10]

Implemented by QAC messages:

2883 This 'goto' statement will always bypass the initialization of local variables.
2961 Definite: Using value of uninitialized automatic object '%s'.
2962 Apparent: Using value of uninitialized automatic object '%s'.
2971 Definite: Passing address of uninitialized object '%s' to a function parameter declared as a pointer to const.
2972 Apparent: Passing address of uninitialized object '%s' to a function parameter declared as a pointer to const.

8.9 Required Rule-9.2 **The initializer for an aggregate or union shall be enclosed in braces**

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.

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.

Reference C90 [Undefined 42], C99 [Undefined 77]

Implemented by QAC messages:

0693 Struct initializer is missing the optional {.
0694 Array initializer is missing the optional {.

8.9 Required Rule-9.3 **Arrays shall not be partially initialized**

Amplification

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

Exception

1. An initializer of the form { 0 } may be used to explicitly initialise 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.

Implemented by QAC messages:

0686 Array has fewer initializers than its declared size. Default initialization is applied to the remainder of the array elements.

8.9 Required Rule-9.4 **An element of an object shall not be initialized more than once**

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 field names they apply to (elements having no initialization value assume the default for uninitialized objects).

8.9 Required Rule-9.5 **Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly**

Amplification

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

8.10 The essential type model

8.10 Required Rule-10.1 Operands shall not be of an inappropriate essential type.

Amplification

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 and indicate why each restriction is imposed.

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
<< >>	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.

Exception

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

Reference

C90 [Implementation 14,17,19,32], C99 [Undefined 49; Implementation J3.4(2,5),J3.5(5),J3.9(6)]

Implemented by QAC messages:

3101 Unary '-' applied to an operand of type unsigned int or unsigned long gives an unsigned result.
3102 Unary '-' applied to an operand whose underlying type is unsigned.
4500 An expression of 'essentially Boolean' type (%1s) is being used as an array subscript.
4501 An expression of 'essentially Boolean' type (%1s) is being used as the %2s operand of this arithmetic operator (%3s).
4502 An expression of 'essentially Boolean' type (%1s) is being used as the %2s operand of this bitwise operator (%3s).
4503 An expression of 'essentially Boolean' type (%1s) is being used as the left-hand operand of this shift operator (%2s).
4504 An expression of 'essentially Boolean' type (%1s) is being used as the right-hand operand of this shift operator (%2s).
4505 An expression of 'essentially Boolean' type (%1s) is being used as the %2s operand of this relational operator (%3s).
4507 An expression of 'essentially Boolean' type (%1s) is being used as the operand of this increment/decrement operator (%2s).
4510 An expression of 'essentially character' type (%1s) is being used as an array subscript.
4511 An expression of 'essentially character' type (%1s) is being used as the %2s operand of this arithmetic operator (%3s).
4512 An expression of 'essentially character' type (%1s) is being used as the %2s operand of this bitwise operator (%3s).
4513 An expression of 'essentially character' type (%1s) is being used as the left-hand operand of this shift operator (%2s).
4514 An expression of 'essentially character' type (%1s) is being used as the right-hand operand of this shift operator (%2s).
4518 An expression of 'essentially character' type (%1s) is being used as the %2s operand of this logical operator (%3s).
4519 An expression of 'essentially character' type (%1s) is being used as the first operand of this conditional operator (%2s).
4521 An expression of 'essentially enum' type (%1s) is being used as the %2s operand of this arithmetic operator (%3s).
4522 An expression of 'essentially enum' type (%1s) is being used as the %2s operand of this bitwise operator (%3s).
4523 An expression of 'essentially enum' type (%1s) is being used as the left-hand operand of this shift operator (%2s).
4524 An expression of 'essentially enum' type (%1s) is being used as the right-hand operand of this shift operator (%2s).
4527 An expression of 'essentially enum' type is being used as the operand of this increment/decrement operator.
4528 An expression of 'essentially enum' type (%1s) is being used as the %2s operand of this logical operator (%3s).
4529 An expression of 'essentially enum' type (%1s) is being used as the first operand of this conditional operator (%2s).
4532 An expression of 'essentially signed' type (%1s) is being used as the %2s operand of this bitwise operator (%3s).

4533 An expression of 'essentially signed' type (%1s) is being used as the left-hand operand of this shift operator (%2s).
4534 An expression of 'essentially signed' type (%1s) is being used as the right-hand operand of this shift operator (%2s).
4538 An expression of 'essentially signed' type (%1s) is being used as the %2s operand of this logical operator (%3s).
4539 An expression of 'essentially signed' type (%1s) is being used as the first operand of this conditional operator (%2s).
4542 A non-negative constant expression of 'essentially signed' type (%1s) is being used as the %2s operand of this bitwise operator (%3s).
4543 A non-negative constant expression of 'essentially signed' type (%1s) is being used as the left-hand operand of this shift operator (%2s).
4544 A non-negative constant expression of 'essentially signed' type (%1s) is being used as the right-hand operand of this shift operator (%2s).
4548 A non-negative constant expression of 'essentially signed' type (%1s) is being used as the %2s operand of this logical operator (%3s).
4549 A non-negative constant expression of 'essentially signed' type (%1s) is being used as the first operand of this conditional operator (%2s).
4558 An expression of 'essentially unsigned' type (%1s) is being used as the %2s operand of this logical operator (%3s).
4559 An expression of 'essentially unsigned' type (%1s) is being used as the first operand of this conditional operator (%2s).
4568 An expression of 'essentially floating' type (%1s) is being used as the %2s operand of this logical operator (%3s).
4569 An expression of 'essentially floating' type (%1s) is being used as the first operand of this conditional operator (%2s).

8.10 Required Rule-10.2 **Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations**

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.

Implemented by QAC messages:

1810 An operand of 'essentially character' type is being added to another operand of 'essentially character' type.
1811 An operand of 'essentially character' type is being subtracted from an operand of 'essentially signed' type.
1812 An operand of 'essentially character' type is being subtracted from an operand of 'essentially unsigned' type.
1813 An operand of 'essentially character' type is being balanced with an operand of 'essentially floating' type in this arithmetic operation.

8.10 Required 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.**

Amplification

The following operations are covered by this rule:

1. Assigning 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.

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.

Reference C90 [Undefined 15,17; Implementation 16], C99 [Undefined 15,16,20,21; Implementation 3.5(4)]

Implemented by QAC messages:

0570 This switch case label of 'essential type' '%1s', is not consistent with a controlling expression of essential type '%2s'.
0572 This switch case label of 'essential type' '%1s' is not consistent with a controlling expression which has an essential type of lower rank (%2s).
1257 An integer constant suffixed with L or LL is being converted to a type of lower rank on assignment.
1264 A suffixed floating constant is being converted to a different floating type on assignment.
1265 An unsuffixed floating constant is being converted to a different floating type on assignment.
1266 A floating constant is being converted to integral type on assignment.

1291	An integer constant of 'essentially unsigned' type is being converted to signed type on assignment.
1292	An integer constant of 'essentially signed' type is being converted to type char on assignment.
1293	An integer constant of 'essentially unsigned' type is being converted to type char on assignment.
1294	An integer constant of 'essentially signed' type is being converted to type _Bool on assignment.
1295	An integer constant of 'essentially unsigned' type is being converted to type _Bool on assignment.
1296	An integer constant of 'essentially signed' type is being converted to enum type on assignment.
1297	An integer constant of 'essentially unsigned' type is being converted to enum type on assignment.
1298	An integer constant of 'essentially signed' type is being converted to floating type on assignment.
1299	An integer constant of 'essentially unsigned' type is being converted to floating type on assignment.
2850	Constant: Implicit conversion to a signed integer type of insufficient size.
2851	Definite: Implicit conversion to a signed integer type of insufficient size.
2852	Apparent: Implicit conversion to a signed integer type of insufficient size.
2900	Constant: Positive integer value truncated by implicit conversion to a smaller unsigned type.
2901	Definite: Positive integer value truncated by implicit conversion to a smaller unsigned type.
2902	Apparent: Positive integer value truncated by implicit conversion to a smaller unsigned type.
4401	An expression of 'essentially Boolean' type (%1s) is being converted to character type, '%2s' on assignment.
4402	An expression of 'essentially Boolean' type (%1s) is being converted to enum type, '%2s' on assignment.
4403	An expression of 'essentially Boolean' type (%1s) is being converted to signed type, '%2s' on assignment.
4404	An expression of 'essentially Boolean' type (%1s) is being converted to unsigned type, '%2s' on assignment.
4405	An expression of 'essentially Boolean' type (%1s) is being converted to floating type, '%2s' on assignment.
4410	An expression of 'essentially character' type (%1s) is being converted to Boolean type, '%2s' on assignment.
4412	An expression of 'essentially character' type (%1s) is being converted to enum type, '%2s' on assignment.
4413	An expression of 'essentially character' type (%1s) is being converted to signed type, '%2s' on assignment.
4414	An expression of 'essentially character' type (%1s) is being converted to unsigned type, '%2s' on assignment.
4415	An expression of 'essentially character' type (%1s) is being converted to floating type, '%2s' on assignment.
4420	An expression of 'essentially enum' type (%1s) is being converted to Boolean type, '%2s' on assignment.
4421	An expression of 'essentially enum' type (%1s) is being converted to character type, '%2s' on assignment.
4422	An expression of 'essentially enum' type (%1s) is being converted to a different enum type, '%2s' on assignment.
4423	An expression of 'essentially enum' type (%1s) is being converted to signed type, '%2s' on assignment.
4424	An expression of 'essentially enum' type (%1s) is being converted to unsigned type, '%2s' on assignment.
4425	An expression of 'essentially enum' type (%1s) is being converted to floating type, '%2s' on assignment.
4430	An expression of 'essentially signed' type (%1s) is being converted to Boolean type, '%2s' on assignment.
4431	An expression of 'essentially signed' type (%1s) is being converted to character type, '%2s' on assignment.
4432	An expression of 'essentially signed' type (%1s) is being converted to enum type, '%2s' on assignment.
4434	A non-constant expression of 'essentially signed' type (%1s) is being converted to unsigned type, '%2s' on assignment.
4435	A non-constant expression of 'essentially signed' type (%1s) is being converted to floating type, '%2s' on assignment.
4436	A constant expression of 'essentially signed' type (%1s) is being converted to unsigned type, '%2s' on assignment.
4437	A constant expression of 'essentially signed' type (%1s) is being converted to floating type, '%2s' on assignment.
4440	An expression of 'essentially unsigned' type (%1s) is being converted to Boolean type, '%2s' on assignment.
4441	An expression of 'essentially unsigned' type (%1s) is being converted to character type, '%2s' on assignment.
4442	An expression of 'essentially unsigned' type (%1s) is being converted to enum type, '%2s' on assignment.
4443	A non-constant expression of 'essentially unsigned' type (%1s) is being converted to a wider signed type, '%2s' on assignment.
4445	An expression of 'essentially unsigned' type (%1s) is being converted to floating type, '%2s' on assignment.
4446	A non-constant expression of 'essentially unsigned' type (%1s) is being converted to signed type, '%2s' on assignment.
4447	A constant expression of 'essentially unsigned' type (%1s) is being converted to signed type, '%2s' on assignment.
4450	An expression of 'essentially floating' type (%1s) is being converted to Boolean type, '%2s' on assignment.
4451	An expression of 'essentially floating' type (%1s) is being converted to character type, '%2s' on assignment.
4452	An expression of 'essentially floating' type (%1s) is being converted to enum type, '%2s' on assignment.
4453	An expression of 'essentially floating' type (%1s) is being converted to signed type, '%2s' on assignment.
4454	An expression of 'essentially floating' type (%1s) is being converted to unsigned type, '%2s' on assignment.
4460	A non-constant expression of 'essentially signed' type (%1s) is being converted to narrower signed type, '%2s' on assignment.
4461	A non-constant expression of 'essentially unsigned' type (%1s) is being converted to narrower unsigned type, '%2s' on assignment.
4462	A non-constant expression of 'essentially floating' type (%1s) is being converted to narrower floating type, '%2s' on assignment.
4463	A constant expression of 'essentially signed' type (%1s) is being converted to narrower signed type, '%2s' on assignment.
4464	A constant expression of 'essentially unsigned' type (%1s) is being converted to narrower unsigned type, '%2s' on assignment.

4465 A constant expression of 'essentially floating' type (%1s) is being converted to narrower floating type, '%2s' on assignment.

8.10 Required Rule-10.4 **Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category**

Amplification

This rule applies to operators that are described in usual arithmetic conversions [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 2nd and 3rd operands of the ternary operator are covered by this rule.

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

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.

Reference

C90 [Undefined 15,17], C99 [Undefined 15,16,20,21]

Implemented by QAC messages:

1800 The %1s operand (essential type: '%2s') will be implicitly converted to a floating type, '%3s', in this arithmetic operation.
1802 The %1s operand (essential type: '%2s') will be implicitly converted to a floating type, '%3s', in this relational operation.
1803 The %1s operand (essential type: '%2s') will be implicitly converted to a floating type, '%3s', in this equality operation.
1804 The %1s operand (essential type: '%2s') will be implicitly converted to a floating type, '%3s', in this conditional operation.
1820 The %1s operand is non-constant and 'essentially signed' (%2s) but will be implicitly converted to an unsigned type (%3s) in this arithmetic operation.
1821 The %1s operand is non-constant and 'essentially signed' (%2s) but will be implicitly converted to an unsigned type (%3s) in this bitwise operation.
1822 The %1s operand is non-constant and 'essentially signed' (%2s) but will be implicitly converted to an unsigned type (%3s) in this relational operation.
1823 The %1s operand is non-constant and 'essentially signed' (%2s) but will be implicitly converted to an unsigned type (%3s) in this equality operation.
1824 The %1s operand is non-constant and 'essentially signed' (%2s) but will be implicitly converted to an unsigned type (%3s) in this conditional operation.
1830 The %1s operand is constant, 'essentially signed' (%2s) and negative but will be implicitly converted to an unsigned type (%3s) in this arithmetic operation.
1831 The %1s operand is constant, 'essentially signed' (%2s) and negative but will be implicitly converted to an unsigned type (%3s) in this bitwise operation.
1832 The %1s operand is constant, 'essentially signed' (%2s) and negative but will be implicitly converted to an unsigned type (%3s) in this relational operation.
1833 The %1s operand is constant, 'essentially signed' (%2s) and negative but will be implicitly converted to an unsigned type (%3s) in this equality operation.
1834 The %1s operand is constant, 'essentially signed' (%2s) and negative but will be implicitly converted to an unsigned type (%3s) in this conditional operation.
1840 The %1s operand is constant, 'essentially signed' (%2s) and non-negative but will be implicitly converted to an unsigned type (%3s) in this arithmetic operation.
1841 The %1s operand is constant, 'essentially signed' (%2s) and non-negative but will be implicitly converted to an unsigned type (%3s) in this bitwise operation.
1842 The %1s operand is constant, 'essentially signed' (%2s) and non-negative but will be implicitly converted to an unsigned type (%3s) in this relational operation.
1843 The %1s operand is constant, 'essentially signed' (%2s) and non-negative but will be implicitly converted to an unsigned type (%3s) in this equality operation.
1844 The %1s operand is constant, 'essentially signed' (%2s) and non-negative but will be implicitly converted to an unsigned type (%3s) in this conditional operation.
1850 The %1s operand is 'essentially unsigned' (%2s) but will be implicitly converted to a signed type (%3s) in this arithmetic operation.
1851 The %1s operand is 'essentially unsigned' (%2s) but will be implicitly converted to a signed type (%3s) in this bitwise operation.
1852 The %1s operand is 'essentially unsigned' (%2s) but will be implicitly converted to a signed type (%3s) in this relational operation.

1853 The %1s operand is 'essentially unsigned' (%2s) but will be implicitly converted to a signed type (%3s) in this equality operation.
1854 The %1s operand is 'essentially unsigned' (%2s) but will be implicitly converted to a signed type (%3s) in this conditional operation.
1860 The operands of this arithmetic operator are of different 'essential signedness' but will generate a result of type 'signed int'.
1861 The operands of this bitwise operator are of different 'essential signedness' but will generate a result of type 'signed int'.
1862 The operands of this relational operator are of different 'essential signedness' but will both be promoted to 'signed int' for comparison.
1863 The operands of this equality operator are of different 'essential signedness' but will both be promoted to 'signed int' for comparison.
1864 The 2nd and 3rd operands of this conditional operator are of different 'essential signedness'. The result will be in the promoted type 'signed int'.
1880 The operands of this relational operator are expressions of different 'essential type' categories (%1s and %2s).
1881 The operands of this equality operator are expressions of different 'essential type' categories (%1s and %2s).
1882 The 2nd and 3rd operands of this conditional operator are expressions of different 'essential type' categories (%1s and %2s).

8.10 Advisory Rule-10.5 **The value of an expression should not be cast to an inappropriate essential type**

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 that 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

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.

Implemented by QAC messages:

4301 An expression of 'essentially Boolean' type (%1s) is being cast to character type '%2s'.
4302 An expression of 'essentially Boolean' type (%1s) is being cast to enum type '%2s'.
4303 An expression of 'essentially Boolean' type (%1s) is being cast to signed type '%2s'.
4304 An expression of 'essentially Boolean' type (%1s) is being cast to unsigned type '%2s'.
4305 An expression of 'essentially Boolean' type (%1s) is being cast to floating type '%2s'.
4310 An expression of 'essentially character' type (%1s) is being cast to Boolean type, '%2s'.
4312 An expression of 'essentially character' type (%1s) is being cast to enum type, '%2s'.
4315 An expression of 'essentially character' type (%1s) is being cast to floating type, '%2s'.
4320 An expression of 'essentially enum' type (%1s) is being cast to Boolean type, '%2s'.
4322 An expression of 'essentially enum' type (%1s) is being cast to a different enum type, '%2s'.
4330 An expression of 'essentially signed' type (%1s) is being cast to Boolean type '%2s'.
4332 An expression of 'essentially signed' type (%1s) is being cast to enum type, '%2s'.
4340 An expression of 'essentially unsigned' type (%1s) is being cast to Boolean type '%2s'.
4342 An expression of 'essentially unsigned' type (%1s) is being cast to enum type '%2s'.
4350 An expression of 'essentially floating' type (%1s) is being cast to Boolean type '%2s'.
4351 An expression of 'essentially floating' type (%1s) is being cast to character type '%2s'.
4352 An expression of 'essentially floating' type (%1s) is being cast to enum type, '%2s'.

8.10 Required Rule-10.6 — The value of a composite expression shall not be assigned to an object with wider essential type

Amplification

This rule covers the assigning operations described in Rule 10.3.

Implemented by QAC messages:

4490	A composite expression of 'essentially signed' type (%1s) is being converted to wider signed type, '%2s' on assignment.
4491	A composite expression of 'essentially unsigned' type (%1s) is being converted to wider unsigned type, '%2s' on assignment.
4492	A composite expression of 'essentially floating' type (%1s) is being converted to wider floating type, '%2s' on assignment.
4499	An expression which is the result of a ~ or << operation has been converted to a wider essential type on assignment.

8.10 Required 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

Implemented by QAC messages:

1890	A composite expression of 'essentially signed' type (%1s) is being implicitly converted to a wider signed type, '%2s'.
1891	A composite expression of 'essentially unsigned' type (%1s) is being implicitly converted to a wider unsigned type, '%2s'.
1892	A composite expression of 'essentially floating' type (%1s) is being implicitly converted to a wider floating type, '%2s'.
1893	The 2nd and 3rd operands of this conditional operator are both 'essentially signed' ('%1s' and '%2s') but one is a composite expression of a narrower type than the other.
1894	The 2nd and 3rd operands of this conditional operator are both 'essentially unsigned' ('%1s' and '%2s') but one is a composite expression of a narrower type than the other.
1895	The 2nd and 3rd operands of this conditional operator are both 'essentially floating' ('%1s' and '%2s') but one is a composite expression of a narrower type than the other.
4397	An expression which is the result of a ~ or << operation has not been cast to its essential type.

8.10 Required Rule-10.8 — The value of a composite expression shall not be cast to a different essential type category or a wider essential type

Implemented by QAC messages:

4390	A composite expression of 'essentially signed' type (%1s) is being cast to a wider signed type, '%2s'.
4391	A composite expression of 'essentially unsigned' type (%1s) is being cast to a wider unsigned type, '%2s'.
4392	A composite expression of 'essentially floating' type (%1s) is being cast to a wider floating type, '%2s'.
4393	A composite expression of 'essentially signed' type (%1s) is being cast to a different type category, '%2s'.
4394	A composite expression of 'essentially unsigned' type (%1s) is being cast to a different type category, '%2s'.
4395	A composite expression of 'essentially floating' type (%1s) is being cast to a different type category, '%2s'.
4398	An expression which is the result of a ~ or << operation has been cast to a different essential type category.
4399	An expression which is the result of a ~ or << operation has been cast to a wider type.

8.11 Pointer type conversions

8.11 Required Rule-11.1 — Conversions shall not be performed between a pointer to a function and any other type

Amplification

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

Reference C90 [Undefined 24,27-29], C99 [Undefined 21,23,39,41]

Implemented by QAC messages:

0302 [u] Cast between a pointer to function and a floating type.
0305 [l] Cast between a pointer to function and an integral type.
0307 [u] Cast between a pointer to object and a pointer to function.
0313 Casting to different function pointer type.

8.11 Required Rule-11.2 **Conversions shall not be performed between a pointer to an incomplete type and any other type**

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.

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.

Reference C90 [Undefined 29], C99 [Undefined 21,22,41]

Implemented by QAC messages:

0308 Non-portable cast involving pointer to an incomplete type.

8.11 Required Rule-11.3 **A cast shall not be performed between a pointer to object type and a pointer to a different object type**

Amplification

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

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.

Reference C90 [Undefined 20], C99 [Undefined 22,34]

Implemented by QAC messages:

0310 Casting to different object pointer type.
3305 Pointer cast to stricter alignment.

8.11 Advisory Rule-11.4 **A conversion should not be performed between a pointer to object and an integer type**

Amplification

A pointer should not be converted into an integer.

An integer should not be converted into a pointer.

Exception

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

Reference C90 [Implementation 24], C99 [Undefined 21,22; Implementation J.3.7(1)]

Implemented by QAC messages:

0303 [l] Cast between a pointer to volatile object and an integral type.
0306 [l] Cast between a pointer to object and an integral type.
0360 An expression of pointer type is being converted to type _Bool on assignment.

0361 An expression of pointer type is being cast to type _Bool.
0362 An expression of essentially Boolean type is being cast to a pointer.

8.11 Advisory Rule-11.5 **A conversion should not be performed from pointer to void into pointer to object**

Exception

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

Reference

C99 [Undefined 22]

Implemented by QAC messages:

0316 [l] Cast from a pointer to void to a pointer to object type.
0317 [l] Implicit conversion from a pointer to void to a pointer to object type.

8.11 Required Rule-11.6 **A cast shall not be performed between pointer to void and an arithmetic type**

Exception

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

Reference

C90 [Undefined 29; Implementation 24], C99 [Undefined 21,22,41; Implementation J.3.7(1)]

Implemented by QAC messages:

0306 [l] Cast between a pointer to object and an integral type.

8.11 Required Rule-11.7 **A cast shall not be performed between pointer to object and a non-integer arithmetic type**

Amplification

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

- essentially Boolean;
- essentially character;
- essentially enum;
- essentially floating.

Reference

C90 [Undefined 29]; C99 [Undefined 21,22,41]

Implemented by QAC messages:

0301 [u] Cast between a pointer to object and a floating type.

8.11 Required Rule-11.8 **A cast shall not remove any const or volatile qualification from the type pointed to by a pointer**

Reference

C90 [Undefined 39-40], C99 [Undefined 61-62]

Implemented by QAC messages:

0311 Dangerous pointer cast results in loss of const qualification.
0312 Dangerous pointer cast results in loss of volatile qualification.

8.11 Required Rule-11.9 **The macro NULL shall be the only permitted form of integer null pointer constant**

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 2nd operand of a ?: operator whose 3rd operand is a pointer;
- As the 3rd operand of a ?: operator whose 2nd 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.

Implemented by QAC messages:

3003 This character constant is being interpreted as a NULL pointer constant.
3004 This integral constant expression is being interpreted as a NULL pointer constant.

8.12 Expressions

8.12 Advisory Rule-12.1 **The precedence of operators within expressions should be made explicit**

Amplification

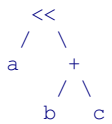
The following table is used in the definition of this rule.

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:



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.

Implemented by QAC messages:

3389 Extra parentheses recommended to clarify the ordering of a % operator and another arithmetic operator (* / % + -).
3391 Extra parentheses recommended. A conditional operation is the operand of another conditional operator.
3392 Extra parentheses recommended. A shift, relational or equality operation is the operand of a second identical operator.
3394 Extra parentheses recommended. A shift, relational or equality operation is the operand of a different operator with the

same precedence.

3395 Extra parentheses recommended. A * or / operation is the operand of a + or - operator.

3396 Extra parentheses recommended. A binary operation is the operand of a conditional operator.

3397 Extra parentheses recommended. A binary operation is the operand of a binary operator with different precedence.

8.12 Required 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

Reference

C90 [Undefined 32], C99 [Undefined 48]

Implemented by QAC messages:

0499 Right operand of shift operator is greater than or equal to the width of the essential type of the left operand.

2790 Constant: Right hand operand of shift operator is negative or too large.

8.12 Advisory Rule-12.3 — The comma operator should not be used

Implemented by QAC messages:

3417 The comma operator has been used outside a 'for' statement.

3418 The comma operator has been used in a 'for' statement.

8.12 Advisory Rule-12.4 — Evaluation of constant expressions should not lead to unsigned integer wrap-around

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.

Implemented by QAC messages:

2910 Constant: Wraparound in unsigned arithmetic operation.

8.13 Side-effects

8.13 Required Rule-13.1 — Initializer lists shall not contain persistent side-effects

Reference

C99 [Unspecified 17,22]

8.13 Required Rule-13.2 — The value of an expression and its persistent side-effects shall be the same under all permitted evaluation orders

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.

Reference

C90 [Unspecified 7-9; Undefined 18], C99 [Unspecified 15-18; Undefined 32]

Implemented by QAC messages:

0400 [U] '%s' is modified more than once between sequence points - evaluation order unspecified.
0401 [U] '%s' may be modified more than once between sequence points - evaluation order unspecified.
0402 [U] '%s' is modified and accessed between sequence points - evaluation order unspecified.
0403 [U] '%s' may be modified and accessed between sequence points - evaluation order unspecified.

8.13 Advisory 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

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.

Reference

C90 [Unspecified 8], C99 [Unspecified 15]

Implemented by QAC messages:

3440 Using the value resulting from a ++ or -- operation.

8.13 Advisory Rule-13.4

The result of an assignment operator should not be used

Amplification

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

Reference

C90 [Unspecified 8], C99 [Unspecified 15], [Koenig 6]

Implemented by QAC messages:

3226 The result of an assignment is being used in an arithmetic operation or another assigning operation.
3326 The result of an assignment is being used in a logical operation.

8.13 Required Rule-13.5

The right hand operand of a logical && or || operator shall not contain persistent side effects

Implemented by QAC messages:

3415 Right hand operand of '&&' or '||' is an expression with possible side effects.

8.13 Mandatory Rule-13.6

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

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.

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.

Reference

C99 [Unspecified 21]

Implemented by QAC messages:

3307 The operand of 'sizeof' is an expression with implied side effects, but they will not be evaluated.

8.14 Control statement expressions

8.14 Required Rule-14.1

A loop counter shall not have essentially floating type

Implemented by QAC messages:

3340 Floating point variable used as 'for' loop control variable.
3342 Controlling expression of 'for' loop is a floating point comparison.

8.14 Required Rule-14.2 **A for loop shall be well-formed**

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.

Exception

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

Implemented by QAC messages:

2461 Loop control variable, %s, has file scope.
2462 The variable initialized in the first expression of this 'for' statement is not the variable identified as the 'loop control variable' (%s).
2463 The variable incremented in the third expression of this 'for' statement is not the variable identified as the 'loop control variable' (%s).
2464 Loop control variable, %s, modified twice in for-loop header.
2467 Loop control variable, %s, is not modified inside loop.
2469 Loop control variable in this 'for' statement, %s, is modified in the body of the loop.
2471 Unable to identify a loop control variable.
2472 More than one possible loop control variable.

8.14 Required Rule-14.3 **Controlling expressions shall not be invariant**

Amplification

This rule applies to:

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

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.

Implemented by QAC messages:

2990 The value of this loop controlling expression is always 'true'.
2991 The value of this 'if' controlling expression is always 'true'.
2992 The value of this 'if' controlling expression is always 'false'.
2993 The value of this 'do - while' loop controlling expression is always 'false'. The loop will only be executed once.
2994 The value of this 'while' or 'for' loop controlling expression is always 'false'. The loop will not be entered.

8.14 Required Rule-14.4 — The controlling expression of an if-statement and the controlling expression of an iteration-statement shall have essentially Boolean type

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.

Implemented by QAC messages:

3344 Controlling expression is not an 'essentially Boolean' expression.

8.15 Control flow

8.15 Advisory Rule-15.1 — The goto statement should not be used

Implemented by QAC messages:

2001 A 'goto' statement has been used.

8.15 Required Rule-15.2 — The goto statement shall jump to a label declared later in the same function

8.15 Required 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

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.

Implemented by QAC messages:

3311 [u] An earlier jump to this statement will bypass the initialization of local variables.

8.15 Advisory Rule-15.4 — There should be no more than one break or goto statement used to terminate any iteration statement

Implemented by QAC messages:

0771 More than one 'break' statement has been used to terminate this iteration statement.

8.15 Advisory Rule-15.5 — A function should have a single point of exit at the end

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.

Reference

[IEC 61508-3 Table B.9], [ISO 26262-6 Table 8]

Implemented by QAC messages:

2889 This function has more than one 'return' path.

8.15 Required Rule-15.6 — The body of an iteration-statement or a selection-statement shall be a compound-statement

Amplification

The body of an iteration-statement (while, do ... while or for) or a selection-statement (if, else, switch) shall be a compound-statement.

Exception

An if statement immediately following an else need not be contained within a compound statement.

Reference

[Koenig 24]

Implemented by QAC messages:

2212 Body of control statement is not enclosed within braces.

2214 Body of control statement is on the same line and is not enclosed within braces.
3402 Braces are needed to clarify the structure of this 'if'-'if'-'else' statement.

8.15 Required Rule-15.7 All if ... else if constructs shall be terminated with an else statement

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.

Implemented by QAC messages:

2004 No concluding 'else' exists in this 'if'-'else'-'if' statement.

8.16 Switch statements

8.16 Required Rule-16.1 All switch statements shall be well-formed

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<sub>opt</sub> break;
   C90: { declaration-list<sub>opt</sub> statement-list<sub>opt</sub> break; }
   C99: { block-item-list<sub>opt</sub> 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.

Implemented by QAC messages:

2008 Code statements precede the first label in this 'switch' construct.
3234 Declarations precede the first label in this 'switch' construct.

8.16 Required Rule-16.2 A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement

Implemented by QAC messages:

2019 'Switch' label is located within a nested code block.

8.16 Required Rule-16.3 An unconditional break statement shall terminate every switch-clause

Implemented by QAC messages:

2003 The preceding 'switch' clause is not empty and does not end with a 'jump' statement. Execution will fall through.
2020 Final 'switch' clause does not end with an explicit 'jump' statement.

8.16 Required Rule-16.4 Every switch statement shall have a default label

Amplification

The switch-clause following the default label shall, prior to the terminating break statement, contain either:

- A statement, or
- A comment.

Implemented by QAC messages:

2002 No 'default' label found in this 'switch' statement.

8.16 Required Rule-16.5 A default label shall appear as either the first or the last switch label of a switch statement

Implemented by QAC messages:

2009 This 'default' label is not the final 'case' label within the 'switch' block.

8.16 Required Rule-16.6 Every switch statement shall have at least two switch-clauses

Implemented by QAC messages:

3315 This 'switch' statement contains only a single path - it is redundant.

8.16 Required Rule-16.7 A switch-expression shall not have essentially Boolean type

Implemented by QAC messages:

0735 Using relational or logical operators in a 'switch' expression is usually a programming error.

8.17 Functions

8.17 Required Rule-17.1 The features of <stdarg.h> shall not be used

Amplification

None of va_list, va_arg, va_start, va_end and, for C99, va_copy shall be used.

Reference C90 [Unspecified 15; Undefined 25,45,61,70-76], C99 [Unspecified 35; Undefined 37,81,129,130,131,133,135]

Implemented by QAC messages:

1337 Function defined with a variable number of parameters.
5130 Use of standard header file <stdarg.h>.

8.17 Required Rule-17.2 Functions shall not call themselves, either directly or indirectly

Implemented by QAC messages:

1520 Functions are indirectly recursive.
3670 Recursive call to function containing this call.

8.17 Mandatory Rule-17.3 A function shall not be declared implicitly

Reference C90 [Undefined 22-24]

Implemented by QAC messages:

3335 No function declaration. Implicit declaration inserted: 'extern int %s();'.

8.17 Mandatory Rule-17.4 — All exit paths from a function with non-void return type shall have an explicit return statement with an expression

Reference C90 [Undefined 43], C99 [Undefined 82]

Implemented by QAC messages:

0745	[U] 'return;' found in '%s()', which has been defined with a non-'void' return type.
2887	Function 'main' ends with an implicit 'return' statement.
2888	This function has been declared with a non-void 'return' type but ends with an implicit 'return ;' statement.
3113	[U] 'return' statement includes no expression but function '%s()' is implicitly of type 'int'.
3114	[U] Function '%s()' is implicitly of type 'int' but ends without returning a value.

8.17 Advisory Rule-17.5 — The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements

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.

8.17 Mandatory Rule-17.6 — The declaration of an array parameter shall not contain the static keyword between the [ ]

Reference C99 [Undefined 71]

8.17 Required Rule-17.7 — The value returned by a function having non-void return type shall be used

Implemented by QAC messages:

3200	'%s' returns a value which is not being used.

8.17 Advisory Rule-17.8 — A function parameter should not be modified

---

8.18 Pointers and arrays

8.18 Required Rule-18.1 — A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand

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).

Reference C90 [Undefined 30], C99 [Undefined 43,44,46,59]

Implemented by QAC messages:

2930	Constant: Computing an invalid pointer value.
2931	Definite: Computing an invalid pointer value.
2932	Apparent: Computing an invalid pointer value.

8.18 Required Rule-18.2 **Subtraction between pointers shall only be applied to pointers that address elements of the same array**

Reference

C90 [Undefined 31], C99 [Undefined 45]

Implemented by QAC messages:

2771 Definite: These pointers address different objects.
2772 Apparent: These pointers address different objects.

8.18 Required Rule-18.3 **The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object**

Reference

C90 [Undefined 33], C99 [Undefined 50]

Implemented by QAC messages:

2771 Definite: These pointers address different objects.
2772 Apparent: These pointers address different objects.

8.18 Advisory Rule-18.4 **The +, -, += and -= operators should not be applied to an expression of pointer type**

Exception

Subject to Rule-18.2, pointer subtraction between two pointers is allowed.

Implemented by QAC messages:

0488 Performing pointer arithmetic.

8.18 Advisory Rule-18.5 **Declarations should contain no more than two levels of pointer nesting**

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.

Implemented by QAC messages:

3260 Typedef defined with more than 2 levels of indirection.
3261 Member of struct/union defined with more than 2 levels of indirection.
3262 Object defined or declared with more than 2 levels of indirection.
3263 Function defined or declared with a return type which has more than 2 levels of indirection.

8.18 Required 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**

Amplification

The address of an object might be copied by means of:

- Assignment;
- Memory move or copying functions.

Reference

C90 [Undefined 9,26], C99 [Undefined 8,9,40]

Implemented by QAC messages:

3217 Address of automatic object exported to a pointer with linkage or wider scope.
3225 Address of automatic object exported using a function parameter.
3230 Address of automatic object assigned to local pointer with static storage duration.
4140 Address of automatic object exported in function return value.

8.18 Required Rule-18.7 **Flexible array members shall not be declared**

Reference

C99 [Undefined 59]

8.18 Required Rule-18.8 **Variable-length array types shall not be used**

Reference

C99 [Unspecified 21; Undefined 69]

Implemented by QAC messages:

0945 [C99] WARNING. Operand of sizeof is an expression of variable length array type.
1051 [C99] A variable length array has been declared.
1052 [C99] A variable length array of unspecified size has been declared.

8.19 Overlapping storage

8.19 Mandatory Rule-19.1 **An object shall not be assigned or copied to an overlapping object**

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

Reference C90 [Undefined 34,55], C99 [Undefined 51,94]

Implemented by QAC messages:

2776 Definite: Copy between overlapping objects.
2777 Apparent: Copy between overlapping objects.

8.19 Advisory Rule-19.2 **The union keyword should not be used**

Reference C90 [Implementation 27], C99 [Unspecified 10]

Implemented by QAC messages:

0750 A union type specifier has been defined.
0759 An object of union type has been defined.

8.20 Preprocessing directives

8.20 Advisory Rule-20.1 **#include directives should only be preceded by preprocessor directives or comments**

Amplification

The rule shall be applied to the contents of a file before preprocessing occurs.

Reference C90 [Undefined 56], C99 [Undefined 96,97]

Implemented by QAC messages:

5087 Use of #include directive after code fragment.

8.20 Required Rule-20.2 **The ', " or \ characters and the /* or // character sequences shall not occur in a header file name**

Reference C90 [Undefined 14], C99 [Undefined 31]

Implemented by QAC messages:

0813 [U] Using any of the characters ' " or /* in '#include <%s>' gives undefined behaviour.
0814 [U] Using the characters ' or /* in '#include "%s"' gives undefined behaviour.
0831 [E] Use of '\\' in this '#include' line is a PC extension - this usage is non-portable.

8.20 Required Rule-20.3 **The #include directive shall be followed by either a <filename> or "filename" sequence**

Amplification

This rule applies after macro replacement has been performed.

Reference C90 [Undefined 48], C99 [Undefined 85]

Implemented by QAC messages:

0809 [U] The '#include' preprocessing directive has not been followed by <h-char-sequence> or "s-char-sequence".

8.20 Required Rule-20.4 **A macro shall not be defined with the same name as a keyword**

Amplification

This rule applies to all keywords, including those that implement language extensions.

Reference

C90 [Undefined 56], C99 [Undefined 98]

Implemented by QAC messages:

3439 Macro redefines a keyword.

8.20 Advisory Rule-20.5 **#undef should not be used**

Implemented by QAC messages:

0841 Using '#undef'.

8.20 Required Rule-20.6 **Tokens that look like a preprocessing directive shall not occur within a macro argument**

Reference

C90 [Undefined 50], C99 [Undefined 87]

Implemented by QAC messages:

0853 [U] Macro arguments contain a sequence of tokens that has the form of a preprocessing directive.

8.20 Required Rule-20.7 **Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses**

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 parenthesised; it is acceptable for parentheses to be provided in macro arguments.

Reference

[Koenig 78-81]

Implemented by QAC messages:

3410 Macro parameter not enclosed in ().

8.20 Required Rule-20.8 **The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1**

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.

8.20 Required Rule-20.9 **All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #define'd before evaluation**

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.

Implemented by QAC messages:

3332 The macro '%s' used in this '#if' or '#elif' expression is not defined.

8.20 Advisory Rule-20.10 **The # and ## preprocessor operators should not be used**

Reference C90 [Unspecified 12], C99 [Unspecified 25]

Implemented by QAC messages:

0341 Using the stringify operator '#'.
0342 Using the glue operator '##'.

8.20 Required Rule-20.11 **A macro parameter immediately following a # operator shall not immediately be followed by a ## operator**

Reference C90 [Unspecified 12], C99 [Unspecified 25]

8.20 Required 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**

8.20 Required Rule-20.13 **A line whose first token is # shall be a valid preprocessing directive**

Amplification

White-space is permitted between the # and preprocessing tokens.

Implemented by QAC messages:

3115 Unrecognized preprocessing directive has been ignored because of conditional inclusion directives.

8.20 Required Rule-20.14 **All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related**

Implemented by QAC messages:

3317 '#if...' not matched by '#endif' in included file. This is probably an error.
3318 '#else'/'#elif'/'#endif' in included file matched '#if...' in parent file. This is probably an error.

8.21 Standard libraries

8.21 Required Rule-21.1 **#define and #undef shall not be used on a reserved identifier or reserved macro name**

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.

Reference C90 [Undefined 54,57,58,62], C99 [Undefined 93,100,101,104,108]

Implemented by QAC messages:

0836 [U] Definition of macro named 'defined'.
0848 [U] Attempting to #undef '%s', which is a predefined macro name.
0854 [U] Attempting to #define '%s', which is a predefined macro name.

4600 The macro '%1s' is also defined in '<%2s>'.
4601 The macro '%1s' is the name of an identifier in '<%2s>'.

8.21 Required Rule-21.2 A reserved identifier or macro name shall not be declared

Amplification

See the Amplification for Rule-21.1 for a description of the relevant identifiers and macro names.

Reference

C90 [Undefined 57,58], C99 [Undefined 100,108]

Implemented by QAC messages:

0602 [U] The identifier '%s' is reserved for use by the library.
4602 The identifier '%1s' is declared as a macro in '<%2s>'.
4603 The object/function '%1s'is being defined with the same name as an ordinary identifier defined in '<%2s>'.
4604 The object/function '%1s' is being declared with the same name as an ordinary identifier defined in '<%2s>'.
4605 The typedef '%1s' is also defined in '<%2s>'.
4606 The typedef '%1s' has the same name as another ordinary identifier in '<%2s>'.
4607 The enum constant '%1s' has the same name as another ordinary identifier in '<%2s>'.
4608 The tag '%1s' is also defined in '<%2s>'.

8.21 Required Rule-21.3 The memory allocation and deallocation functions of <stdlib.h> shall not be used

Amplification

The identifiers calloc, malloc, realloc and free shall not be used and no macro with one of these names shall be expanded.

Reference

C90 [Unspecified 19; Undefined 91,92; Implementation 69], C99 [Unspecified 41; Undefined 9,168,169,170,171; Implementation J.3.12(37)]

Implemented by QAC messages:

5118 Use of memory allocation or deallocation function: calloc, malloc, realloc or free.

8.21 Required Rule-21.4 The standard header file <setjmp.h> shall not be used

Amplification

None of the facilities that are specified as being provided by <setjmp.h> shall be used.

Reference

C90 [Unspecified 14; Undefined 64-67], C99 [Unspecified 34; Undefined 118-121], [Koenig 74]

Implemented by QAC messages:

5132 Use of standard header file <setjmp.h>.

8.21 Required Rule-21.5 The standard header file <signal.h> shall not be used

Amplification

None of the facilities that are specified as being provided by <signal.h> shall be used.

Reference

C90 [Undefined 68,69; Implementation 48-52], C99 [Undefined 125-127], [Koenig 74]

Implemented by QAC messages:

5123 Use of standard header file <signal.h>.

8.21 Required Rule-21.6 The Standard Library input/output functions shall not be used

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.

Reference

C90 [Unspecified 2-5,16-18; Undefined 77-89; Implementation 53-68], C99 [Unspecified 3-6,37-

39; Undefined 138-166; Implementation J.3.12(16-36)]

Implemented by QAC messages:

5124 Use of standard header file <stdio.h>.

8.21 Required Rule-21.7 The atof, atoi, atol and atoll functions of <stdlib.h> shall not be used

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.

Reference

C90 [Undefined 90], C99 [Undefined 113]

Implemented by QAC messages:

5125 Use of function: atof, atoi, atol or atoll.

8.21 Required Rule-21.8 The library functions abort, exit, getenv and system of <stdlib.h> shall not be used

Amplification

The identifiers abort, exit, getenv and system shall not be used and no macro with one of these names shall be expanded.

Reference

C90 [Undefined 93; Implementation 70-73], C99 [Undefined 172; Implementation J3.12(38-40)]

Implemented by QAC messages:

5126 Use of function: abort, exit, getenv or system.

8.21 Required Rule-21.9 The library functions bsearch and qsort of <stdlib.h> shall not be used

Amplification

The identifiers bsearch and qsort shall not be used and no macro with one of these names shall be expanded.

Reference

C90 [Unspecified 20,21], C99 [Unspecified 43,44; Undefined 1,177]

Implemented by QAC messages:

5135 Use of function: bsearch or qsort.

8.21 Required Rule-21.10 The Standard Library time and date functions shall not be used

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.

Reference

C90 [Unspecified 22; Undefined 97; Implementation 75,76], C99 [Unspecified 45,46; Undefined 154; Implementation J.3.12(41-44)]

Implemented by QAC messages:

5127 Use of standard header file <time.h>.

8.21 Required Rule-21.11 The standard header file <tgmath.h> shall not be used

Amplification

None of the facilities that are specified as being provided by <tgmath.h> shall be used.

Reference

C99 [Undefined 184]

Implemented by QAC messages:

5131 Use of standard header file <tgmath.h>.

8.21 Advisory Rule-21.12 The exception handling features of <fenv.h> should not be used

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.

Reference C99 [Unspecified 27,28; Undefined 109; Implementation J.3.6(9)]

Implemented by QAC messages:

5136 Use of exception handling identifier: feclearexcept, fegetexceptflag, feraiseexcept, fesetexceptflag or fetestexcept.

8.22 Resources

8.22 Required Rule-22.1 **All resources obtained dynamically by means of Standard Library functions shall be explicitly released**

Amplification

The Standard Library functions that allocate resources are malloc, calloc, realloc and fopen.

8.22 Mandatory Rule-22.2 **A block of memory shall only be freed if it was allocated by means of a Standard Library function**

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.

Reference C90 [Undefined 92], C99 [Undefined 169]

8.22 Required Rule-22.3 **The same file shall not be open for read and write access at the same time on different streams**

Amplification

This rule applies to files opened with The Standard Library functions. It may also apply to similar features provided by the execution environment.

Reference C90 [Implementation 59], C99 [Implementation J.3.12(24)]

8.22 Mandatory Rule-22.4 **There shall be no attempt to write to a stream which has been opened as read-only**

8.22 Mandatory Rule-22.5 **A pointer to a FILE object shall not be dereferenced**

Amplification

A pointer to a FILE object shall not be dereferenced directly or indirectly (e.g. by a call to memcpy or memcmp).

8.22 Mandatory Rule-22.6 **The value of a pointer to a FILE shall not be used after the associated stream has been closed**

Reference C99 [Section 7.1.4; Undefined 102,140]

End of Coding Standard Manual
