IAR Embedded Workbench®

C-STAT® Static Analysis Guide

CSTAT-2

EDITION NOTICE

Second edition: May 2015

Part number: CSTAT-2

Internal reference: M19, Hom7.2, IJOA, ISUD.

Contents

C-STAT for static analysis

- Introduction to C-STAT and static analysis
- Using C-STAT from the command line—ichecks and icstat
- Reference information on icstat and ichecks
- Descriptions of C-STAT options

For information about how to use C-STAT in the IAR Embedded Workbench IDE, see the *IDE Project Management and Building Guide*.

Introduction to C-STAT and static analysis

C-STAT is a static analysis tool that tries to find deviations from specific *packages* of coding *rules*. The various packages are:

- Stdchecks

  Contains checks for rules that come from CWE and CERT, as well as checks specific to C-STAT.
- MISRA C:2004

  Contains checks for selected rules of the MISRA C:2004 standard. This standard identifies unsafe code constructs in the C89 standard.
- MISRA C++:2008

  Contains checks for selected rules of the MISRA C++:2008 standard. This standard identifies unsafe code constructs in the 1998 C++ standard.
- MISRA C:2012

  Contains checks for selected rules of the MISRA C:2012 standard. This standard identifies unsafe code constructs in the C99 and C89 standards.

Each MISRA C rule is either *mandatory*, *required*, or *advisory*. The checks for the mandatory and required rules are by default on, whereas the checks for the advisory rules are by default off. Each rule specifies an unsafe code construct. C-STAT tries to find deviations from a rule by running one or more *checks* for the rule.

C-STAT is an integral part of the IAR Embedded Workbench IDE. C-STAT can also be used from the command line, which is useful if you build your project using a make file.

Note: Some checks compute summary information per file that can be used when analyzing other files. How this information is used depends on the order in which the files are analyzed. This means that the exact number of messages can differ, for example when running C-STAT in the IDE as opposed to using the command line tools.

Note: The maximum time for an analysis of a specific file is about 4 minutes. When the time limed is expired, the analysis will continue with the next file.

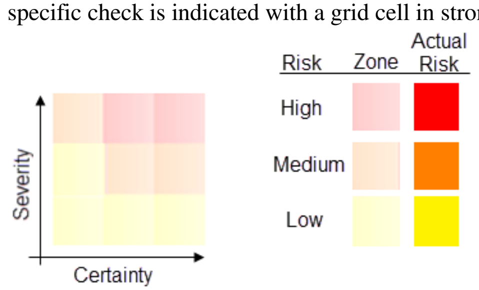
THE CHECKS AND THEIR DOCUMENTATION

A check is a programmatic way of identifying deviations from a rule. Each check has a:

- *Tag*, which is used for referring to the check. For example, `ARR-inv-index-pos`.
- *Default activation*, which can be one of Yes or No.
- *Synopsis*, for example, `Array access may be out of bounds, depending on which path is executed`.
- *Severity level*, which can be Low, Medium, or High.

In addition, the documentation for each check provides information about any vulnerabilities it identifies and a description of the problems that can be caused by code that fails the check, such as memory leaks, undefined or unpredictable behavior, or program crashes. Usually, there are also two source code examples: one that illustrates code that fails the check and generates a message, and one that illustrates code that passes the check. For each check, there is also information about which rules in the different coding standards that the check corresponds to.

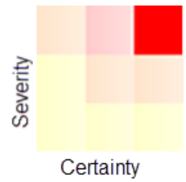
A grid shows the *severity* of the problems that code that does not conform to the rule (non-conformant code) can cause, and the level of *certainty* that the message reflects a true error in the source code. The grid is divided into three *zones*—indicated with pale colors—that reflect the *risks* based on the severity and certainty. The *actual risk* for a specific check is indicated with a grid cell in strong color.



Here follow some example grids.

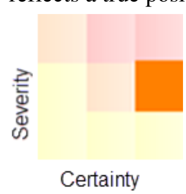
Example 1—high severity and high certainty = high risk

This grid shows a check with high severity and high certainty, which means that it very likely indicates a true bug. While all messages should be investigated, those with a high certainty are more likely to identify real problems in your source code.



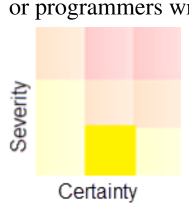
Example 2—medium severity and high certainty = medium risk

This grid shows a check with medium severity and high certainty. A medium severity indicates that, for the code that fails the check, there is a medium risk of causing serious errors in your application. A high certainty means that it is very likely that the message reflects a true positive.



Example 3—low severity and medium certainty = low risk

This grid shows a check with low severity and medium certainty, which indicates that the code probably is safe to use. That the check fails can be due to an offense in a macro, or programmers writing safe, but unusual code.



Using C-STAT from the command line—ichecks and icstat

To use C-STAT from the command line, you need:

- `ichecks.exe`—use the `ichecks` tool to generate a *manifest file* that contains only the checks that you want to run.

- `icstat.exe`—use the `icstat` tool to run a C-STAT static analysis on a project, with the manifest file as input.

For information about the checks, see *C-STAT checks*, page 17.

USING ICSTAT

The input to `icstat` consists of:

- The source files for your application, with the compiler command lines.
- The linker command line for your application.
- A file that lists the enabled checks that will be run (or more specifically, the *tags* for the checks). You create this file using the `ichecks` tool.
- A file where the deviations from the performed checks will be stored in a database.

For an example of how to perform a static analysis using C-STAT, follow these steps based on two example source code files `cstat1.c` and `ctat2.c`. You can find these files in the directory *`target`*`\src`.

To perform a static analysis using C-STAT:

1. Select which checks you want to run by creating a manifest file using `ichecks`, for example like this:

   ```
   ichecks --default stdchecks --output checks.ch
   ```

   The `checks.ch` file lists all the checks that you have selected, in this case, all checks that are enabled by default for the `stdchecks` package (`--default`). The file will look like this:

   ```
   ARR-inv-index-pos
   ARR-inv-index-ptr-pos
   ...
   ```

   To modify the file on check-level, you can manually add or delete checks from the file.

2. Make sure that your project builds without errors.

3. To analyze your application, specify your `icstat` commands. For example like this:

   ```
   icstat --db a.db --checks checks.ch analyze -- iccxxxxx
   compiler_opts cstat1.c

   icstat --db a.db --checks checks.ch analyze -- iccxxxxx
   compiler_opts cstat2.c

   icstat --db a.db --checks checks.ch link_analyze -- ilinkxxxxx
   linker_opts cstat1.o cstat2.o
   ```

Note: *xxxxx* should be replaced with an identifier that is unique to your IAR Embedded Workbench product package. If your product package comes with the IAR XLINK Linker instead of the IAR ILINK Linker, `ilink`*xxxxx* should be `xlink` and the filename extension `o` should be `r`*xx*, where *xx* is a numeric part that identifies your product package.

In these example command lines, `--db` specifies a file where the resulting data base is stored, and the `--checks` option specifies the `checks.ch` manifest file. The commands will be executed serially.

Alternatively, if you have many source files to be analyzed and want to speed up the analysis, you can use the `command` command which means that you collect all your commands in a specific file. In this case, `icstat` will perform the analysis in parallel instead. The command line would then look like this:

```
icstat --db a.db --checks checks.ch command commands.txt
```

`commands.txt` contains:

```
analyze -- iccxxxxx compiler_opts cstat1.c
analyze -- iccxxxxx compiler_opts cstat2.c
link_analyze -- ilinkxxxxx linker_opts cstat1.o cstat2.o
```

See the note above regarding `ilink`*xxxxx* and the filename extensions.

4 After running `icstat` on the `cstat1.c` file, these messages are listed ion the console an stored in the database (assuming all default checks are performed):

```
"cstat1.c",15 Severity-High[PTR-null-fun-pos]: Function call
`f1()' is immediately dereferenced, without checking for NULL.
CERT-EXP34-C,CWE-476
     15: ! - possible_null
     15: > - Entering into f1
      7: ! -   Return NULL

"cstat1.c",18 Severity-Low[RED-unused-assign]: Value assigned to
variable `ch' is never used.  CERT-MSC13-C,CWE-563
```

Note that the first message is followed by *trace information*, which describes the required execution path to trigger the deviation from the rule, including information about assumptions made on conditional statements.

5 This message is listed for the `cstat2.c` file:

```
"cstat2.c",16 Severity-High[ARR-inv-index]: Array `arr' 1st
subscript 20 is out of bounds [0,9].
CERT-ARR33-C,CWE-119,CWE-120,CWE-121,CWE-124,CWE-126,CWE-127,CWE-
129,MISRAC++2008-5-0-16,MISRAC2012-Rule-18.1
```

6 Edit the source files to remove the problem and repeat the analysis.

Note: C-STAT has a built-in preprocessor symbol, `__CSTAT__`, that you can use to explicitly include or exclude specific source code from the analysis.

Reference information on icstat and ichecks

Reference information about:

See the compiler documentation for information about generic syntax rules for options, exit statuses, etc.

INVOCATION SYNTAX FOR ICSTAT

The invocation syntax for `icstat`:

```
icstat parameters [-- command_line]
```

Parameters

The different parts are:

Syntax parts	Description
commands	Commands that define an operation to be performed, see *Summary of icstat commands*, page 11.
options	Command line options that define actions to be performed, see *Summary of icstat options*, page 11. These options can be placed anywhere on the command line, but must come before `--`.
command_line	Compiler or linker command line for the `analyze` and `link_analyze` commands.

Table 1: icstat syntax

For an example, see *Using icstat*, page 8.

SUMMARY OF ICSTAT COMMANDS

This table summarizes the `icstat` commands:

Icstat commands	Description
`analyze`	Analyzes a source file. The command line must end with a compiler invocation (`--`).
`link_analyze`	Analyzes an application. The command line must end with a linker invocation (`--`).
`load`	Outputs the analyze messages from the database file.
`clear`	Clears the database file.
`commands` *`cmd`*	Executes the commands in the *`cmd`* file.

Table 2: icstat commands summary

For an example, see *Using icstat*, page 8.

When running `icstat` with the commands `analyze` or `link_analyze`, identified deviations will be listed on `stdout` on the format:

`Severity[`*`check-tag`*`]: `*`message. Alias tags`*`.`

SUMMARY OF ICSTAT OPTIONS

This table summarizes the `icstat` options:

Command line option	Description
`--checks`	Specifies the manifest file, which contains the checks to run.
`--db`	Contains analyze information (mandatory).
`-f`	Extends the command line.

Table 3: icstat options summary

For more information, see *Descriptions of C-STAT options*, page 12.

INVOCATION SYNTAX FOR ICHECKS

The invocation syntax for `ichecks`:

`ichecks `*`options`*

The default name of the output file is `cstat_sel_checks.txt`.

For an example, see *Using icstat*, page 8.

SUMMARY OF ICHECKS OPTIONS

This table summarizes the `ichecks` options:

Command line option	Description
`--all`	Generates all checks to an output file.
`--check`	Generates a specified check to an output file.
`--default`	Generates all default checks for a specific package to an output file.
`--group`	Generates a selected group of checks to an output file.
`--output`	Specifies an output filename other than the default.
`--package`	Generates all checks for a specific package to an output file.

Table 4: ichecks options summary

For more information, see *Descriptions of C-STAT options*, page 12.

Descriptions of C-STAT options

The following is detailed reference information about each command line option available for `icstat` and `ichecks`.

--all

Syntax

`--all`

For use with

`ichecks`

Description

Causes `ichecks` to generate all checks (including non-default checks) to an output file. When you use the output file with `icstat`, `icstat` will run all checks.

To set related options, choose:

Project>Options>Static Analysis>C-STAT Static Analysis>Select Checks

--check

Syntax

`--check` *`tag`*`[,...]`

Parameters

`tag`	The tag of a specific check that you want to run, for example `ARR-inv-index-pos`. You can specify one or several tags.

For use with
`ichecks`

Description
Causes `icheck` to generate the specified check to an output file. When you use the output file with `icstat`, `icstat` will run the specified check.

To set related options, choose:

Project>Options>Static Analysis>C-STAT Static Analysis>Select Checks

--checks

Syntax
`--checks` *`file`*

Parameters

`file`	The name of the manifest file that contains the checks that `icstat` will run. See the rules for specifying a filename or directory as parameters in the compiler documentation.

For use with
`icstat`

Description
Use this option to specify the file that contains the checks to run. You create the file using `ichecks`, see *Using icstat*, page 8.

This option is not available in the IDE.

--db

Syntax
`--db` *`database`*

Parameters

`database`	The name of the file where the analysis result will be stored as a database.

For use with
`icstat`

Description
Use this option to specify the file where the result of the performed analysis should be stored. The result will be stored as a database.

This option is mandatory.

This option is not available in the IDE.

--default

Syntax

`--default package[,...]`

Parameters

package — The name of package to use. Choose between: `stdchecks`, `miscrac2004`, `misrac2012`, or `miscrac++2008`.

For use with

`ichecks`

Description

Causes `ichecks` to generate all default checks for the specified package to an output file. When you use the output file with `icstat`, `icstat` will run the default checks.

To set related options, choose:

Project>Options>Static Analysis>C-STAT Static Analysis>Select Checks

-f

Syntax

`-f filename`

Parameters

See the compiler documentation for information about the rules for specifying a filename or directory as parameters.

For use with

`icstat`

Description

Use this option to make the tool read command line options from the named file, with the default filename extension `xcl`.

In the command file, you format the items exactly as if they were on the command line itself, except that you can use multiple lines, because the newline character is treated as a space or tab character.

Both C and C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.

This option is not available in the IDE.

--group

Syntax

`--group` *`group`*`[,...]`

Parameters

`group`	The group of checks that you want to run, for example `ARR` for array bounds or `ATH` for arithmetic errors. For information about available groups, see the **Options** dialog box in the IAR Embedded Workbench IDE. You can specify one or several groups.

For use with

`ichecks`

Description

Causes `ichecks` to generate the specified group of checks to an output file. When you use the output file with `icstat`, `icstat` will run the specified group of checks.

To set related options, choose:

Project>Options>Static Analysis>C-STAT Static Analysis>Select Checks

--package

Syntax

`--package` *`package`*`[,...]`

Parameters

`package`	The package of checks that you want to run. Choose between: `stdchecks`, `miscrac2004`, `misrac2012`, or `miscrac++2008`. You can specify one or several packages.

For use with

`ichecks`

Description

Causes `ichecks` to generate the specified package of checks to an output file. When you use the output file with `icstat`, `icstat` will run the specified package of checks.

To set related options, choose:

Project>Options>Static Analysis>C-STAT Static Analysis>Select Checks

--output

Syntax `--output {file|-}`

Parameters

file	The name of the output file.
-	Directs the output to `stdout`.

For use with `ichecks`

Description By default, the generated output produced by `ichecks` is located in a file with the name `cstat_sel_checks.txt`. Use this option to explicitly specify a different output filename.

This option is not available in the IDE.

C-STAT checks

- Summary of checks
- Descriptions of checks

Summary of checks

This table summarizes the C-STAT checks

Check	Synopsis
`ARR-inv-index-pos`	Array access might be out of bounds, depending on which path is executed.
`ARR-inv-index-ptr-pos`	A pointer to an array is potentially used outside the array bounds
`ARR-inv-index-ptr`	A pointer to an array is used outside the array bounds
`ARR-inv-index`	Array access is out of bounds.
`ARR-neg-index`	An array is accessed with a negative subscript value.
`ARR-uninit-index`	An array is indexed with an uninitialized variable
`ATH-cmp-float`	Floating point comparisons using == or !=
`ATH-cmp-unsign-neg`	An unsigned value is compared to see whether it is negative.
`ATH-cmp-unsign-pos`	An unsigned value is compared to see whether it is greater than or equal to `0`.
`ATH-div-0-assign`	A variable is assigned the value `0`, then used as a divisor.
`ATH-div-0-cmp-aft`	After a successful comparison with `0`, a variable is used as a divisor.
`ATH-div-0-cmp-bef`	A variable used as a divisor is afterwards compared with `0`.
`ATH-div-0-interval`	Interval analysis has found a value that is `0` and used as a divisor.
`ATH-div-0-pos`	Interval analysis has found an expression that might be `0` and is used as a divisor.

Table 5: Summary of checks

Check	Synopsis
ATH-div-0-unchk-global	A global variable is used as a divisor without having been determined to be non-zero.
ATH-div-0-unchk-local	A local variable is used as a divisor without having been determined to be non-zero.
ATH-div-0-unchk-param	A parameter is used as a divisor without having been determined to be non-zero.
ATH-div-0	An expression that results in 0 is used as a divisor.
ATH-inc-bool (C++ only)	Deprecated operation on bool.
ATH-malloc-overrun	The size of memory passed to malloc to allocate overflows.
ATH-neg-check-nonneg	A variable is checked for a non-negative value after being used, instead of before.
ATH-neg-check-pos	A variable is checked for a positive value after being used, instead of before.
ATH-new-overrun (C++ only)	An arithmetic overflow is caused by an allocation using new[].
ATH-overflow-cast	An expression is cast to a different type, resulting in an overflow or underflow of its value.
ATH-overflow	An expression is implicitly converted to a narrower type, resulting in an overflow or underflow of its value.
ATH-shift-bounds	Out of range shifts
ATH-shift-neg	The left-hand side of a right shift operation might be a negative value.
ATH-sizeof-by-sizeof	Multiplying sizeof by sizeof.
CAST-old-style (C++ only)	Old style casts (other than void casts) are used
CATCH-object-slicing (C++ only)	Exception objects are caught by value
CATCH-xtor-bad-member (C++ only)	Exception handler in constructor or destructor accesses non-static member variable that might not exist.
COMMA-overload (C++ only)	Overloaded comma operator
COMMENT-nested	Appearances of /* inside comments
CONST-local	A local variable that is not modified after initialization is not declared const.

Table 5: Summary of checks

Check	Synopsis
`CONST-member-ret (C++ only)`	A member function qualified as `const` returns a pointer member variable.
`CONST-param`	A function does not modify one of its parameters.
`COP-alloc-ctor (C++ only)`	A class member is deallocated in the class' destructor, but not allocated in a constructor or assignment operator.
`COP-assign-op-ret (C++ only)`	An assignment operator of a C++ class does not return a non-`const` reference to `this`.
`COP-assign-op-self (C++ only)`	Assignment operator does not check for self-assignment before allocating member functions
`COP-assign-op (C++ only)`	There is no assignment operator defined for a class whose destructor deallocates memory.
`COP-copy-ctor (C++ only)`	A class which uses dynamic memory allocation does not have a user-defined copy constructor.
`COP-dealloc-dtor (C++ only)`	A class member has memory allocated in a constructor or an assignment operator, that is not released in the destructor.
`COP-dtor-throw (C++ only)`	An exception is thrown, or might be thrown, in a class destructor.
`COP-dtor (C++ only)`	A class which dynamically allocates memory in its copy control functions does not have a destructor.
`COP-init-order (C++ only)`	Data members are initialized with other data members that are in the same initialization list.
`COP-init-uninit (C++ only)`	An initializer list reads the values of still uninitialized members.
`COP-member-uninit (C++ only)`	A member of a class is not initialized in one of the class constructors.
`CPU-ctor-call-virt (C++ only)`	A virtual member function is called in a class constructor.
`CPU-ctor-implicit (C++ only)`	Constructors that are callable with a single argument of fundamental type are not declared `explicit`.
`CPU-delete-throw (C++ only)`	An exception is thrown, or might be thrown, in an overloaded `delete` or `delete[]` operator.
`CPU-delete-void (C++ only)`	A pointer to void is used in `delete`, causing the destructor not to be called.

Table 5: Summary of checks

Check	Synopsis
CPU-dtor-call-virt (C++ only)	A virtual member function is called in a class destructor.
CPU-malloc-class (C++ only)	An allocation of a class instance with malloc() does not call a constructor.
CPU-nonvirt-dtor (C++ only)	A public non-virtual destructor is defined in a class with virtual methods.
CPU-return-ref-to-class-data (C++ only)	Member functions return non-const handles to members.
DECL-implicit-int	An object or function of the type int is declared or defined, but its type is not explicitly stated.
DEFINE-hash-multiple	Multiple # or ## operators in a macro definition.
ENUM-bounds	Conversions to enum that are out of range of the enumeration.
EXP-cond-assign	An assignment might be mistakenly used as the condition for an if, for, while, or do statement.
EXP-dangling-else	An else branch might be connected to an unexpected if statement.
EXP-loop-exit	An unconditional break, continue, return, or goto within a loop.
EXP-main-ret-int	The return type of main() is not int.
EXP-null-stmt	The body of an if, while, or for statement is a null statement.
EXP-stray-semicolon	Stray semicolons on the same line as other code
EXPR-const-overflow	A constant unsigned integer expression overflows.
FPT-cmp-null	The address of a function is compared with NULL.
FPT-literal	A function pointer that refers to a literal address is dereferenced.
FPT-misuse	A function pointer is used in an invalid context.
FUNC-implicit-decl	Functions are used without prototyping.
FUNC-unprototyped-all	Functions are declared with an empty () parameter list that does not form a valid prototype.
FUNC-unprototyped-used	Arguments are passed to functions without a valid prototype.
INCLUDE-c-file	A .c file includes one or more .c files.

Table 5: Summary of checks

Check	Synopsis
INT-use-signed-as-unsigned-pos	A negative signed integer is implicitly cast to an unsigned integer.
INT-use-signed-as-unsigned	A negative signed integer is implicitly cast to an unsigned integer.
ITR-end-cmp-aft (C++ only)	An iterator is used, then compared with end()
ITR-end-cmp-bef (C++ only)	An iterator is compared with end() or rend(), then dereferenced.
ITR-invalidated (C++ only)	An iterator assigned to point into a container is used or dereferenced even though it might be invalidated.
ITR-mismatch-alg (C++ only)	A pair of iterators passed to an STL algorithm function point to different containers.
ITR-store (C++ only)	A container's begin() or end() iterator is stored and subsequently used.
ITR-uninit (C++ only)	An iterator is dereferenced or incremented before it is assigned to point into a container.
LIB-bsearch-overrun-pos	Arguments passed to bsearch might cause it to overrun.
LIB-bsearch-overrun	Arguments passed to bsearch cause it to overrun.
LIB-buf-size	A call to a string function has a size argument larger than the size of the target buffer.
LIB-fn-unsafe	A potentially unsafe library function is used.
LIB-fread-overrun-pos	A call to fread might cause a buffer overrun.
LIB-fread-overrun	A call to fread causes a buffer overrun.
LIB-memchr-overrun-pos	A call to memchr might cause a buffer overrun.
LIB-memchr-overrun	A call to memchr causes a buffer overrun.
LIB-memcpy-overrun-pos	A call to memcpy might cause the memory to overrun.
LIB-memcpy-overrun	A call to memcpy or memmove causes the memory to overrun.
LIB-memset-overrun-pos	A call to memset might cause a buffer overrun.
LIB-memset-overrun	A call to memset causes a buffer overrun.
LIB-putenv	putenv used to set environment variable values.

Table 5: Summary of checks

Check	Synopsis
`LIB-qsort-overrun-pos`	Arguments passed to `qsort` might cause it to overrun.
`LIB-qsort-overrun`	Arguments passed to `qsort` cause it to overrun.
`LIB-return-const`	The return value of a `const` standard library function is not used.
`LIB-return-error`	The return value for a library function that might return an error value is not used.
`LIB-return-leak`	The return values from one or more library functions were not stored, returned, or passed as a parameter.
`LIB-return-neg`	A variable assigned using a library function that can return -1 as an error value is subsequently used where the value must be non-negative.
`LIB-return-null`	A pointer is assigned using a library function that can return `NULL` as an error value. This pointer is subsequently dereferenced without checking its value.
`LIB-sprintf-overrun`	A call to `sprintf` causes a destination buffer overrun.
`LIB-std-sort-overrun-pos (C++ only)`	Using `std::sort` might cause buffer overrun.
`LIB-std-sort-overrun (C++ only)`	A buffer overrun is caused by use of `std::sort`.
`LIB-strcat-overrun-pos`	A call to `strcat` might cause destination buffer overrun.
`LIB-strcat-overrun`	A call to `strcat` causes a destination buffer overrun.
`LIB-strcpy-overrun-pos`	A call to `strcpy` might cause destination buffer overrun.
`LIB-strcpy-overrun`	A call to `strcpy` causes a destination buffer overrun.
`LIB-strncat-overrun-pos`	A call to `strncat` might cause a destination buffer overrun.
`LIB-strncat-overrun`	A call to `strncat` causes a destination buffer overrun.
`LIB-strncmp-overrun-pos`	A call to `strncmp` might cause a buffer overrun.

Table 5: Summary of checks

Check	Synopsis
LIB-strncmp-overrun	A buffer overrun is caused by a call to strncmp.
LIB-strncpy-overrun-pos	A call to strncpy might cause a destination buffer overrun.
LIB-strncpy-overrun	A call to strncpy causes a destination buffer overrun.
LOGIC-overload (C++ only)	Overloaded && and || operators
MEM-delete-array-op (C++ only)	A memory location allocated with new is deleted with delete[]
MEM-delete-op (C++ only)	A memory location allocated with new [] is deleted with delete or free.
MEM-double-free-alias	Freeing a memory location more than once.
MEM-double-free-some	A memory location is freed more than once on some paths but not on others.
MEM-double-free	Freeing a memory location more than once.
MEM-free-field	A struct or a class field is possibly freed.
MEM-free-fptr	A function pointer is deallocated.
MEM-free-no-alloc-struct	A struct field is deallocated without first having been allocated.
MEM-free-no-alloc	A pointer is freed without having been allocated.
MEM-free-no-use	Memory is allocated and then freed without being used.
MEM-free-op	Memory allocated with malloc deallocated using delete.
MEM-free-struct-field	A struct's field is deallocated, but is not dynamically allocated.
MEM-free-variable-alias	A stack address might be freed.
MEM-free-variable	A stack address might be freed.
MEM-leak-alias	Incorrect deallocation causes memory leak.
MEM-leak	Incorrect deallocation causes memory leak.
MEM-malloc-arith	An assignment contains both a malloc() and pointer arithmetic on the right-hand side.
MEM-malloc-diff-type	A call to malloc tries to allocate memory based on a sizeof operator, but the destination type of the call is of a different type.

Table 5: Summary of checks

Check	Synopsis
`MEM-malloc-sizeof-ptr`	`malloc(sizeof(p))`, where `p` is a pointer type, is assigned to a non-pointer variable.
`MEM-malloc-sizeof`	Allocating memory with `malloc` without using `sizeof`.
`MEM-malloc-strlen`	Dangerous arithmetic with `strlen` in argument to `malloc`.
`MEM-realloc-diff-type`	The variable that stores the result of `realloc` does not match the type of the first argument.
`MEM-return-free`	A function deallocates memory, then returns a pointer to that memory.
`MEM-return-no-assign`	A function that allocates memory's return value is not stored.
`MEM-stack-alias`	Might return address on the stack.
`MEM-stack-global-alias`	A stack address is stored in a global pointer.
`MEM-stack-global-field`	A stack address is stored in the field of a global struct.
`MEM-stack-global`	A stack address is stored in a global pointer.
`MEM-stack-param-ref (C++ only)`	Stack address is stored via reference parameter.
`MEM-stack-param`	Stack address is stored outside function via parameter.
`MEM-stack-pos`	Might return address on the stack.
`MEM-stack-ref (C++ only)`	A stack object is returned from a function as a reference.
`MEM-stack`	Might return address on the stack.
`MEM-use-free-all`	A pointer is used after it has been freed.
`MEM-use-free-some`	A pointer is used after it has been freed.
`PTR-arith-field`	Direct access to a field of a struct, using an offset from the address of the struct.
`PTR-arith-stack`	Pointer arithmetic applied to a pointer that references a stack address
`PTR-arith-var`	Invalid pointer arithmetic with an automatic variable that is neither an array nor a pointer.
`PTR-cmp-str-lit`	A variable is tested for equality with a string literal.
`PTR-null-assign-fun-pos`	Possible `NULL` pointer dereferenced by a function.

Table 5: Summary of checks

Check	Synopsis
`PTR-null-assign-pos`	A pointer is assigned a value that might be `NULL`, and then dereferenced.
`PTR-null-assign`	A pointer is assigned the value `NULL`, then dereferenced.
`PTR-null-cmp-aft`	A pointer is dereferenced, then compared with `NULL`.
`PTR-null-cmp-bef-fun`	A pointer is compared with `NULL`, then dereferenced by a function.
`PTR-null-cmp-bef`	A pointer is compared with `NULL`, then dereferenced.
`PTR-null-fun-pos`	A possible `NULL` pointer is returned from a function, and immediately dereferenced without checking.
`PTR-null-literal-pos`	A literal pointer expression (like `NULL`) is dereferenced by a function call.
`PTR-overload (C++ only)`	An `&` operator is overloaded.
`PTR-singleton-arith-pos`	Pointer arithmetic might be performed on a pointer that points to a single object.
`PTR-singleton-arith`	Pointer arithmetic is performed on a pointer that points to a single object.
`PTR-unchk-param-some`	A pointer is dereferenced after being determined not to be `NULL` on some paths, but not checked on others.
`PTR-unchk-param`	A pointer parameter is not compared to `NULL`
`PTR-uninit-pos`	Possible dereference of an uninitialized or `NULL` pointer.
`PTR-uninit`	Dereference of an uninitialized or `NULL` pointer.
`RED-case-reach`	A case statement within a switch statement cannot be reached.
`RED-cmp-always`	A comparison using ==, <, <=, >, or >= is always true.
`RED-cmp-never`	A comparison using ==, <, <=, >, or >= is always false.
`RED-cond-always`	The condition in an if, for, while, do-while, or ternary operator will always be true.
`RED-cond-const-assign`	A constant assignment in a conditional expression.

Table 5: Summary of checks

Check	Synopsis
RED-cond-const-expr	A conditional expression with a constant value
RED-cond-const	A constant value is used as the condition for a loop or if statement.
RED-cond-never	The condition in if, for, while, do-while, or ternary operator will never be true.
RED-dead	A part of the application is never executed.
RED-expr	Some expressions, such as x & x and x | x, are redundant.
RED-func-no-effect	A function is declared that has no return type and creates no side effects.
RED-local-hides-global	The definition of a local variable hides a global definition.
RED-local-hides-local	The definition of a local variable hides a previous local definition.
RED-local-hides-member (C++ only)	The definition of a local variable hides a member of the class.
RED-local-hides-param	A variable declaration hides a parameter of the function
RED-no-effect	A statement potentially contains no side effects.
RED-self-assign	In a C++ class member function, a variable is assigned to itself.
RED-unused-assign	A variable is assigned a non-trivial value that is never used.
RED-unused-param	A function parameter is declared but not used.
RED-unused-return-val	There are unused function return values (other than overloaded operators).
RED-unused-val	A variable is assigned a value that is never used.
RED-unused-var-all	A variable is neither read nor written for any execution path.
RESOURCE-deref-file	A pointer to a FILE object is dereferenced.
RESOURCE-double-close	A file resource is closed multiple times
RESOURCE-file-no-close-all	A file pointer is never closed.
RESOURCE-file-pos-neg	A file handler might be negative
RESOURCE-file-use-after-close	A file resource is used after it has been closed.

Table 5: Summary of checks

Check	Synopsis
RESOURCE-implicit-deref-file	A file pointer is implicitly dereferenced by a library function.
RESOURCE-write-ronly-file	A file opened as read-only is written to.
SIZEOF-side-effect	sizeof expressions containing side effects
SPC-init-list	The initalization list of an array contains side effects.
SPC-order	Expressions which depend on order of evaluation
SPC-uninit-arr-all	Reads from local buffers are not preceded by writes.
SPC-uninit-struct-field-heap	A field of a dynamically allocated struct is read before it is initialized.
SPC-uninit-struct-field	A field of a local struct is read before it is initialized.
SPC-uninit-struct	A struct has one or more fields read before they are initialized.
SPC-uninit-var-all	A variable is read before it is assigned a value.
SPC-uninit-var-some	A variable is read before it is assigned a value.
SPC-volatile-reads	There are multiple read accesses with volatile-qualified type within one and the same sequence point.
SPC-volatile-writes	There are multiple write accesses with volatile-qualified type within one and the same sequence point.
STR-trigraph	Trigraphs are used (in string literals only).
STRUCT-signed-bit	There are signed single-bit fields (excluding anonymous fields).
SWITCH-fall-through	There are non-empty switch cases not terminated by break and without 'fallthrough' comment.
THROW-empty (C++ only)	Unsafe rethrow of exception.
THROW-main (C++ only)	No default exception handler for try.
THROW-null	Throw of NULL integer constant
THROW-ptr	Throw of exceptions by pointer
THROW-static (C++ only)	Exceptions thrown without a handler in some call paths that lead to that point.
THROW-unhandled (C++ only)	There are calls to functions explicitly declared to throw an exception type that is not handled (or declared as thrown) by the caller.

Table 5: Summary of checks

Check	Synopsis
UNION-overlap-assign	Assignments from one field of a union to another.
UNION-type-punning	Writing to a field of a union after reading from a different field, effectively re-interpreting the bit pattern with a different type.
MISRAC2004-1.2_a	Checks reads from local buffers are preceded by writes.
MISRAC2004-1.2_b	In all executions, a struct has one or more fields read before they are initialized.
MISRAC2004-1.2_c	An expression resulting in 0 is used as a divisor.
MISRAC2004-1.2_d	A variable is assigned the value 0, then used as a divisor.
MISRAC2004-1.2_e	After a successful comparison with 0, a variable is used as a divisor.
MISRAC2004-1.2_f	A variable used as a divisor is subsequently compared with 0.
MISRAC2004-1.2_g	Interval analysis determines a value is 0, then it is used as a divisor.
MISRAC2004-1.2_h	An expression that may be 0 is used as a divisor.
MISRAC2004-1.2_i	A global variable is not checked against 0 before it is used as a divisor.
MISRAC2004-1.2_j	A local variable is not checked against 0 before it is used as a divisor.
MISRAC2004-2.1	Inline asm statements that are not encapsulated in functions
MISRAC2004-2.2	Uses of // comments
MISRAC2004-2.3	Appearances of /* inside comments
MISRAC2004-2.4	To allow comments to contain pseudo-code or code samples, only comments that end in ';', '{', or '}' characters are considered to be commented-out code.
MISRAC2004-4.2	Uses of trigraphs (in string literals only)
MISRAC2004-5.2_a	The definition of a local variable hides a global definition.
MISRAC2004-5.2_b	The definition of a local variable hides a previous local definition.

Table 5: Summary of checks

Check	Synopsis
MISRAC2004-5.2_c	A variable declaration hides a parameter of the function
MISRAC2004-5.3	Typedef with this name already declared.
MISRAC2004-5.4	A class, struct, union or enum declaration that clashes with a previous declaration.
MISRAC2004-5.5	A identifier is used that can clash with another static identifier.
MISRAC2004-6.1	Arithmetic on objects of type plain char, without an explicit signed or unsigned qualifier
MISRAC2004-6.3	Uses of basic types char, int, short, long, double, and float without typedef
MISRAC2004-6.4	Bitfields with plain int type
MISRAC2004-6.5	Signed single-bit fields (excluding anonymous fields)
MISRAC2004-7.1	Uses of octal integer constants
MISRAC2004-8.1	Functions used without prototyping
MISRAC2004-8.2	Whenever an object or function is declared or defined, its type shall be explicitly stated.
MISRAC2004-8.5_a	A header file shall not contain global variable.
MISRAC2004-8.5_b	Non-inline functions defined in header files
MISRAC2004-8.12	External arrays declared without size stated explicitly or defined implicitly by initialization.
MISRAC2004-9.1_a	In all executions, a variable is read before it is assigned a value.
MISRAC2004-9.1_b	In some execution, a variable is read before it is assigned a value.
MISRAC2004-9.1_c	Dereference of an uninitialized or NULL pointer.
MISRAC2004-9.2	This check points out where a non-zero array initialisation does not exactly match the structure of the array declaration.
MISRAC2004-10.1_a	An expression of integer type is implicitly converted to a narrower or different sign underlying type
MISRAC2004-10.1_b	A complex expression of integer type is implicitly converted to a different underlying type.

Table 5: Summary of checks

Check	Synopsis
MISRAC2004-10.1_c	A non-constant expression of integer type is implicitly converted to a different underlying type in a function argument.
MISRAC2004-10.1_d	A non-constant expression of integer type is implicitly converted to a different underlying type in a return expression.
MISRAC2004-10.2_a	An expression of floating type is implicitly converted to a narrower underlying type
MISRAC2004-10.2_b	An expression of floating type is implicitly converted to a narrower underlying type
MISRAC2004-10.2_c	A non-constant expression of floating type is implicitly converted to a different underlying type in a function argument.
MISRAC2004-10.2_d	A non-constant expression of floating type is implicitly converted to a different underlying type in a return expression.
MISRAC2004-10.3	A complex expression of integer type is cast to a wider or different sign underlying type.
MISRAC2004-10.4	A complex expression of floating type is cast to a wider or different underlying type.
MISRAC2004-10.5	Bitwise operation on unsigned char or unsigned short, that are not immediately cast to this type to ensure consistent truncation
MISRAC2004-10.6	A U suffix shall be applied to all constants of unsigned type.
MISRAC2004-11.1	Conversions shall not be performed between a pointer to a function and any type other than an integral type.
MISRAC2004-11.3	A cast should not be performed between a pointer type and an integral type.
MISRAC2004-11.4	A pointer to object type is cast to a pointer to different object type
MISRAC2004-11.5	Casts that remove any const or volatile qualification.
MISRAC2004-12.1	Add parentheses to avoid implicit operator precedence.

Table 5: Summary of checks

Check	Synopsis
MISRAC2004-12.2_a	Expressions which depend on order of evaluation
MISRAC2004-12.2_b	There shall be no more than one read access with volatile-qualified type within one sequence point
MISRAC2004-12.2_c	There shall be no more than one modification access with volatile-qualified type within one sequence point
MISRAC2004-12.3	Sizeof expressions containing side effects
MISRAC2004-12.4	Right hand operands of && or || that contain side effects
MISRAC2004-12.6_a	Operands of logical operators (&&, ||, and !) that are not effectively Boolean.
MISRAC2004-12.6_b	Uses of arithmetic operators on boolean operands.
MISRAC2004-12.7	Applications of bitwise operators to signed operands
MISRAC2004-12.8	Out of range shifts
MISRAC2004-12.9	Uses of unary - on unsigned expressions
MISRAC2004-12.10	Uses of the comma operator
MISRAC2004-12.11	A constant unsigned integer expression overflows
MISRAC2004-12.12_a	Reading from a field of a union following a write to a different field, effectively re-interpreting the bit pattern with a different type.
MISRAC2004-12.12_b	An expression provides access to the bit-representation of a floating point variable.
MISRAC2004-12.13	Uses of increment (++) and decrement (--) operators mixed with other operators in an expression.
MISRAC2004-13.1	Assignment operators shall not be used in expressions that yield a boolean value.
MISRAC2004-13.2_a	Non-boolean termination conditions in do ... while statements.
MISRAC2004-13.2_b	Non-boolean termination conditions in for loops.
MISRAC2004-13.2_c	Non-boolean conditions in if statements.
MISRAC2004-13.2_d	Non-boolean termination conditions in while statements.

Table 5: Summary of checks

Check	Synopsis
MISRAC2004-13.2_e	Non-boolean operands to the conditional (? :) operator
MISRAC2004-13.3	Floating point comparisons using == or !=
MISRAC2004-13.4	Floating-point values in the controlling expression of a for statement.
MISRAC2004-13.5	A `for` loop counter variable is not initialized in the `for` loop.
MISRAC2004-13.6	A `for` loop counter variable is modified in the body of the loop.
MISRAC2004-13.7_a	A comparison using ==, <, <=, >, or >= is always true.
MISRAC2004-13.7_b	A comparison using ==, <, <=, >, or >= is always false.
MISRAC2004-14.1	In all executions, a part of the program is not executed.
MISRAC2004-14.2	A statement that potentially contains no side effects.
MISRAC2004-14.3	Stray semicolons on the same line as other code
MISRAC2004-14.4	Uses of goto.
MISRAC2004-14.5	Uses of continue.
MISRAC2004-14.6	Multiple break points from loop.
MISRAC2004-14.7	A function shall have a single point of exit at the end of the function.
MISRAC2004-14.8_a	Missing braces in do ... while statements
MISRAC2004-14.8_b	Missing braces in for statements
MISRAC2004-14.8_c	Missing braces in switch statements
MISRAC2004-14.8_d	Missing braces in while statements
MISRAC2004-14.9	Missing braces in if, else, and else if statements
MISRAC2004-14.10	If ... else if constructs that are not terminated with an else clause.
MISRAC2004-15.0	Switch statements that do not conform to the MISRA C switch syntax.
MISRAC2004-15.1	Switch labels in nested blocks.
MISRAC2004-15.2	Non-empty switch cases not terminated by break

Table 5: Summary of checks

Check	Synopsis
MISRAC2004-15.3	Switch statements with no default clause, or a default clause that is not the final clause.
MISRAC2004-15.4	A switch expression shall not represent a value that is effectively boolean.
MISRAC2004-15.5	Switch statements with no cases.
MISRAC2004-16.1	Functions defined using ellipsis (...) notation
MISRAC2004-16.2_a	Functions that call themselves directly.
MISRAC2004-16.2_b	Functions that call themselves indirectly.
MISRAC2004-16.3	Function prototypes must name all parameters
MISRAC2004-16.5	Functions declared with an empty () parameter list that does not form a valid prototype
MISRAC2004-16.7	A function does not modify one of its parameters.
MISRAC2004-16.8	For some execution, no return statement is executed in a function with a non-void return type
MISRAC2004-16.9	Function addresses taken without explicit &
MISRAC2004-16.10	The return value for a library function that may return an error value is not used.
MISRAC2004-17.1_a	Direct access to a field of a struct using an offset from the address of the struct.
MISRAC2004-17.1_b	Pointer arithmetic applied to a pointer that references a stack address
MISRAC2004-17.1_c	Invalid pointer arithmetic with an automatic variable that is neither an array nor a pointer.
MISRAC2004-17.4_a	Array indexing shall be the only allowed form of pointer arithmetic.
MISRAC2004-17.4_b	Array indexing shall only be applied to objects defined as an array type.
MISRAC2004-17.5	The declaration of objects should contain no more than two levels of pointer indirection.
MISRAC2004-17.6_a	May return address on the stack.
MISRAC2004-17.6_b	Store a stack address in a global pointer.
MISRAC2004-17.6_c	Store a stack address in the field of a global struct.
MISRAC2004-17.6_d	Store stack address outside function via parameter.

Table 5: Summary of checks

Check	Synopsis
MISRAC2004-18.1	Structs and unions that are used without being defined.
MISRAC2004-18.2	Assignments from one field of a union to another.
MISRAC2004-18.4	All unions
MISRAC2004-19.2	Illegal characters in header file names
MISRAC2004-19.6	All #undef's
MISRAC2004-19.7	Function-like macros
MISRAC2004-19.12	Multiple # or ## operators in a macro definition
MISRAC2004-19.13	The # and ## operators should not be used
MISRAC2004-19.15	Header files without #include guards
MISRAC2004-20.1	#define or #undef of a reserved identifier in the standard library
MISRAC2004-20.4	All uses of malloc, calloc, realloc, and free
MISRAC2004-20.5	All uses of errno
MISRAC2004-20.6	All uses of the offsetof built-in function
MISRAC2004-20.7	All uses of <setjmp.h>
MISRAC2004-20.8	All uses of <signal.h>
MISRAC2004-20.9	All uses of <stdio.h>
MISRAC2004-20.10	All uses of atof, atoi, atol and atoll
MISRAC2004-20.11	All uses of abort, exit, getenv, and system
MISRAC2004-20.12	All uses of <time.h> functions: asctime, clock, ctime, difftime, gmtime, localtime, mktime, strftime, and time
MISRAC2012-Dir-4.3	Inline asm statements that are not encapsulated in functions
MISRAC2012-Dir-4.4	To allow comments to contain pseudo-code or code samples, only comments that end in ';', '{', or '}' characters are considered to be commented-out code.
MISRAC2012-Dir-4.6_a	Uses of basic types char, int, short, long, double, and float without typedef
MISRAC2012-Dir-4.9	Function-like macros
MISRAC2012-Dir-4.10	Header files without #include guards
MISRAC2012-Rule-1.3_a	An expression resulting in `0` is used as a divisor.

Table 5: Summary of checks

Check	Synopsis
MISRAC2012-Rule-1.3_b	A variable is assigned the value 0, then used as a divisor.
MISRAC2012-Rule-1.3_c	After a successful comparison with 0, a variable is used as a divisor.
MISRAC2012-Rule-1.3_d	A variable used as a divisor is subsequently compared with 0.
MISRAC2012-Rule-1.3_e	Interval analysis determines a value is 0, then it is used as a divisor.
MISRAC2012-Rule-1.3_f	An expression that may be 0 is used as a divisor.
MISRAC2012-Rule-1.3_g	A global variable is not checked against 0 before it is used as a divisor.
MISRAC2012-Rule-1.3_h	A local variable is not checked against 0 before it is used as a divisor.
MISRAC2012-Rule-2.1_a	A case statement within a switch statement is unreachable.
MISRAC2012-Rule-2.1_b	In all executions, a part of the program is not executed.
MISRAC2012-Rule-2.2_a	A statement that potentially contains no side effects.
MISRAC2012-Rule-2.2_c	A variable is assigned a value that is never used.
MISRAC2012-Rule-2.7	A function parameter is declared but not used.
MISRAC2012-Rule-3.1	The character sequences /* and // shall not be used within a comment
MISRAC2012-Rule-4.2	Uses of trigraphs (in string literals only)
MISRAC2012-Rule-5.1	An external identifier is not unique for the first 31 characters but not identical
MISRAC2012-Rule-5.3_a	The definition of a local variable hides a global definition.
MISRAC2012-Rule-5.3_b	The definition of a local variable hides a previous local definition.
MISRAC2012-Rule-5.3_c	A variable declaration hides a parameter of the function
MISRAC2012-Rule-5.4_c89	Macro names that are not distinct in their first 31 characters from their macro parameters or other macro names

Table 5: Summary of checks

Check	Synopsis
MISRAC2012-Rule-5.4_c99	Macro names that are not distinct in their first 63 characters from their macro parameters or other macro names
MISRAC2012-Rule-5.5_c89	Non-macro identifiers that are not distinct in their first 31 characters from macro names
MISRAC2012-Rule-5.5_c99	Non-macro identifiers that are not distinct in their first 63 characters from macro names
MISRAC2012-Rule-5.6	Typedef with this name already declared.
MISRAC2012-Rule-5.7	A class, struct, union or enum declaration that clashes with a previous declaration.
MISRAC2012-Rule-5.8	External identifier names should be unique
MISRAC2012-Rule-6.1	Bitfields with plain int type
MISRAC2012-Rule-6.2	Signed single-bit fields (excluding anonymous fields)
MISRAC2012-Rule-7.1	Uses of octal integer constants
MISRAC2012-Rule-7.2	A U suffix shall be applied to all constants of unsigned type.
MISRAC2012-Rule-7.3	Lower case character 'l' should not be used as a suffix.
MISRAC2012-Rule-7.4_a	A string literal is assigned to a variable not declared as constant
MISRAC2012-Rule-7.4_b	Part of string literal is modified via array subscript operator []
MISRAC2012-Rule-8.1	Whenever an object or function is declared or defined, its type shall be explicitly stated.
MISRAC2012-Rule-8.2_a	Functions declared with an empty () parameter list that does not form a valid prototype
MISRAC2012-Rule-8.2_b	Function prototypes must name all parameters
MISRAC2012-Rule-8.10	All inline functions should be declared as static
MISRAC2012-Rule-8.11	External arrays declared without size stated explicitly or defined implicitly by initialization.
MISRAC2012-Rule-8.14	The use of the `restrict' type qualifier is forbidden for function parameters
MISRAC2012-Rule-9.1_a	Possibly dereference of an uninitialized or NULL pointer.

Table 5: Summary of checks

Check	Synopsis
MISRAC2012-Rule-9.1_b	Checks reads from local buffers are preceded by writes.
MISRAC2012-Rule-9.1_c	In all executions, a struct has one or more fields read before they are initialized.
MISRAC2012-Rule-9.1_d	A field of a local struct is read before it is initialized.
MISRAC2012-Rule-9.1_e	In all executions, a variable is read before it is assigned a value.
MISRAC2012-Rule-9.1_f	In some execution, a variable is read before it is assigned a value.
MISRAC2012-Rule-9.3	Arrays shall not be partially initialized
MISRAC2012-Rule-9.5_a	Arrays initialized with designated initializers must have a fixed length
MISRAC2012-Rule-9.5_b	Flexible array members cannot be initalized with a designated initalizer
MISRAC2012-Rule-10.1_R2	An expression of essentially Boolean type should always be used where an operand is interpreted as a Boolean value
MISRAC2012-Rule-10.1_R3	An operand of essentially Boolean type should noe be used where an operand is interpreted as a numeric value
MISRAC2012-Rule-10.1_R4	An operand of essentially character type should not be used where an operand is interpreted as a numeric value
MISRAC2012-Rule-10.1_R5	An operand of essentially enum type should not be used in an arithmetic operation because an enum object uses an implementation-defined integer type.
MISRAC2012-Rule-10.1_R6	Shift and bitwise opearation should only be performed on operands of essentially unsigned type.
MISRAC2012-Rule-10.1_R7	The right hand operand of a shift operator should be of essentially unsigned type to ensure that undefined behaviour does not result from a negative shift.
MISRAC2012-Rule-10.1_R8	An operand of essentially unsigned typed should not be used as the operand to the unary minus operator, as the signedness of the result is determined by the implementation size of int

Table 5: Summary of checks

Check	Synopsis
MISRAC2012-Rule-10.2	Expressions of essentially character type shall not be used inapproriately in addition and subtraction operations
MISRAC2012-Rule-10.3	The value of an expression shall not be assigned to an object with a narrower essential type or a different essential type category
MISRAC2012-Rule-10.4	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category
MISRAC2012-Rule-10.6	The value of a composite expression shall not be assigned to an object with wider essential type
MISRAC2012-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
MISRAC2012-Rule-10.8	The value of a composite expression shall not be cast to a different essential type category or a wider essential type
MISRAC2012-Rule-11.1	Conversion shall not be performed between a pointer to a function and any other type
MISRAC2012-Rule-11.3	A pointer to object type is cast to a pointer to different object type
MISRAC2012-Rule-11.4	A cast should not be performed between a pointer type and an integral type.
MISRAC2012-Rule-11.7	A cast shall not be performed between pointer to object and a non-integer arithmetic type
MISRAC2012-Rule-11.8	Casts that remove any const or volatile qualification.
MISRAC2012-Rule-11.9	An integer constant is used where the NULL macro should be
MISRAC2012-Rule-12.1	Add parentheses to avoid implicit operator precedence.
MISRAC2012-Rule-12.2	Out of range shifts
MISRAC2012-Rule-12.3	Uses of the comma operator
MISRAC2012-Rule-12.4	Evaluation of constant expressions should not lead to unsigned integer wrap-around

Table 5: Summary of checks

Check	Synopsis
MISRAC2012-Rule-13.1	The initalisation list of an array should not contain side effects
MISRAC2012-Rule-13.2_a	Expressions which depend on order of evaluation
MISRAC2012-Rule-13.2_b	There shall be no more than one read access with volatile-qualified type within one sequence point
MISRAC2012-Rule-13.2_c	There shall be no more than one modification access with volatile-qualified type within one sequence point
MISRAC2012-Rule-13.3	Uses of increment (++) and decrement (--) operators mixed with other operators in an expression.
MISRAC2012-Rule-13.4_a	An assignment may be mistakenly used as the condition for an `if`, `for`, `while` or `do` statement.
MISRAC2012-Rule-13.4_b	Assignment in a sub-expression.
MISRAC2012-Rule-13.5	Right hand operands of && or || that contain side effects
MISRAC2012-Rule-13.6	The operand of the sizeof operator shall not contain any expression which has potential side effects
MISRAC2012-Rule-14.1_a	Floating-point values in the controlling expression of a for statement.
MISRAC2012-Rule-14.1_b	An essentially float variable, used in the loop condition, is modified in the loop body
MISRAC2012-Rule-14.2	A `for` loop counter variable is modified in the body of the loop.
MISRAC2012-Rule-14.3_a	The condition in if, for, while, do-while and ternary operator will always be met.
MISRAC2012-Rule-14.3_b	The condition in if, for, while, do-while and ternary operator will never be met.
MISRAC2012-Rule-14.4_a	Non-boolean termination conditions in do ... while statements.
MISRAC2012-Rule-14.4_b	Non-boolean termination conditions in for loops.
MISRAC2012-Rule-14.4_c	Non-boolean conditions in if statements.
MISRAC2012-Rule-14.4_d	Non-boolean termination conditions in while statements.
MISRAC2012-Rule-15.1	Uses of goto.

Table 5: Summary of checks

Check	Synopsis
MISRAC2012-Rule-15.2	Goto declared after target label.
MISRAC2012-Rule-15.3	The target of the goto is a nested code block.
MISRAC2012-Rule-15.4	There should be no more than one break or goto statement used to terminate any iteration statement
MISRAC2012-Rule-15.5	A function shall have a single point of exit at the end of the function.
MISRAC2012-Rule-15.6_a	Missing braces in do ... while statements
MISRAC2012-Rule-15.6_b	Missing braces in for statements
MISRAC2012-Rule-15.6_c	Missing braces in if, else, and else if statements
MISRAC2012-Rule-15.6_d	Missing braces in switch statements
MISRAC2012-Rule-15.6_e	Missing braces in while statements
MISRAC2012-Rule-15.7	If ... else if constructs that are not terminated with an else clause.
MISRAC2012-Rule-16.1	Switch statements that do not conform to the MISRA C switch syntax.
MISRAC2012-Rule-16.2	Switch labels in nested blocks.
MISRAC2012-Rule-16.3	Non-empty switch cases not terminated by break
MISRAC2012-Rule-16.4	Switch statements with no default clause.
MISRAC2012-Rule-16.5	A switch's default label should be either the first or last label of the switch
MISRAC2012-Rule-16.6	Switch statements with no cases.
MISRAC2012-Rule-16.7	A switch expression shall not represent a value that is effectively boolean.
MISRAC2012-Rule-17.1	The use of the stdarg header is not permitted
MISRAC2012-Rule-17.2_a	Functions that call themselves directly.
MISRAC2012-Rule-17.2_b	Functions that call themselves indirectly.
MISRAC2012-Rule-17.3	Functions used without prototyping
MISRAC2012-Rule-17.4	For some execution, no return statement is executed in a function with a non-void return type
MISRAC2012-Rule-17.6	Array parameters shall not have the static keyword between the []
MISRAC2012-Rule-17.7	Unused function return values (excluding overloaded operators)

Table 5: Summary of checks

Check	Synopsis
MISRAC2012-Rule-18.1_a	Array access is out of bounds.
MISRAC2012-Rule-18.1_b	Array access may be out of bounds, depending on which path is executed.
MISRAC2012-Rule-18.1_c	A pointer to an array is used outside the array bounds
MISRAC2012-Rule-18.1_d	A pointer to an array is potentially used outside the array bounds
MISRAC2012-Rule-18.5	The declaration of objects should contain no more than two levels of pointer indirection.
MISRAC2012-Rule-18.6_a	May return address on the stack.
MISRAC2012-Rule-18.6_b	Store a stack address in a global pointer.
MISRAC2012-Rule-18.6_c	Store a stack address in the field of a global struct.
MISRAC2012-Rule-18.6_d	Store stack address outside function via parameter.
MISRAC2012-Rule-18.7	Flexible array members shall not be declared
MISRAC2012-Rule-18.8	Arrays shall not be declared with a variable length
MISRAC2012-Rule-19.1	Assignments from one field of a union to another.
MISRAC2012-Rule-19.2	All unions
MISRAC2012-Rule-20.2	Illegal characters in header file names
MISRAC2012-Rule-20.4_c89	A macro shall not be defined with the same name as a keyword.
MISRAC2012-Rule-20.4_c99	A macro shall not be defined with the same name as a keyword.
MISRAC2012-Rule-20.5	All #undef's
MISRAC2012-Rule-20.10	# or ## operator used in a macro definition
MISRAC2012-Rule-21.1	#define or #undef of a reserved identifier in the standard library
MISRAC2012-Rule-21.2	A library function is being overridden.
MISRAC2012-Rule-21.3	All uses of malloc, calloc, realloc, and free
MISRAC2012-Rule-21.4	All uses of <setjmp.h>
MISRAC2012-Rule-21.5	All uses of <signal.h>
MISRAC2012-Rule-21.6	All uses of <stdio.h>
MISRAC2012-Rule-21.7	All uses of atof, atoi, atol and atoll
MISRAC2012-Rule-21.8	All uses of abort, exit, getenv, and system

Table 5: Summary of checks

Check	Synopsis
MISRAC2012-Rule-21.9	(Required) The library functions bsearch and qsort of <stdlib.h> shall not be used.
MISRAC2012-Rule-21.10	All uses of <time.h> functions: asctime, clock, ctime, difftime, gmtime, localtime, mktime, strftime, and time
MISRAC2012-Rule-21.11	The use of the tgmath header is not permitted
MISRAC2012-Rule-22.1_a	A memory leak due to improper deallocation.
MISRAC2012-Rule-22.1_b	All file pointers obtained dynamically by means of Standard Library functions shall be explicitly released
MISRAC2012-Rule-22.2_a	Freeing a memory location more than once.
MISRAC2012-Rule-22.2_b	Freeing a memory location more than once on some paths but not others.
MISRAC2012-Rule-22.2_c	A stack address is possibly freed.
MISRAC2012-Rule-22.4	A file opened as read-only is written to
MISRAC2012-Rule-22.5_a	A pointer to a FILE object shall not be dereferenced
MISRAC2012-Rule-22.5_b	A file pointer is implicitly derefrenced by a library function
MISRAC2012-Rule-22.6	A file pointer is used after it has been closed.
MISRAC++2008-0-1-1	In all executions, a part of the program is not executed.
MISRAC++2008-0-1-2_a	The condition in if, for, while, do-while and ternary operator will always be met.
MISRAC++2008-0-1-2_b	The condition in if, for, while, do-while and ternary operator will never be met.
MISRAC++2008-0-1-2_c	A case statement within a switch statement is unreachable.
MISRAC++2008-0-1-3	A variable is neither read nor written for any execution.
MISRAC++2008-0-1-4	A variable is assigned a value that is never used.
MISRAC++2008-0-1-6	A variable is assigned a value that is never used.
MISRAC++2008-0-1-7	Unused function return values (excluding overloaded operators)
MISRAC++2008-0-1-8	A function with no return type and no side effects effectively does nothing.

Table 5: Summary of checks

Check	Synopsis
MISRAC++2008-0-1-9	In all executions, a part of the program is not executed.
MISRAC++2008-0-1-11	A function parameter is declared but not used.
MISRAC++2008-0-2-1	Assignments from one field of a union to another.
MISRAC++2008-0-3-2	The return value for a library function that may return an error value is not used.
MISRAC++2008-2-3-1	Uses of trigraphs (in string literals only)
MISRAC++2008-2-7-1	Appearances of /* inside comments
MISRAC++2008-2-7-2	To allow comments to contain pseudo-code or code samples, only comments that end in ';', '{', or '}' characters are considered to be commented-out code.
MISRAC++2008-2-7-3	To allow comments to contain pseudo-code or code samples, only comments that end in ';', '{', or '}' characters are considered to be commented-out code.
MISRAC++2008-2-10-2_a	The definition of a local variable hides a global definition.
MISRAC++2008-2-10-2_b	The definition of a local variable hides a previous local definition.
MISRAC++2008-2-10-2_c	A variable declaration hides a parameter of the function
MISRAC++2008-2-10-2_d (C++ only)	The definition of a local variable hides a member of the class.
MISRAC++2008-2-10-3	Typedef with this name already declared.
MISRAC++2008-2-10-4	A class, struct, union or enum declaration that clashes with a previous declaration.
MISRAC++2008-2-10-5	A identifier is used that can clash with another static identifier.
MISRAC++2008-2-13-2	Uses of octal integer constants
MISRAC++2008-2-13-3	A U suffix shall be applied to all constants of unsigned type.
MISRAC++2008-2-13-4_a	Lower case suffixes on floating constants
MISRAC++2008-2-13-4_b	Lower case suffixes on integer constants
MISRAC++2008-3-1-1	Non-inline functions defined in header files

Table 5: Summary of checks

Check	Synopsis
MISRAC++2008-3-1-3	External arrays declared without size stated explicitly or defined implicitly by initialization.
MISRAC++2008-3-9-2	Uses of basic types char, int, short, long, double, and float without typedef
MISRAC++2008-3-9-3	An expression provides access to the bit-representation of a floating point variable.
MISRAC++2008-4-5-1	Uses of arithmetic operators on boolean operands.
MISRAC++2008-4-5-2	Use of unsafe operators on variable of enumeration type.
MISRAC++2008-4-5-3	Arithmetic on objects of type plain char, without an explicit signed or unsigned qualifier
MISRAC++2008-5-0-1_a	Expressions which depend on order of evaluation
MISRAC++2008-5-0-1_b	There shall be no more than one read access with volatile-qualified type within one sequence point
MISRAC++2008-5-0-1_c	There shall be no more than one modification access with volatile-qualified type within one sequence point
MISRAC++2008-5-0-2	Add parentheses to avoid implicit operator precedence.
MISRAC++2008-5-0-3	A cvalue expression shall not be implicitly converted to a different underlying type.
MISRAC++2008-5-0-4	An implicit integral conversion shall not change the signedness of the underlying type.
MISRAC++2008-5-0-5	There shall be no implicit floating-integral conversions.
MISRAC++2008-5-0-6	An implicit integral or floating-point conversion shall not reduce the size of the underlying type.
MISRAC++2008-5-0-7	There shall be no explicit floating-integral conversions of a cvalue expression.
MISRAC++2008-5-0-8	An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.
MISRAC++2008-5-0-9	An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.

Table 5: Summary of checks

Check	Synopsis
MISRAC++2008-5-0-10	Bitwise operation on unsigned char or unsigned short, that are not immediately cast to this type to ensure consistent truncation
MISRAC++2008-5-0-13_a	Non-boolean termination conditions in do ... while statements.
MISRAC++2008-5-0-13_b	Non-boolean termination conditions in for loops.
MISRAC++2008-5-0-13_c	Non-boolean conditions in if statements.
MISRAC++2008-5-0-13_d	Non-boolean termination conditions in while statements.
MISRAC++2008-5-0-14	Non-boolean operands to the conditional (? :) operator
MISRAC++2008-5-0-15_a	Array indexing shall be the only allowed form of pointer arithmetic.
MISRAC++2008-5-0-15_b	Array indexing shall only be applied to objects defined as an array type.
MISRAC++2008-5-0-16_a	Pointer arithmetic applied to a pointer that references a stack address
MISRAC++2008-5-0-16_b	Invalid pointer arithmetic with an automatic variable that is neither an array nor a pointer.
MISRAC++2008-5-0-16_c	Array access is out of bounds.
MISRAC++2008-5-0-16_d	Array access may be out of bounds, depending on which path is executed.
MISRAC++2008-5-0-16_e	A pointer to an array is used outside the array bounds
MISRAC++2008-5-0-16_f	A pointer to an array is potentially used outside the array bounds
MISRAC++2008-5-0-19	The declaration of objects should contain no more than two levels of pointer indirection.
MISRAC++2008-5-0-21	Applications of bitwise operators to signed operands
MISRAC++2008-5-2-4 (C++ only)	Uses of old style casts (other than void casts)
MISRAC++2008-5-2-5	Casts that remove any const or volatile qualification.
MISRAC++2008-5-2-6	A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.

Table 5: Summary of checks

Check	Synopsis
MISRAC++2008-5-2-7	A pointer to object type is cast to a pointer to different object type
MISRAC++2008-5-2-9	A cast should not be performed between a pointer type and an integral type.
MISRAC++2008-5-2-10	Uses of increment (++) and decrement (--) operators mixed with other operators in an expression.
MISRAC++2008-5-2-11_a (C++ only)	Overloaded && and || operators
MISRAC++2008-5-2-11_b (C++ only)	Overloaded comma operator
MISRAC++2008-5-3-1	Operands of logical operators (&&, ||, and !) that are not of type bool.
MISRAC++2008-5-3-2_a	Uses of unary - on unsigned expressions
MISRAC++2008-5-3-2_b	Uses of unary - on unsigned expressions
MISRAC++2008-5-3-3 (C++ only)	The & operator shall not be overloaded.
MISRAC++2008-5-3-4	Sizeof expressions containing side effects
MISRAC++2008-5-8-1	Out of range shifts
MISRAC++2008-5-14-1	Right hand operands of && or || that contain side effects
MISRAC++2008-5-18-1	Uses of the comma operator
MISRAC++2008-5-19-1	A constant unsigned integer expression overflows
MISRAC++2008-6-2-1	Assignment in a sub-expression.
MISRAC++2008-6-2-2	Floating point comparisons using == or !=
MISRAC++2008-6-2-3	Stray semicolons on the same line as other code
MISRAC++2008-6-3-1_a	Missing braces in do ... while statements
MISRAC++2008-6-3-1_b	Missing braces in for statements
MISRAC++2008-6-3-1_c	Missing braces in switch statements
MISRAC++2008-6-3-1_d	Missing braces in while statements
MISRAC++2008-6-4-1	Missing braces in if, else, and else if statements
MISRAC++2008-6-4-2	If ... else if constructs that are not terminated with an else clause.
MISRAC++2008-6-4-3	Switch statements that do not conform to the MISRA C switch syntax.

Table 5: Summary of checks

Check	Synopsis
MISRAC++2008-6-4-4	Switch labels in nested blocks.
MISRAC++2008-6-4-5	Non-empty switch cases not terminated by break
MISRAC++2008-6-4-6	Switch statements with no default clause, or a default clause that is not the final clause.
MISRAC++2008-6-4-7	A switch expression shall not represent a value that is effectively boolean.
MISRAC++2008-6-4-8	Switch statements with no cases.
MISRAC++2008-6-5-1_a	Floating-point values in the controlling expression of a for statement.
MISRAC++2008-6-5-2	Loop counter may not match loop condition test.
MISRAC++2008-6-5-3	A `for` loop counter variable is modified in the body of the loop.
MISRAC++2008-6-5-4	Potential inconsistent loop counter modification.
MISRAC++2008-6-5-6	A non-boolean variable is modified in the loop and used as loop condition.
MISRAC++2008-6-6-1	The target of the goto is a nested code block.
MISRAC++2008-6-6-2	Goto declared after target label.
MISRAC++2008-6-6-4	Multiple break points from loop.
MISRAC++2008-6-6-5	A function shall have a single point of exit at the end of the function.
MISRAC++2008-7-1-1	A local variable is not modified after its initialization and so should be const qualified.
MISRAC++2008-7-1-2	A function does not modify one of its parameters.
MISRAC++2008-7-2-1	Conversions to enum that are out of range of the enumeration.
MISRAC++2008-7-4-3	Inline asm statements that are not encapsulated in functions
MISRAC++2008-7-5-1_a (C++ only)	A stack object is returned from a function as a reference.
MISRAC++2008-7-5-1_b	May return address on the stack.
MISRAC++2008-7-5-2_a	Store a stack address in a global pointer.
MISRAC++2008-7-5-2_b	Store a stack address in the field of a global struct.
MISRAC++2008-7-5-2_c	Store stack address outside function via parameter.

Table 5: Summary of checks

Check	Synopsis
MISRAC++2008-7-5-2_d (C++ only)	Store stack address via reference parameter.
MISRAC++2008-7-5-4_a	Functions that call themselves directly.
MISRAC++2008-7-5-4_b	Functions that call themselves indirectly.
MISRAC++2008-8-0-1	Declarations shall only contain one variable or constant each.
MISRAC++2008-8-4-1	Functions defined using ellipsis (...) notation
MISRAC++2008-8-4-3	For some execution, no return statement is executed in a function with a non-void return type
MISRAC++2008-8-4-4	Function addresses taken without explicit &
MISRAC++2008-8-5-1_a	In all executions, a variable is read before it is assigned a value.
MISRAC++2008-8-5-1_b	In some execution, a variable is read before it is assigned a value.
MISRAC++2008-8-5-1_c	Dereference of an uninitialized or NULL pointer.
MISRAC++2008-8-5-2	This check points out where a non-zero array initialisation does not exactly match the structure of the array declaration.
MISRAC++2008-9-3-1 (C++ only)	A member function qualified as `const` returns a pointer member variable.
MISRAC++2008-9-3-2 (C++ only)	Member functions that return non-const handles to members
MISRAC++2008-9-5-1	All unions
MISRAC++2008-9-6-2	Bitfields with plain int type
MISRAC++2008-9-6-3	Bitfields with plain int type
MISRAC++2008-9-6-4	Signed single-bit fields (excluding anonymous fields)
MISRAC++2008-12-1-1_a (C++ only)	A virtual member function is called in a class constructor.
MISRAC++2008-12-1-1_b (C++ only)	A virtual member function is called in a class destructor.
MISRAC++2008-12-1-3 (C++ only)	All constructors that are callable with a single argument of fundamental type shall be declared `explicit`.
MISRAC++2008-15-0-2	Throw of exceptions by pointer
MISRAC++2008-15-1-2	Throw of NULL integer constant

Table 5: Summary of checks

Check	Synopsis
`MISRAC++2008-15-1-3 (C++ only)`	Unsafe rethrow of exception.
`MISRAC++2008-15-3-1 (C++ only)`	Exceptions thrown without a handler in some call paths leading to that point
`MISRAC++2008-15-3-2 (C++ only)`	No default exception handler for try.
`MISRAC++2008-15-3-3 (C++ only)`	Exception handler in constructor or destructor accesses non-static member variable that may not exist.
`MISRAC++2008-15-3-4 (C++ only)`	Calls to functions explicitly declared to throw an exception type that is not handled (or declared as thrown) by the caller
`MISRAC++2008-15-3-5 (C++ only)`	Catch of exception objects by value
`MISRAC++2008-15-5-1 (C++ only)`	An exception is thrown, or may be thrown, in a class' destructor.
`MISRAC++2008-16-0-3`	All #undef's
`MISRAC++2008-16-0-4`	Function-like macros
`MISRAC++2008-16-2-2 (C++ only)`	Definition of macros (except include guards)
`MISRAC++2008-16-2-3`	Header files without #include guards
`MISRAC++2008-16-2-4`	Illegal characters in header file names
`MISRAC++2008-16-2-5`	Illegal characters in header file names
`MISRAC++2008-16-3-1`	Multiple # or ## operators in a macro definition
`MISRAC++2008-16-3-2`	The # and ## operators should not be used
`MISRAC++2008-17-0-1`	#define or #undef of a reserved identifier in the standard library
`MISRAC++2008-17-0-3`	A library function is being overridden.
`MISRAC++2008-17-0-5`	All uses of <setjmp.h>
`MISRAC++2008-18-0-1 (C++ only)`	Uses of C library includes
`MISRAC++2008-18-0-2`	All uses of atof, atoi, atol and atoll
`MISRAC++2008-18-0-3`	All uses of abort, exit, getenv, and system
`MISRAC++2008-18-0-4`	All uses of <time.h> functions: asctime, clock, ctime, difftime, gmtime, localtime, mktime, strftime, and time
`MISRAC++2008-18-0-5`	All uses of strcpy, strcmp, strcat, strchr, strspn, strcspn, strpbrk, strrchr, strstr, strtok, and strlen
`MISRAC++2008-18-2-1`	All uses of the offsetof built-in function

Table 5: Summary of checks

Check	Synopsis
MISRAC++2008-18-4-1	All uses of malloc, calloc, realloc, and free
MISRAC++2008-18-7-1	All uses of <signal.h>
MISRAC++2008-19-3-1	All uses of errno
MISRAC++2008-27-0-1	All uses of <stdio.h>

Table 5: Summary of checks

Descriptions of checks

The following is detailed reference information about each check.

ARR-inv-index-pos

Synopsis

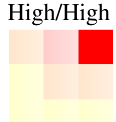
Array access might be out of bounds, depending on which path is executed.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

An element of an array is accessed, but one or more of the executable paths means that the element is outside the bounds of the array. This might corrupt data and/or crash the application, and result in security vulnerabilities.

Coding standards

CERT ARR33-C

Guarantee that copies are made into storage of sufficient size

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

CWE 121

Stack-based Buffer Overflow

CWE 124

Buffer Underwrite ('Buffer Underflow')

CWE 126

Buffer Over-read

CWE 127

Buffer Under-read

CWE 129

Improper Validation of Array Index

MISRA C:2012 Rule-18.1

(Required) A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand

MISRA C++ 2008 5-0-16

(Required) A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.

Code examples

The following code example fails the check and will give a warning:

```
int cond;

int main(void)
{
  int a[7];
  int x;

  if (cond)
    x = 3;
  else
    x = 20;

  a[x] = 0;  //x may be set to 20 in line 11
             //but a only has an interval of [0,6]
  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

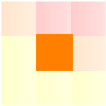
```
int cond;

int main(void)
{
  int a[25];
  int x;

  if (cond)
    x = 3;
  else
    x = 20;

  a[x] = 0;  //here, both possible values of
             //x are in the interval [0,24]
  return 0;
}
```

ARR-inv-index-ptr-pos

Synopsis
A pointer to an array is potentially used outside the array bounds

Enabled by default
Yes

Severity/Certainty
Medium/Medium



Full description
A pointer to an array is potentially used outside the array bounds. This might cause an invalid memory access, and might be a serious security risk. The application might also crash.

Coding standards
CERT ARR33-C

Guarantee that copies are made into storage of sufficient size

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

CWE 121

Stack-based Buffer Overflow

CWE 122

Heap-based Buffer Overflow

CWE 124

Buffer Underwrite ('Buffer Underflow')

CWE 126

Buffer Over-read

CWE 127

Buffer Under-read

CWE 129

Improper Validation of Array Index

MISRA C:2012 Rule-18.1

(Required) A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand

MISRA C++ 2008 5-0-16

(Required) A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.

Code examples

The following code example fails the check and will give a warning:

```
void example(int b) {
  int arr[11];
  int *p = arr;
  int x = (b<10 ? 8 : 11);
  p[x];
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(int b) {
  int arr[12];
  int *p = arr;
  int x = (b<10 ? 8 : 11);
  p[x];
}
```

ARR-inv-index-ptr

Synopsis

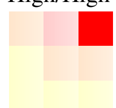
A pointer to an array is used outside the array bounds

Enabled by default

Yes

Severity/Certainty

High/High



Full description

A pointer to an array is used outside the array bounds. This will cause an invalid memory access, and might be a serious security risk. The application might also crash.

Coding standards

CERT ARR33-C

Guarantee that copies are made into storage of sufficient size

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

CWE 121

Stack-based Buffer Overflow

CWE 122

Heap-based Buffer Overflow

CWE 124

Buffer Underwrite ('Buffer Underflow')

CWE 126

Buffer Over-read

CWE 127

Buffer Under-read

CWE 129

Improper Validation of Array Index

MISRA C:2012 Rule-18.1

(Required) A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand

MISRA C++ 2008 5-0-16

(Required) A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int arr[10];
  int *p = arr;
  p[10];
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int arr[10];
  int *p = arr;
  p[9];
}
```

ARR-inv-index

Synopsis

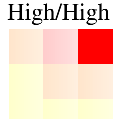
Array access is out of bounds.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

An element of an array is accessed when that element is outside the bounds of the array. This might corrupt data and/or crash the application, and result in security vulnerabilities.

Coding standards

CERT ARR33-C

Guarantee that copies are made into storage of sufficient size

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

CWE 121

Stack-based Buffer Overflow

CWE 124

Buffer Underwrite ('Buffer Underflow')

CWE 126

Buffer Over-read

CWE 127

Buffer Under-read

CWE 129

Improper Validation of Array Index

MISRA C:2012 Rule-18.1

(Required) A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand

MISRA C++ 2008 5-0-16

(Required) A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.

Code examples

The following code example fails the check and will give a warning:

```
/* Goanna correctly detects that the array access,
   a[x - 10] is always within bounds, because 'x'
   is always in the range 10 <= x < 20, but a[x]
   is not. */

int ex(int x, int y)
{
  int a[10];

  if((x >= 0) && (x < 20)) {
    if(x < 10) {
      y = a[x];
    } else {
      y = a[x - 10];
      y = a[x];
    }
  }

  return y;
}
```

The following code example passes the check and will not give a warning about this issue:

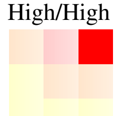
```
int main(void)
{
  int a[4];

  a[3] = 0;

  return 0;
}
```

ARR-neg-index

Synopsis — An array is accessed with a negative subscript value.

Enabled by default — Yes

Severity/Certainty — High/High



Full description

An array is accessed with a negative subscript value, causing an illegal memory access. This might corrupt data and/or crash the application, and result in security vulnerabilities.

Coding standards

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

CWE 124

Buffer Underwrite ('Buffer Underflow')

CWE 127

Buffer Under-read

Code examples

The following code example fails the check and will give a warning:

```
void foo(int n)
{
  int x[n];
  int i = 0;
  if (i == 0)
    i--;
  x[i] = 5; //i is -1 at this point
}
```

The following code example passes the check and will not give a warning about this issue:

```
void foo(int n)
{
  int x[n];
  int i = 5;
  if (i == 0)
    i--;
  x[i] = 5;  //OK, since i is 4
}
```

ARR-uninit-index

Synopsis

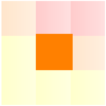
An array is indexed with an uninitialized variable

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

An array is indexed with an uninitialized variable. The value of the variable is not defined, which might cause an array overrun.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
int example(int b[20]) {
  int a;
  return b[a];
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(int b[20]) {
  int a;
  a = 5;
  return b[a];
}
```

ATH-cmp-float

Synopsis

Floating point comparisons using == or !=

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

A comparison for equality with a floating-point type uses the `==` or `!=` operator. This might have an unexpected result because the value of the `float` varies with the environment and the operation. The comparison might be evaluated incorrectly, especially if either of the floating-point numbers has been operated on arithmetically. In that case, the application logic will be compromised.

Coding standards

CERT FLP06-C

Understand that floating-point arithmetic in C is inexact

CERT FLP35-CPP

Take granularity into account when comparing floating point values

MISRA C:2004 13.3

(Required) Floating-point expressions shall not be tested for equality or inequality.

MISRA C++ 2008 6-2-2

(Required) Floating-point expressions shall not be directly or indirectly tested for equality or inequality.

Code examples

The following code example fails the check and will give a warning:

```
int main(void)
{
  float f = 3.0;
  int i = 3;

  if (f == i) //comparison of a float and an int
    ++i;

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int main(void)
{
  int i = 60;
  char c = 60;

  if (i == c)
    ++i;

  return 0;
}
```

ATH-cmp-unsign-neg

Synopsis

An unsigned value is compared to see whether it is negative.

Enabled by default
Yes

Severity/Certainty
Low/High

Full description
A comparison is performed on an unsigned value, to see whether it is negative. This comparison always returns false, and is redundant.

Coding standards
This check does not correspond to any coding standard rules.

Code examples
The following code example fails the check and will give a warning:

```
int foo(unsigned int x)
{
  if (x < 0)  //checking an unsigned for negativity
    return 1;
  else
    return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int foo(unsigned int x)
{
  if (x < 1)  //OK - x might be 0
    return 1;
  else
    return 0;
}
```

ATH-cmp-unsign-pos

Synopsis
An unsigned value is compared to see whether it is greater than or equal to `0`.

Enabled by default
Yes

Severity/Certainty

Low/High

Full description

A comparison is performed on an unsigned value, to see whether it is greater than or equal to `0`. This comparison always returns true, and is redundant.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
int foo(unsigned int x)
{
  if (x >= 0)  //checking an unsigned for negativity
    return 1;
  else
    return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int foo(unsigned int x)
{
  if (x > 0)  //OK - x might be 0
    return 1;
  else
    return 0;
}
```

ATH-div-0-assign

Synopsis

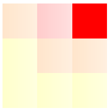
A variable is assigned the value `0`, then used as a divisor.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

A variable is assigned the value `0`, then used as a divisor. This will cause a 'divide by zero' runtime error.

Coding standards

CERT INT33-C

Ensure that division and modulo operations do not result in divide-by-zero errors

CWE 369

Divide By Zero

MISRA C:2004 1.2

(Required) No reliance shall be placed on undefined or unspecified behavior.

MISRA C:2012 Rule-1.3

(Required) There shall be no occurrence of undefined or critical unspecified behaviour

Code examples

The following code example fails the check and will give a warning:

```
int foo(void)
{
  int a = 20, b = 0, c;

  c = a / b;     /* Divide by zero */

  return c;
}
```

The following code example passes the check and will not give a warning about this issue:

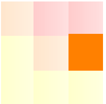
```
int foo(void)
{
  int a = 20, b = 5, c;

  c = a / b; /* b is not 0 */

  return c;
}
int main() {
    int totallen = 0;
    int i=0;
    float tmp=1;

   for( i=1; i<10; i++){
   totallen++;
   }

   foo(2/totallen);

   return 0;
}

int foo(int x){
   return x;
}
```

ATH-div-0-cmp-aft

Synopsis After a successful comparison with `0`, a variable is used as a divisor.

Enabled by default Yes

Severity/Certainty Medium/High



Full description A variable is successfully compared to `0`, then used as a divisor. This will cause a 'divide by zero' runtime error.

Coding standards CERT INT33-C

Ensure that division and modulo operations do not result in divide-by-zero errors

CWE 369

Divide By Zero

MISRA C:2004 1.2

(Required) No reliance shall be placed on undefined or unspecified behavior.

MISRA C:2012 Rule-1.3

(Required) There shall be no occurrence of undefined or critical unspecified behaviour

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>
int foo(void)
{
  int a = 20;
  int p = rand();

  if (p == 0)   /* p is 0 */
    a = 34 / p;

  return a;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>
int foo(void)
{
  int a = 20;
  int p = rand();

  if (p != 0)   /* p is not 0 */
    a = 34 / p;

  return a;
}
```

ATH-div-0-cmp-bef

Synopsis

A variable used as a divisor is afterwards compared with `0`.

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

A variable is compared to `0` after it is used as a divisor, but before it is written to again. This implies that the variable's value might be `0`, and might have been for the preceding statements. Because one of these statements is an operation that uses the variable as a divisor (causing a 'divide by zero' runtime error), the execution can never reach the comparison when the value is `0`, making it redundant.

Coding standards

CERT INT33-C

Ensure that division and modulo operations do not result in divide-by-zero errors

CWE 369

Divide By Zero

MISRA C:2004 1.2

(Required) No reliance shall be placed on undefined or unspecified behavior.

MISRA C:2012 Rule-1.3

(Required) There shall be no occurrence of undefined or critical unspecified behaviour

Code examples

The following code example fails the check and will give a warning:

```
int foo(int p)
{
  int a = 20, b = 1;
  b = a / p;
  if (p == 0) // Checking the value of 'p' too late.
    return 0;
  return b;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int foo(int p)
{
  int a = 20, b;
  if (p == 0)
    return 0;
  b = a / p;     /* Here 'p' is non-zero. */
  return b;
}
```

ATH-div-0-interval

Synopsis

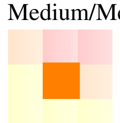
Interval analysis has found a value that is `0` and used as a divisor.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

Interval analysis has found a value that is `0` and used as a divisor. This might cause a 'divide by zero' runtime error.

Coding standards

CERT INT33-C

> Ensure that division and modulo operations do not result in divide-by-zero errors

CWE 369

> Divide By Zero

MISRA C:2004 1.2

> (Required) No reliance shall be placed on undefined or unspecified behavior.

MISRA C:2012 Rule-1.3

> (Required) There shall be no occurrence of undefined or critical unspecified behaviour

Code examples

The following code example fails the check and will give a warning:

```
int foo(void)
{
  int a = 1;
  a--;
  return 5 / a;   /* a is 0 */
}
```

The following code example passes the check and will not give a warning about this issue:

```
int foo(void)
{
  int a = 2;
  a--;
  return 5 / a;   /* OK - a is 1 */
}
```

ATH-div-0-pos

Synopsis

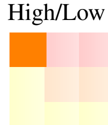
Interval analysis has found an expression that might be `0` and is used as a divisor.

Enabled by default

Yes

Severity/Certainty

High/Low



Full description

Interval analysis has found an expression that contains `0` and is used as a divisor. This might cause a 'divide by zero' runtime error.

Coding standards

CERT INT33-C

> Ensure that division and modulo operations do not result in divide-by-zero errors

CWE 369

> Divide By Zero

MISRA C:2004 1.2

> (Required) No reliance shall be placed on undefined or unspecified behavior.

MISRA C:2012 Rule-1.3

(Required) There shall be no occurrence of undefined or critical unspecified behaviour

Code examples

The following code example fails the check and will give a warning:

```
int main (void)

{

  int x = 2;

  int i;



  /* The second iteration leads to a division by zero*/

  for (i = 1; i < 3; i++) { x = x / (2 - i); }
/*@@ZDV-RED@@ */



  return x;

}

int foo(void)
{
  int a = 3;
  a--;
  return 5 / (a-2);  // a-2 is 0
}
```

The following code example passes the check and will not give a warning about this issue:

```
int foo(void)
{
  int a = 3;
  a--;
  return 5 / (a+2);  // OK - a+2 is 4
}
```

ATH-div-0-unchk-global

Synopsis

A global variable is used as a divisor without having been determined to be non-zero.

Enabled by default

Yes

Severity/Certainty

Medium/Low

Full description

A global variable is used as a divisor without having been determined to be non-zero. This will cause a 'divide by zero' runtime error if the variable has a value of `0`.

Coding standards

CWE 369

Divide By Zero

MISRA C:2004 1.2

(Required) No reliance shall be placed on undefined or unspecified behavior.

MISRA C:2012 Rule-1.3

(Required) There shall be no occurrence of undefined or critical unspecified behaviour

Code examples

The following code example fails the check and will give a warning:

```
int x;

int example() {
  return 5/x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int x;

int example() {
  if (x != 0){
    return 5/x;
  }
}
```

ATH-div-0-unchk-local

Synopsis

A local variable is used as a divisor without having been determined to be non-zero.

Enabled by default

Yes

Severity/Certainty

Medium/Low

Full description

A local variable is used as a divisor without having been determined to be non-zero. This will cause a 'divide by zero' runtime error if the variable has a value of `0`.

Coding standards

CWE 369

Divide By Zero

MISRA C:2004 1.2

(Required) No reliance shall be placed on undefined or unspecified behavior.

MISRA C:2012 Rule-1.3

(Required) There shall be no occurrence of undefined or critical unspecified behaviour

Code examples

The following code example fails the check and will give a warning:

```
int rand();

int example() {
    int x = rand();
    return 5/x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int rand();

int example() {
  int x = rand();
  if (x != 0){
    return 5/x;
  }
}
```

ATH-div-0-unchk-param

Synopsis

A parameter is used as a divisor without having been determined to be non-zero.

Enabled by default

Yes

Severity/Certainty

Medium/Low

Full description

A parameter is used as a divisor without having been determined to be non-zero. This will cause a 'divide by zero' runtime error if the parameter has a value of `0`.

Coding standards

CWE 369

Divide By Zero

Code examples

The following code example fails the check and will give a warning:

```
int example(int x) {
  return 5/x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(int x) {
  if (x != 0){
    return 5/x;
  }
}
```

ATH-div-0

Synopsis

An expression that results in `0` is used as a divisor.

Enabled by default

Yes

Severity/Certainty

High/High

Full description

An expression that results in `0` is used as a divisor. This will cause a 'divide by zero' runtime error.

Coding standards

CERT INT33-C

Ensure that division and modulo operations do not result in divide-by-zero errors

CWE 369

Divide By Zero

MISRA C:2004 1.2

(Required) No reliance shall be placed on undefined or unspecified behavior.

MISRA C:2012 Rule-1.3

(Required) There shall be no occurrence of undefined or critical unspecified behaviour

Code examples

The following code example fails the check and will give a warning:

```
int foo(void)
{
  int a = 3;
  a--;
  return 5 / (a-2);  // a-2 is 0
}
#include <stdlib.h>

int main (void)

{
 int *p = malloc( sizeof(int));
  int x = foo (p);
  /* foo(2) returns 8, so we have a division by zero below)*/
  x = 1 / (x - 8);              /*@@ZDV-RED@@ */

  return x;
}


int foo(int * p){
  return 8;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int foo(void)
{
  int a = 3;
  a--;
  return 5 / (a+2);  // OK - a+2 is 4
}
```

ATH-inc-bool (C++ only)

Synopsis

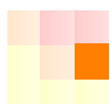
Deprecated operation on `bool`.

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

An undefined increment or decrement operation is performed on a `bool` value. In older versions of C++, Boolean values were modeled by a `typedef` to an integer type, allowing increment and decrement operations. These types are deprecated in Standard C++ and the operations no longer apply to the built-in C++ `bool` type.

Coding standards

CWE 480

Use of Incorrect Operator

Code examples

The following code example fails the check and will give a warning:

```
int main(void)
{
  bool x = true;
  ++x;  //this operation is undefined for a bool
}
```

The following code example passes the check and will not give a warning about this issue:

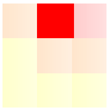
```
int main(void)
{
  int x = 0;
  ++x;  //OK - x is an int
}
```

ATH-malloc-overrun

Synopsis — The size of memory passed to malloc to allocate overflows.

Enabled by default — Yes

Severity/Certainty — High/Medium



Full description — The size of memory passed to malloc to allocate is the result of an arithmetic overflow. As a result, malloc will not allocate the expected amount of memory and accesses to this memory might cause runtime errors.

Coding standards — This check does not correspond to any coding standard rules.

Code examples — The following code example fails the check and will give a warning:

```
#include <stdlib.h>
#include <limits.h>

void example(void) {
  int *b = malloc(sizeof(int)*ULONG_MAX*ULONG_MAX);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>
#include <limits.h>

void example(void) {
  int *b = malloc(sizeof(int)*5);
}
```

ATH-neg-check-nonneg

Synopsis — A variable is checked for a non-negative value after being used, instead of before.

Enabled by default — Yes

Severity/Certainty

Low/High

Full description

A function parameter or index is used in a context that implicitly asserts that it is not negative, but it is not determined to be non-negative until after it is used. If the value actually is negative when the variable is used, data might be corrupted, the application might crash, or a security vulnerability might be exposed.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>
int foo(int p)
{
  int *x = malloc(p); // p was an argument to malloc(),
                      // so it is not negative

  if (p < 0)
    return 0;

  return p;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>
int foo(int p)
{
  int *x;

  if (p < 0)
    return 0;

  x = malloc(p);  // OK - p is non-negative

  return p;
}
#include <stdlib.h>
int foo(int p)
{
  int *x;

  if (p < 1)
    p= 1;

  x = malloc(p);  // OK - p is non-negative

  return p;
}
```

ATH-neg-check-pos

Synopsis

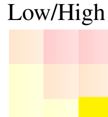
A variable is checked for a positive value after being used, instead of before.

Enabled by default

Yes

Severity/Certainty

Low/High



Full description

A function parameter or index is used in a context that implicitly asserts that it is positive, but it is not compared to `0` until after it is used. If the value actually is negative or `0` when the variable is used, data might be corrupted, the application might crash, or a security vulnerability might be exposed.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>
int foo(int p)
{
  int *x = malloc(p);

  // p was an argument to malloc(), so not negative

  if (p <= 0)
    return 0;

  return p;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>
int foo(int p)
{
  int *x;

  if (p < 0)
    return 0;

  x = malloc(p);  // OK - p is non-negative

  return p;
}
```

ATH-new-overrun (C++ only)

Synopsis

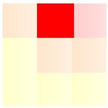
An arithmetic overflow is caused by an allocation using new[].

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

The new a[n] operator performs the operation sizeof(a) * n. This might cause an overflow, leading to an unexpected amount of memory being allocated. Dereferencing this memory might lead to a runtime error.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <new>
#include <climits>

void example(void) {
#ifdef __LP64__
  unsigned long b = (ULONG_MAX / 4) + 1;
#else
  unsigned int b = (UINT_MAX / 4) + 1;
#endif
  int *a = new int[b];
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <new>


void example(void) {
  int *a = new int[10];
}
```

ATH-overflow-cast

Synopsis

An expression is cast to a different type, resulting in an overflow or underflow of its value.

Enabled by default

No

Severity/Certainty

Medium/High

Full description

An expression is cast to a different type, resulting in an overflow or underflow of its value. This might be unintended and can cause logic errors. Because unexpected behavior is much more likely than an application crash, such errors can be very hard to find.

Coding standards

CERT INT31-C

Ensure that integer conversions do not result in lost or misinterpreted data

CWE 194

Unexpected Sign Extension

CWE 195

Signed to Unsigned Conversion Error

CWE 196

Unsigned to Signed Conversion Error

CWE 197

Numeric Truncation Error

CWE 680

Integer Overflow to Buffer Overflow

Code examples

The following code example fails the check and will give a warning:

```
typedef int I;
typedef I J;

void f(){
  J x = 375;
  char c = (char)x;  //overflows to 120
}
```

The following code example passes the check and will not give a warning about this issue:

```
void f(){
  int x = 35;
  char c = (char)x;
}
```

ATH-overflow

Synopsis

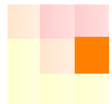
An expression is implicitly converted to a narrower type, resulting in an overflow or underflow of its value.

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

An expression is implicitly converted to a narrower type, resulting in an overflow or underflow of its value. This might be unintended and can cause logic errors. Because unexpected behavior is much more likely than an application crash, such errors can be very hard to find.

Coding standards

CERT INT31-C

Ensure that integer conversions do not result in lost or misinterpreted data

CWE 194

Unexpected Sign Extension

CWE 195

Signed to Unsigned Conversion Error

CWE 196

Unsigned to Signed Conversion Error

CWE 197

Numeric Truncation Error

CWE 680

Integer Overflow to Buffer Overflow

Code examples

The following code example fails the check and will give a warning:

```
typedef int I;
typedef I J;

void f(){
  J x = 375;
  char c = x;  //overflows to 120
}
```

The following code example passes the check and will not give a warning about this issue:

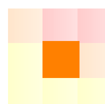
```
void f(){
  int x = 35;
  char c = x;
}
```

ATH-shift-bounds

Synopsis
Out of range shifts

Enabled by default
Yes

Severity/Certainty
Medium/Medium



Full description
The right-hand operand of a shift operator might be negative or too large. A shift operator on an `n`-bit argument should only shift between `0` and `n-1` bits. The behavior here is undefined; the code might work as intended, or data could become erroneous.

Coding standards
CERT INT34-C

Do not shift a negative number of bits or more bits than exist in the operand

CWE 682

Incorrect Calculation

MISRA C:2004 12.8

(Required) The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand.

MISRA C:2012 Rule-12.2

(Required) 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

MISRA C++ 2008 5-8-1

(Required) The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.

Code examples
The following code example fails the check and will give a warning:

```
unsigned int foo(unsigned long long x, unsigned int y)
{
  int shift = 65; // too big
  return 3ULL << shift;
}
unsigned int foo(unsigned int x, unsigned int y)
{
  int shift = 33; // too big
  return 3U << shift;
}
```

The following code example passes the check and will not give a warning about this issue:

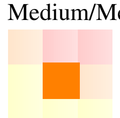
```
unsigned int foo(unsigned int x)
{
  int y = 1;  // OK - this is within the correct range
  return x << y;
}
unsigned int foo(unsigned long long x)
{
  int y = 63; // ok
  return x << y;
}
```

ATH-shift-neg

Synopsis
The left-hand side of a right shift operation might be a negative value.

Enabled by default
Yes

Severity/Certainty
Medium/Medium



Full description
The left-hand side of a right shift operation might be a negative value. Because performing a right shift operation on a negative number is implementation-defined, this operation might have unexpected results.

Coding standards
CWE 682

Incorrect Calculation

Code examples
The following code example fails the check and will give a warning:

```
int example(int x) {
  return -10 >> x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(int x) {
  return 10 >> x;
}
```

ATH-sizeof-by-sizeof

Synopsis

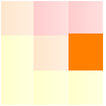
Multiplying `sizeof` by `sizeof`.

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

`sizeof` is multiplied by `sizeof`. This is probably a programming mistake and might have been intended to be `sizeof / sizeof`. This code will not cause any errors, but the product of two `sizeof` results is not a useful value, and might indicate a misunderstanding of the intended behavior of the code.

Coding standards

CWE 480

Use of Incorrect Operator

Code examples

The following code example fails the check and will give a warning:

```
void foo(void)
{
  int x = sizeof(int) * sizeof(char);  //sizeof * sizeof
}
```

The following code example passes the check and will not give a warning about this issue:

```
void foo(void)
{
  int x = sizeof(int) * 7;  //OK
}
```

CAST-old-style (C++ only)

Synopsis

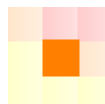
Old style casts (other than void casts) are used

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

Old style casts (other than void casts) are used. These casts override type information about the variables or pointers being cast, which might cause portability problems. A particular cast might for example not be valid on a system, but the compiler will perform the cast anyway. The new style casts static_cast, const_cast, and reinterpret_cast should be used instead because they make clear the intention of the cast. Moreover, the new style casts can easily be searched for in source code files, unlike old style casts.

Coding standards

CERT EXP05-CPP

Do not use C-style casts

MISRA C++ 2008 5-2-4

(Required) C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
int example(float b)
{
    return (int)b;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(float b)
{
    return static_cast<int>(b);
}
```

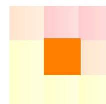
CATCH-object-slicing (C++ only)

Synopsis

Exception objects are caught by value

Enabled by default Yes

Severity/Certainty Medium/Medium



Full description Class type exception objects are caught by value, leading to slicing. That is, if the exception object is of a derived class and is caught as the base, only the base class' functions (including virtual functions) can be called. Moreover, any additional member data in the derived class cannot be accessed. If the exception is instead caught by reference, slicing does not occur.

Coding standards CERT ERR09-CPP

> Throw anonymous temporaries and catch by reference

MISRA C++ 2008 15-3-5

> (Required) A class type exception shall always be caught by reference.

Code examples The following code example fails the check and will give a warning:

```
typedefcharchar_t;

// base class for exceptions
class ExpBase {
public:
    virtual const char_t *who ( ) { return "base"; }
};

class ExpD1: public ExpBase {
public:
    virtual const char_t *who ( ) { return "type 1 exception"; }
};

class ExpD2: public ExpBase {
public:
    virtual const char_t *who ( ) { return "type 2 exception"; }
};

void example()
{
    try {
        // ...
        throw ExpD1 ( );
        // ...
        throw ExpBase ( );
    }
    catch ( ExpBase b ) { // Non-compliant - derived type objects
will be
                          // caught as the base type
        b.who();          // Will always be "base"
        throw b;          // The exception re-thrown is of the
base class,
                          // not the original exception type
    }
}
```

The following code example passes the check and will not give a warning about this issue:

```
typedefcharchar_t;

// base class for exceptions
class ExpBase {
public:
    virtual const char_t *who ( ) { return "base"; }
};

class ExpD1: public ExpBase {
public:
    virtual const char_t *who ( ) { return "type 1 exception"; }
};

class ExpD2: public ExpBase {
public:
    virtual const char_t *who ( ) { return "type 2 exception"; }
};

void example()
{
    try {
        // ...
        throw ExpD1 ( );
        // ...
        throw ExpBase ( );
    }
    catch ( ExpBase &b ) { // Compliant - exceptions caught by
reference
        // ...
        b.who(); // "base", "type 1 exception" or "type 2
exception"
                  // depending upon the type of the thrown object
    }
}
```

CATCH-xtor-bad-member (C++ only)

Synopsis — Exception handler in constructor or destructor accesses non-static member variable that might not exist.

Enabled by default — No

Severity/Certainty

Medium/Low

Full description

The exception handler in a constructor or destructor accesses a non-static member function. Such members might or might not exist at this point in construction/destruction and accessing them might result in undefined behavior.

Coding standards

MISRA C++ 2008 15-3-3

(Required) Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.

Code examples

The following code example fails the check and will give a warning:

```
int throws();

class C
{
public:
  int x;
  static char c;
  C ( )
  {
    x = 0;
  }

  ~C ( )
  {
    try
    {
      throws();
      // Action that may raise an exception
    }
    catch ( ... )
    {
      if ( 0 == x ) // Non-compliant - x may not exist at this
point
      {
        // Action dependent on value of x
      }
    }
  }
};
```

The following code example passes the check and will not give a warning about this issue:

```
class C
{
public:
  int x;
  static char c;
  C ( )
  {
    try
    {
      // Action that may raise an exception
    }
    catch ( ... )
    {
      if ( 0 == c )
      {
        // Action dependent on value of c
      }
    }
  }

  ~C ( )
  {
    try
    {
      // Action that may raise an exception
    }
    catch (int i) {}
    catch ( ... )
    {
      if ( 0 == c )
      {
        // Action dependent on value of c
      }
    }
  }
};
```

COMMA-overload (C++ only)

Synopsis	Overloaded comma operator
Enabled by default	No

Severity/Certainty

Low/Low

Full description

There are overloaded versions of the comma and logical conjunction operators. These have the semantics of function calls whose sequence point and ordering semantics are different from those of the built-in versions. Because it might not be clear at the point of use that these operators are overloaded, developers might be unaware which semantics apply.

Coding standards

MISRA C++ 2008 5-2-11

> (Required) The comma operator, && operator and the || operator shall not be overloaded.

Code examples

The following code example fails the check and will give a warning:

```
class C{
  bool x;
  bool operator,(bool other);
};

bool C::operator,(bool other){
  return x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
class C{
  int x;
  int operator+(int other);
};

int C::operator+(int other){
  return x + other;
}
```

COMMENT-nested

Synopsis

Appearances of /* inside comments

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

Appearances of /* inside comments. C does not support nesting of comments. This can cause confusion when some code does not execute as expected. For example: /* A comment, end comment marker accidentally omitted <<New Page>> initialize(X); /* this comment is not compliant */ In this case, X will not be initialized because the code is hidden in a comment.

Coding standards

MISRA C:2004 2.3

> (Required) The character sequence /* shall not be used within a comment.

MISRA C++ 2008 2-7-1

> (Required) The character sequence /* shall not be used within a C-style comment.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  /* This comment starts here
  /* Nested comment starts here
  */
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  /* This comment starts here */
  /* Nested comment starts here
  */
}
```

CONST-local

Synopsis

A local variable that is not modified after initialization is not declared const.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

A local variable that is not modified after initialization is not declared `const`. Declaring it `const` makes it more clear that it will not be changed and makes the compiler warn if the application tries to write to the variable.

Coding standards

MISRA C++ 2008 7-1-1

(Required) A variable which is not modified shall be const qualified.

Code examples

The following code example fails the check and will give a warning:

```
int example( void ){
  int x = 7;
  return x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example( void ){
  int x = 7;
  ++x;
  return x;
}
```

CONST-member-ret (C++ only)

Synopsis

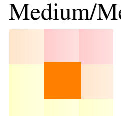
A member function qualified as `const` returns a pointer member variable.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

A member function qualified as `const` returns a pointer member variable. This might violate the semantics of the function's `const` qualification, as the data at that address might be overwritten, or the memory itself might be freed. This will not be identified by a compiler, because the pointer being returned is a copy even though the memory to which it refers is vulnerable.

Coding standards

MISRA C++ 2008 9-3-1

> (Required) const member functions shall not return non-const pointers or references to class-data.

Code examples

The following code example fails the check and will give a warning:

```
class C{
  int* foo() const {
    return p;
  }
  int* p;
};
```

The following code example passes the check and will not give a warning about this issue:

```
class C{
  int* foo() {
    return p;
  }
  int* p;
};
```

CONST-param

Synopsis

A function does not modify one of its parameters.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

A function does not modify one of its parameters. A parameter that is either a pointer or a reference should be `const`-qualified if it is not modified by the function. That way callers will be able to provide a `const` object as an argument, making the function more inclusive. It will also cause a compile-time error if a non-`const` object is mistakenly used as an argument.

Coding standards

MISRA C:2004 16.7

(Required) A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.

MISRA C++ 2008 7-1-2

(Required) A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.

Code examples

The following code example fails the check and will give a warning:

```
int example(int* x) {  //x should be const
  if (*x > 5){
    return *x;
  } else {
    return 5;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(const int* x) {  //OK
  if (*x > 5){
    return *x;
  } else {
    return 5;
  }
}
```

COP-alloc-ctor (C++ only)

Synopsis

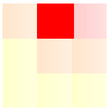
A class member is deallocated in the class' destructor, but not allocated in a constructor or assignment operator.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

A class member is deallocated in the class' destructor but is not allocated in a constructor or assignment operator (`operator=`). Even if this is intentional (and the class' pointer attributes are allocated elsewhere) it is still dangerous, because it subverts the Resource

Acquisition is Initialization convention, and consequently users of the class might accidentally misuse it.

Coding standards

CWE 401

Improper Release of Memory Before Removing Last Reference ('Memory Leak')

Code examples

The following code example fails the check and will give a warning:

```
class MyClass{
  int *p;

public:
  MyClass(){}  //p is not allocated in
               //this constructor
  ~MyClass(){
    delete p;
  }
};
```

The following code example passes the check and will not give a warning about this issue:

```
class MyClass{
  int *p;

 public:
  MyClass(){
     p = new int(0);  //OK - p is allocated
  }

  ~MyClass(){
    delete p;
  }
};
```

COP-assign-op-ret (C++ only)

Synopsis

An assignment operator of a C++ class does not return a non-`const` reference to `this`.

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

An assignment operator of a C++ class is incorrectly defined. Probably it does not return a non-`const` reference to the left-hand side of the assignment. This can cause unexpected behavior in situations where the assignment is chained with others, or the return value is used as a left-hand side argument to a subsequent assignment. A non-`const` reference as the return type should be used because it is the convention; it will not achieve any added code safety, and it makes the assignment operator more restrictive.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
class MyClass{
  int x;
public:
  MyClass &operator=(MyClass &rhs){
    x = rhs.x;
    return rhs; // should return *this
  }
};
```

The following code example passes the check and will not give a warning about this issue:

```
class MyClass{
  int x;
public:
  MyClass &operator=(const MyClass &rhs) {
    x = rhs.x;
    return *this; // a properly defined operator=
  }
};
```

COP-assign-op-self (C++ only)

Synopsis

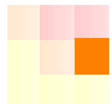
Assignment operator does not check for self-assignment before allocating member functions

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

An assignment operator does not check for self-assignment before allocating member functions. If self-assignment occurs in a user-defined object which uses dynamic memory allocation, references to allocated memory will be lost if they are reassigned. This will most likely cause a memory leak, as well as unexpected results, because the objects referred to by any pointers are lost.

Coding standards

CERT MEM42-CPP

> Ensure that copy assignment operators do not damage an object that is copied to itself

Code examples

The following code example fails the check and will give a warning:

```
class MyClass{
  int* p;
  MyClass& operator=(const MyClass& rhs){
    p = new int(*(rhs.p));  //reference to the old
                            //memory is lost
    return *this;
  }
};
```

The following code example passes the check and will not give a warning about this issue:

```
class MyClass{
  int* p;
  MyClass& operator=(const MyClass& rhs){
    if (&rhs != this)   //the pointer is not reallocated
                        //if the object is assigned to itself
      p = new int(*(rhs.p));
    return *this;
  }
};
```

COP-assign-op (C++ only)

Synopsis

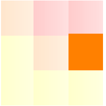
There is no assignment operator defined for a class whose destructor deallocates memory.

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

There is no assignment operator defined for a class whose destructor deallocates memory, so the compiler's synthesized assignment operator will be created and used if needed. This will only perform shallow copies of any pointer values, meaning that multiple instances of a class might inadvertently contain pointers to the same memory. Although a synthesized assignment operator might be adequate and appropriate for classes whose members include only (non-pointer) built-in types, in a class that dynamically allocates memory it could easily lead to unexpected behavior or attempts to access freed memory. In that case, if a copy is made and one of the two is destroyed, any deallocated pointers in the other will become invalid. This check should only be selected if all of a class' copy control functions are defined in the same file.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
class MyClass{
  int* p;
public:
  ~MyClass(){
    delete p;  //this class has no assignment operator
  }
};

int main(){
  MyClass *original = new MyClass;
  MyClass copy;
  copy = *original;  //copy's p == original's p
  delete original;  //p is deallocated; copy now has an invalid
pointer
}
```

The following code example passes the check and will not give a warning about this issue:

```
class MyClass{
  int* p;

  ~MyClass(){
    delete p;  //OK - the assignment operator will
               //not be synthesized
  }

  MyClass& operator=(const MyClass& rhs){
    if (this != &rhs)
      p = new int;
    return *this;
  }
};
```

COP-copy-ctor (C++ only)

Synopsis

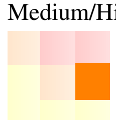
A class which uses dynamic memory allocation does not have a user-defined copy constructor.

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

A class which uses dynamic memory allocation does not have a user-defined copy constructor, so the compiler's synthesized copy constructor will be created and used if needed. This will only perform shallow copies of any pointer values, meaning that multiple instances of a class might inadvertently contain pointers to the same memory. Although a synthesized copy constructor might be adequate and appropriate for classes whose members include only (non-pointer) built-in types, in a class that dynamically allocates memory, it might easily lead to unexpected behavior or attempts to access freed memory. In that case, if a copy is made and one of the two is destroyed, any deallocated pointers in the other will become invalid. This check should only be selected if all of a class' copy control functions are defined in the same file.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
class MyClass{
  int *p;
 public:
  MyClass(){      //not a copy constructor
    p = new int;  //one will be synthesized
  }

  ~MyClass(){
    delete p;
  }
};

int main(){
  MyClass *original = new MyClass;
  MyClass copy(*original);  //copy's p == original's p
  delete original;  //p is deallocated; copy now has an invalid
pointer
}
```

The following code example passes the check and will not give a warning about this issue:

```
class MyClass{
  int *p;
 public:

  MyClass(MyClass& rhs){
    p = new int;
    *p = *(rhs.p);
  }

  ~MyClass(){
    delete p;
  }
};
```

COP-dealloc-dtor (C++ only)

Synopsis

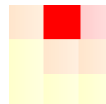
A class member has memory allocated in a constructor or an assignment operator, that is not released in the destructor.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

A class member has memory allocated to it in a constructor or assignment operator, that is not released in the class' destructor. This will most likely cause a memory leak when objects of this class are created and destroyed. Even if this is intentional (and the memory is released elsewhere) it is still dangerous, because it subverts the Resource Acquisition is Initialization convention, and consequently users of the class might not release the memory at all.

Coding standards

CWE 401

Improper Release of Memory Before Removing Last Reference ('Memory Leak')

Code examples

The following code example fails the check and will give a warning:

```
class MyClass{
  int *p;

public:
  MyClass() {
    p = 0;
  }

  MyClass(int i) {
    p = new int[i];
  }

  ~MyClass() {}  //p not deleted here
};

int main(void){
  MyClass *cp = new MyClass(5);
  delete cp;
}
```

The following code example passes the check and will not give a warning about this issue:

```
class MyClass{
  int *p;

public:
  MyClass(){
    p = 0;
  }

  MyClass(int i){
    p = new int[i];
  }

  ~MyClass(){
    if(p)
      delete[] p;  //OK - p is deleted here
  }
};

int main(void){
  MyClass *cp = new MyClass(5);
  delete cp;
}
```

COP-dtor-throw (C++ only)

Synopsis

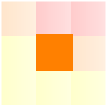
An exception is thrown, or might be thrown, in a class destructor.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

An exception is thrown, or might be thrown, in a class destructor. When the destructor is called, stack unwinding takes place. If an exception is thrown at this time, the application will crash.

Coding standards

CERT ERR33-CPP

Destructors must not throw exceptions

MISRA C++ 2008 15-5-1

(Required) A class destructor shall not exit with an exception.

Code examples

The following code example fails the check and will give a warning:

```
class E{};

class C {
  ~C() {
    if (!p){
      throw E();  //may throw an exception here
    }
  }
  int* p;
};
class E{};

void do_something();

class C {
 ~C() throw (E) {  //may throw an exception
    if (!p){
      do_something();
    }
  }
  int* p;
};
```

The following code example passes the check and will not give a warning about this issue:

```
void do_something();

class C {
  ~C() {  //OK
    if (!p){
      do_something();
    }
  }
  int* p;
};
```

COP-dtor (C++ only)

Synopsis

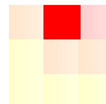
A class which dynamically allocates memory in its copy control functions does not have a destructor.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

A class which dynamically allocates memory in its copy control functions does not have a destructor. This will most likely result in a memory leak. If memory is dynamically allocated in the constructors or assignment operators, there must be a matching destructor to free it. If a destructor is not defined, the compiler will synthesize one, which will destroy any pointers but will not release their contents back to the heap. Even if this is intentional (and the memory is released elsewhere) it is still dangerous, because it subverts the Resource Acquisition is Initialization convention, and consequently users of the class might not release the memory at all. This check should only be used if all of a class' copy control functions are defined in the same file.

Coding standards

CWE 401

> Improper Release of Memory Before Removing Last Reference ('Memory Leak')

Code examples

The following code example fails the check and will give a warning:

```
class MyClass{
  int* p;

public:
  MyClass(){
    p = new int;
  }
};
```

The following code example passes the check and will not give a warning about this issue:

```
class MyClass{
  int* p;

 public:
  MyClass(){
    p = new int;
  }

  ~MyClass(){
    delete p;
  }
};
```

COP-init-order (C++ only)

Synopsis

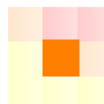
Data members are initialized with other data members that are in the same initialization list.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

Data members are initialized with other data members that are in the same initialization list. This can cause confusion, and might produce incorrect output, because data members are initialized in order of their declaration and not in the order of the initialization list.

Coding standards

CERT OOP37-CPP

Constructor initializers should be ordered correctly

CWE 456

Missing Initialization

Code examples

The following code example fails the check and will give a warning:

```
class C{
  int x;
  int y;
  C():
    x(5),
    y(x)  //Initializing using another member
  {}
};
```

The following code example passes the check and will not give a warning about this issue:

```
class C{
  int x;
  int y;
  C():
    x(5),
    y(5)  //OK
  {}
};
```

COP-init-uninit (C++ only)

Synopsis

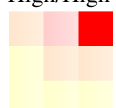
An initializer list reads the values of still uninitialized members.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

The expressions used to initialize a class member contain other class members, that have not yet been initialized themselves. The order in which they are initialized depends on the order of their declarations in the class definition and not on the order in which the members appear in the list, which might feel counter-intuitive. This might cause some of the object's attributes to have incorrect values, leading to logic errors or an application crash if the class handles dynamic memory.

Coding standards

CWE 456

Missing Initialization

Code examples

The following code example fails the check and will give a warning:

```
class C{
  int y;
  int x;
  C():
    x(5),
    y(x)  //x has not been initialized yet,
          //as y was defined first (line 2)
  {}
};
```

The following code example passes the check and will not give a warning about this issue:

```
class C{
  int x;
  int y;
  C():
    x(5),
    y(x)  //OK - x has been initialized
  {}
};
```

COP-member-uninit (C++ only)

Synopsis

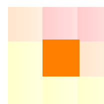
A member of a class is not initialized in one of the class constructors.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

A member of a class is not initialized in one of the class constructors. This might cause unexpected or unpredictable program behavior, and can be very difficult to identify as the cause. Because members of built-in types are not given a default initialization, constructors must initialize all members of a class. Even if this is intentional (and the attribute is initialized elsewhere) it is still dangerous, because it subverts the Resource Acquisition is Initialization convention, and consequently users of the class might not initialize the attribute. Uninitialized data can lead to incorrect program flow, and might cause the application to crash if the class handles dynamic memory.

Coding standards

CWE 456

Missing Initialization

Code examples

The following code example fails the check and will give a warning:

```
struct S{
  int x;
  S() {}  //this constructor should initialize x
};
```

The following code example passes the check and will not give a warning about this issue:

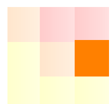
```
struct S{
  int x;

  S(){
    x = 1; //OK - x is initialized
  }
};
struct S{
  int x;
  S() : x(1) {} //OK - x is initialized
};
```

CPU-ctor-call-virt (C++ only)

Synopsis
A virtual member function is called in a class constructor.

Enabled by default
Yes

Severity/Certainty
Medium/High



Full description
When an instance is constructed, the virtual member function of its base class is called, rather than the function of the actual class being constructed. This might result in the incorrect function being called, and consequently incorrect data or uninitialized elements.

Coding standards
CERT OOP30-CPP

> Do not invoke virtual functions from constructors or destructors

MISRA C++ 2008 12-1-1

> (Required) An object's dynamic type shall not be used from the body of its constructor or destructor.

Code examples
The following code example fails the check and will give a warning:

```
#include <iostream>
#ifndef __embedded_cplusplus
  using namespace std;
#endif

class A {
public:
  A() { f(); }  //virtual member function is called
  virtual void f() const { cout << "A::f\n"; }
};

class B: public A {
public:
  virtual void f() const { cout << "B::f\n"; }
};

int main(void) {
  B *b = new B();
  delete b;
  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <iostream>
#ifndef __embedded_cplusplus
  using namespace std;
#endif

class A {
public:
  A() { }  //OK - contructor does not call any virtual
           //member functions
  virtual void f() const { cout << "A::f\n"; }
};

class B: public A {
public:
  virtual void f() const { cout << "B::f\n"; }
};

int main(void) {
  B *b = new B();
  delete b;
  return 0;
}
```

CPU-ctor-implicit (C++ only)

Synopsis

Constructors that are callable with a single argument of fundamental type are not declared `explicit`.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

Constructors that are callable with a single argument of fundamental type are not declared `explicit`. This means that nothing prevents the constructor from being used to implicitly convert from a fundamental type to the class type.

Coding standards

CERT OOP32-CPP

Ensure that single-argument constructors are marked "explicit"

MISRA C++ 2008 12-1-3

(Required) All constructors that are callable with a single argument of fundamental type shall be declared explicit.

Code examples

The following code example fails the check and will give a warning:

```
class C{
  C(double x){} //should be explicit
};
```

The following code example passes the check and will not give a warning about this issue:

```
class C{
  explicit C(double x){} //OK
};
```

CPU-delete-throw (C++ only)

Synopsis

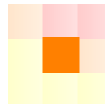
An exception is thrown, or might be thrown, in an overloaded `delete` or `delete[]` operator.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

An exception is thrown, or might be thrown, in an overloaded `delete` or `delete[]` operator. Because memory is often deallocated in a destructor, an exception that is thrown in a `delete` or `delete[]` operator is likely to be thrown during stack unwinding, which will cause the application to crash.

Coding standards

CERT ERR38-CPP

Deallocation functions must not throw exceptions

Code examples

The following code example fails the check and will give a warning:

```
class E{};

class C {
  void operator delete[ ](void* p) {
    if (!p){
      throw E();  //may throw an exception here
    }
  }
  int* p;
};
class E{};

void do_something();

class C {
  void operator delete[ ](void* p) throw (E) {  //may throw an 
exception
    if (!p){
      do_something();
    }
  }
  int* p;
};
```

The following code example passes the check and will not give a warning about this issue:

```
void do_something();

class C {
  void operator delete[ ](void* p) {   //OK
    if (!p){
      do_something();
    }
  }
  int* p;
};
```

CPU-delete-void (C++ only)

Synopsis

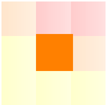
A pointer to void is used in `delete`, causing the destructor not to be called.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

A pointer to void is used in `delete`. When `delete` is called on a void pointer in C++, the object is deallocated from memory but its destructor is not called.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
void example(void *a) {
  delete a;
}
```

The following code example passes the check and will not give a warning about this issue:

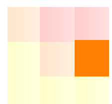
```
void example(int *a) {
  delete a;
}
```

CPU-dtor-call-virt (C++ only)

Synopsis
: A virtual member function is called in a class destructor.

Enabled by default
: Yes

Severity/Certainty
: Medium/High



Full description
: When an instance is destroyed, the virtual member function of its base class is called, rather than the function of the actual class being destroyed. This might result in the incorrect function being called, and consequently dynamic memory might not be properly deallocated, or some other unwanted behavior might occur.

Coding standards
: CERT OOP30-CPP

    Do not invoke virtual functions from constructors or destructors

  MISRA C++ 2008 12-1-1

    (Required) An object's dynamic type shall not be used from the body of its constructor or destructor.

Code examples
: The following code example fails the check and will give a warning:

```
#include <iostream>
#ifndef __embedded_cplusplus
  using namespace std;
#endif

class A {
public:
  ~A() { f(); }  //virtual member function is called
  virtual void f() const { cout << "A::f\n"; }
};

class B: public A {
public:
  virtual void f() const { cout << "B::f\n"; }
};

int main(void) {
  B *b = new B();
  delete b;
  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <iostream>
#ifndef __embedded_cplusplus
  using namespace std;
#endif

class A {
public:
  ~A() { }  //OK - contructor does not call any virtual
            //member functions
  virtual void f() const { cout << "A::f\n"; }
};

class B: public A {
public:
  virtual void f() const { cout << "B::f\n"; }
};

int main(void) {
  B *b = new B();
  delete b;
  return 0;
}
```

CPU-malloc-class (C++ only)

Synopsis

An allocation of a class instance with `malloc()` does not call a constructor.

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

When allocating memory for a class instance with `malloc()`, no class constructor is called. Using `malloc()` creates an uninitialized object. To initialize the object at allocation, use the `new` operator

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

class Foo {
 public:
  void setA(int val){
    a=val;
  }
 private:
   int a;
};

void main(){

  Foo *fooArray;

  //malloc of class Foo
  fooArray  = static_cast<Foo*>(malloc(5 * sizeof(Foo)));

  fooArray->setA(4);

}
```

The following code example passes the check and will not give a warning about this issue:

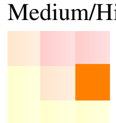
```
#include <stdlib.h>

void main(){

int *fooArray;
fooArray  = static_cast<int*>(malloc(5 * sizeof(int)));
*fooArray = 4;


}
```

CPU-nonvirt-dtor (C++ only)

Synopsis
A public non-virtual destructor is defined in a class with virtual methods.

Enabled by default
Yes

Severity/Certainty
Medium/High



Full description
A public non-virtual destructor is defined in a class with virtual methods. Calling `delete` on a pointer to any class derived from this one might call the wrong destructor. If any class might be a base class (by having virtual methods), then its destructor should be either be `virtual` or `protected` so that callers cannot destroy derived objects via pointers to the base.

Coding standards
CERT OOP34-CPP

Ensure the proper destructor is called for polymorphic objects

Code examples
The following code example fails the check and will give a warning:

```
#include <iostream>
#ifndef __embedded_cplusplus
  using namespace std;
#endif

class Base
{
public:
  Base() { cout<<"Constructor: Base"<<endl;}
  virtual void f(void) {}
  //non-virtual destructor:
  ~Base() { cout<<"Destructor : Base"<<endl;}
};

class Derived: public Base
{
public:
  Derived() { cout<<"Constructor: Derived"<<endl;}
  void f(void) { cout <<"Calling f()"; }
  virtual ~Derived() { cout<<"Destructor : Derived"<<endl;}
  };

int main(void)
{
  Base *Var = new Derived();
  delete Var;
  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <iostream>
#ifndef __embedded_cplusplus
  using namespace std;
#endif

class Base
{
public:
  Base() { cout<<"Constructor: Base"<<endl;}
  virtual void f(void) {}
  virtual ~Base() { cout<<"Destructor : Base"<<endl;}
};

class Derived: public Base
{
public:
  Derived() { cout<<"Constructor: Derived"<<endl;}
  void f(void) { cout <<"Calling f()"; }
  ~Derived() { cout<<"Destructor : Derived"<<endl;}
  };

int main(void)
{
  Base *Var = new Derived();
  delete Var;
  return 0;
}
```

CPU-return-ref-to-class-data (C++ only)

Synopsis

Member functions return non-`const` handles to members.

Enabled by default

Yes

Severity/Certainty

Medium/High

Full description

Member functions return non-`const` handles to members. Implement class interfaces with member functions to retain more control over how the object state can be modified and to make it easier to maintain a class without affecting clients. Returning a handle to class-data allows clients to modify the state of the object without using any interfaces.

Coding standards

CERT OOP35-CPP

Do not return references to private data

MISRA C++ 2008 9-3-2

(Required) Member functions shall not return non-const handles to class-data.

Code examples

The following code example fails the check and will give a warning:

```
class C{
  int x;
 public:
  int& foo();
  int* bar();
};

int& C::foo() {
  return x;  //returns a non-const reference to x
}

int* C::bar() {
  return &x;  //returns a non-const pointer to x
}
```

The following code example passes the check and will not give a warning about this issue:

```
class C{
  int x;
 public:
  const int& foo();
  const int* bar();
};

const int& C::foo() {
  return x;  //OK - returns a const reference
}

const int* C::bar() {
  return &x;  //OK - returns a const pointer
}
```

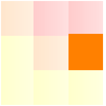
DECL-implicit-int

Synopsis

An object or function of the type `int` is declared or defined, but its type is not explicitly stated.

Enabled by default: No

Severity/Certainty: Medium/High



Full description: An object or function of the type `int` is declared or defined, but its type is not explicitly stated. The type of an object or function must be explicitly stated.

Coding standards:

CERT DCL31-C

Declare identifiers before using them

MISRA C:2004 8.2

(Required) Whenever an object or function is declared or defined, its type shall be explicitly stated.

MISRA C:2012 Rule-8.1

(Required) Types shall be explicitly specified

Code examples: The following code example fails the check and will give a warning:

```
void func(void)
{
    static y;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void func(void)
{
    int x;
}
```

DEFINE-hash-multiple

Synopsis: Multiple `#` or `##` operators in a macro definition.

Enabled by default: Yes

Severity/Certainty

Medium/Low

Full description

The order of evaluation associated with both the `#` and `##` preprocessor operators is unspecified. Avoid this problem by having only one occurrence of either operator in any single macro definition (i.e. one `#`, or one `##`, or neither).

Coding standards

MISRA C:2004 19.12

> (Required) There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.

MISRA C++ 2008 16-3-1

> (Required) There shall be at most one occurrence of the # or ## operators in a single macro definition.

Code examples

The following code example fails the check and will give a warning:

```
#defineD(x, y, z, yz)x ## y ## z/* Non-compliant */
#define C(x, y)# x ## y/* Non-compliant */
```

The following code example passes the check and will not give a warning about this issue:

```
#define A(x)#x/* Compliant */
#defineB(x, y)x ## y/* Compliant */
```

ENUM-bounds

Synopsis

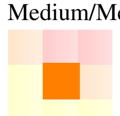
Conversions to `enum` that are out of range of the enumeration.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

There are conversions to `enum` that are out of range of the enumeration.

Coding standards

MISRA C++ 2008 7-2-1

(Required) An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.

Code examples

The following code example fails the check and will give a warning:

```
enum ens { ONE, TWO, THREE };

void example(void)
{
  ens one = (ens)10;
}
enum ens { ONE, TWO, THREE };

int func()
{
  return 10;
}

void example(void)
{
  ens one = (ens)func();
}
```

The following code example passes the check and will not give a warning about this issue:

```
enum ens { ONE, TWO, THREE };

int func()
{
  return 1;
}

void example(void)
{
  ens one = (ens)func();
}
enum ens { ONE, TWO, THREE };

void example(void)
{
  ens one = ONE;
  ens two = TWO;
  two = one;
}
```

EXP-cond-assign

Synopsis

An assignment might be mistakenly used as the condition for an `if`, `for`, `while`, or `do` statement.

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

An assignment might be mistakenly used as the condition for an `if`, `for`, `while`, or `do` statement. This condition will either always or never hold, depending on the value of the second operand. This was most likely intended to be a comparison, not an assignment. This might cause incorrect program flow, and possibly an infinite loop.

Coding standards

CERT EXP18-C

Do not perform assignments in selection statements

CERT EXP19-CPP

Do not perform assignments in conditional expressions

CWE 481

Assigning instead of Comparing

MISRA C:2012 Rule-13.4

(Advisory) The result of an assignment operator should not be used

Code examples

The following code example fails the check and will give a warning:

```
int example(void) {
  int x = 2;
  if (x = 3)
    return 1;
  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(void) {
  int x = 2;
  if (x == 3)
    return 1;
  return 0;
}
```

EXP-dangling-else

Synopsis

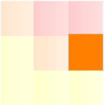
An `else` branch might be connected to an unexpected `if` statement.

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

An `else` branch might be connected to an unexpected `if` statement. An `else` branch is always connected with the closest possible `if` statement, but this might not always be the intention of the programmer. By explicitly putting braces around `if` statements where there might be ambiguity, you make the code more readable and your intentions clearer.

Coding standards

CWE 483

Incorrect Block Delimitation

Code examples

The following code example fails the check and will give a warning:

```
void foo(int x, int y){
  if (x < y)
    if (x == 1)
      ++y;
  else
    ++x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void foo(int x, int y){
  if (x < y){
    if (x == 1)
      ++y;
  }
  else
    ++x;
}
```

EXP-loop-exit

Synopsis

An unconditional `break`, `continue`, `return`, or `goto` within a loop.

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

There is an unconditional `break`, `goto`, `continue` or `return` in a loop. This means that some iterations of the loop will never be executed. This is most likely not the intended behavior.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int x = 1;
  int i;

  for (i = 0; i < 10; i++) {
    x = x + 1;
    break;  /* Unexpected loop exit */
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(int a) {
  int x = 1;
  int i;

  for (i = 0; i < 10; i++) {
    x = x + 1;
    if (x > a) {
      break;  /* loop exit is conditional */
    }
  }
}
```

EXP-main-ret-int

Synopsis

The return type of `main()` is not `int`.

Enabled by default

No

Severity/Certainty

Low/High

Full description

The return type of the `main` function is not `int`. The `main` function is expected to return an integer, so that the caller of the application can determine whether the application executed successfully or failed.

Coding standards

MISRA C:2004 1.2

(Required) No reliance shall be placed on undefined or unspecified behavior.

MISRA C:2012 Rule-1.3

(Required) There shall be no occurrence of undefined or critical unspecified behaviour

Code examples

The following code example fails the check and will give a warning:

```
void main() { };  //main does not return an int
```

The following code example passes the check and will not give a warning about this issue:

```
int main() {return 1;}  //OK - main returns an int
```

EXP-null-stmt

Synopsis

The body of an `if`, `while`, or `for` statement is a null statement.

Enabled by default

No

Severity/Certainty

Low/High

Full description

The body of an `if`, `while`, or `for` statement is a null statement. This might be intentional (a placeholder), but because a null statement as the body is difficult to find when debugging or reviewing code, it is good practice to use an empty block to identify a stub body. Note that if the condition expression of a `for` loop has possible side-effects, or if an `if` statement has a null body but carries an `else` clause, this check will not give a warning.

Coding standards

CERT EXP15-C

Do not place a semicolon on the same line as an if, for, or while statement

CWE 483

Incorrect Block Delimitation

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int i;
  for (i=0; i!=10; ++i);  //Null statement as the
                          //body of this for loop
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int i;
  for (i=0; i!=10; ++i){  //An empty block is much
  }                       //more readable
}
```

EXP-stray-semicolon

Synopsis

Stray semicolons on the same line as other code

Enabled by default

No

Severity/Certainty

Low/Low

Full description

There are stray semicolons on the same line as other code. Before preprocessing, a null statement should only be on a line by itself; it can be followed by a comment only if the first character following the null statement is a whitespace character.

Coding standards

CERT EXP15-C

Do not place a semicolon on the same line as an if, for, or while statement

MISRA C:2004 14.3

(Required) Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a whitespace character.

MISRA C++ 2008 6-2-3

(Required) Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int i;
  for (i=0; i!=10; ++i);  //Null statement as the
                          //body of this for loop
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int i;
  for (i=0; i!=10; ++i){  //An empty block is much
  }                       //more readable
}
```

EXPR-const-overflow

Synopsis

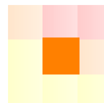
A constant unsigned integer expression overflows.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

A constant unsigned integer expression overflows.

Coding standards

MISRA C:2004 12.11

(Advisory) Evaluation of constant unsigned integer expressions should not lead to wrap-around.

MISRA C++ 2008 5-19-1

(Advisory) Evaluation of constant unsigned integer expressions should not lead to wrap-around.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  (0xFFFFFFFF + 1u);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  0x7FFFFFFF + 0;
}
```

FPT-cmp-null

Synopsis

The address of a function is compared with `NULL`.

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

The address of a function is compared with NULL. This is incorrect, because the address of a function is never NULL. If the intention was to call the function, but the parentheses were accidentally omitted, the application might behave unexpectedly because the address of the function is checked, not the return value. This means that the condition always holds, and any of the function's side-effects will not occur. If this was intentional, it is an unnecessary comparison, because a function address will never be NULL. If the function is declared but not defined, its address might fail to link if the function is called.

Coding standards

CWE 480

Use of Incorrect Operator

Code examples

The following code example fails the check and will give a warning:

```
int foo() {
    return 1;
}

int main(void) {
  if (foo == 0) {  /* foo, not foo() */
    return 1;
  }

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int foo() {
    return 0;
}

int main(void) {
  if (foo() == 0) {   /* foo() returns an int */
    return 1;
  }

  return 0;
}
```

FPT-literal

Synopsis

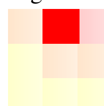
A function pointer that refers to a literal address is dereferenced.

Enabled by default

No

Severity/Certainty

High/Medium



Full description

A function pointer that refers to a literal address is dereferenced. A literal address is always invalid as a function pointer, and dereferencing it is an illegal memory access that might cause the application to crash.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

typedef void (*fn)(int);

void baz(int x){
  ++x;
}

void example(void) {
  fn bar = NULL;

  /* ... */

  bar(1); //ERROR
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

typedef void (*fn)(int);

void baz(int x){
  ++x;
}

void example(void) {
  fn bar = NULL;

  /* ... */

  bar = baz;
  bar(1);
}
```

FPT-misuse

Synopsis
A function pointer is used in an invalid context.

Enabled by default
Yes

Severity/Certainty
Low/High

Full description
A function pointer is used in an invalid context. It is an error to use a function pointer to do anything other than calling the function being pointed to, comparing the function pointer to another pointer using `!=` or `==`, passing the function pointer to a function, returning the function pointer from a function, or storing the function pointer in a data structure. Misusing a function pointer might result in erroneous behavior, and in junk data being interpreted as instructions and being executed as such.

Coding standards
CERT EXP16-C

Do not compare function pointers to constant values

CWE 480

Use of Incorrect Operator

Code examples
The following code example fails the check and will give a warning:

```
/* declare a function */
int foo(int x, int y){
  return x+y;
}

#pragma diag_suppress=Pa153

int foo2(int x, int y) {

  if (foo)
    return (foo)(x,y);

  if (foo < foo2)
    return (foo)(x,y);
return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

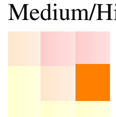
```
typedef int (*fptr)(int,int);

int f_add(int x, int y){
  return x+y;

}

int f_sub(int x, int y){
  return x-y;
}


int foo(int opcode, int x, int y){

  fptr farray[2];
  farray[0] = f_add;
  farray[1] = f_sub;

  return (farray[opcode])(x,y);

}

int foo2(fptr f1, fptr f2){

  if (f1 == f2)
    return 1;
  else
    return 0;

}
```

FUNC-implicit-decl

Synopsis
Functions are used without prototyping.

Enabled by default
No

Severity/Certainty
Medium/High



Full description
Functions are used without prototyping. Functions must be prototyped before use.

Coding standards

CERT DCL31-C

Declare identifiers before using them

MISRA C:2004 8.1

(Required) Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.

MISRA C:2012 Rule-17.3

(Mandatory) A function shall not be declared implicitly

Code examples

The following code example fails the check and will give a warning:

```
void func2(void)
{
    func();
}
```

The following code example passes the check and will not give a warning about this issue:

```
void func(void);
void func2(void)
{
    func();
}
```

FUNC-unprototyped-all

Synopsis

Functions are declared with an empty () parameter list that does not form a valid prototype.

Enabled by default

No

Severity/Certainty

Medium/High

Full description

Functions are declared with an empty () parameter list that does not form a valid prototype. Functions must be prototyped before use.

Coding standards

CERT DCL20-C

Always specify void even if a function accepts no arguments

MISRA C:2004 16.5

(Required) Functions with no parameters shall be declared and defined with the parameter list void.

MISRA C:2012 Rule-8.2

(Required) Function types shall be in prototype form with named parameters

Code examples

The following code example fails the check and will give a warning:

```
void func();/* not a valid prototype in C */
void func2(void)
{
    func();
}
```

The following code example passes the check and will not give a warning about this issue:

```
void func(void);
void func2(void)
{
    func();
}
```

FUNC-unprototyped-used

Synopsis

Arguments are passed to functions without a valid prototype.

Enabled by default

Yes

Severity/Certainty

Low/Low

Full description

Arguments are passed to functions without a valid prototype. This is permitted in C89, but it is unsafe because it bypasses all type checking.

Coding standards

CERT DCL20-C

Always specify void even if a function accepts no arguments

CERT DCL31-C

Declare identifiers before using them

Code examples

The following code example fails the check and will give a warning:

```
void func();/* not a valid prototype in C */
void func2(void)
{
    func(77);
    func(77.0);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void func(void);
void func2(void)
{
    func();
}
```

INCLUDE-c-file

Synopsis

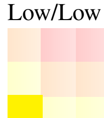
A .c file includes one or more .c files.

Enabled by default

No

Severity/Certainty

Low/Low



Full description

A C file includes one or more C files. C files shall not include other C files.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include "header.c"
void example(void) {}
```

The following code example passes the check and will not give a warning about this issue:

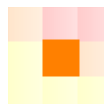
```
#include <stdlib.h>
void example(void) {}
```

INT-use-signed-as-unsigned-pos

Synopsis
A negative signed integer is implicitly cast to an unsigned integer.

Enabled by default
No

Severity/Certainty
Medium/Medium



Full description
A negative signed integer is implicitly cast to an unsigned integer. The result of this cast will be a large integer, and using this value might result in unexpected behavior.

Coding standards
This check does not correspond to any coding standard rules.

Code examples
The following code example fails the check and will give a warning:

```
void example(int c) {
  int a = 5;
  if (c) {
    a=-10;
  }
  unsigned int b = a;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(int c) {
  int a = 10;
  if (c) {
    a=5;
  }
  unsigned int b = a;
}
```

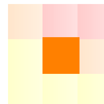
INT-use-signed-as-unsigned

Synopsis
A negative signed integer is implicitly cast to an unsigned integer.

Enabled by default
Yes

Severity/Certainty

Medium/Medium



Full description

A negative signed integer is implicitly cast to an unsigned integer. The result of this cast will be a large integer, and using this value might result in unexpected behavior.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int a = -10;
  unsigned int b = a;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int a = 10;
  unsigned int b = a;
}
```

ITR-end-cmp-aft (C++ only)

Synopsis

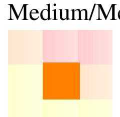
An iterator is used, then compared with `end()`

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

An iterator is used, then compared with `end()`. Using an iterator requires that it does not point to the end of a container. Subsequently comparing it with `end()` or `rend()` means that it might have been invalid at the point of dereference.

Coding standards

CERT ARR35-CPP

Do not allow loops to iterate beyond the end of an array or container

Code examples

The following code example fails the check and will give a warning:

```
#include <vector>
#include "iar.h"

int example(STD vector<int>& vec,
            STD vector<int>::iterator iter) {

  *iter = 4;  //line 9 asserts that iter may be
              //at the end of vec

  if (iter != vec.end()) {
    return 0;
  }
  return 1;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <vector>
#include "iar.h"

int example(STD vector<int>& vec,
            STD vector<int>::iterator iter) {

  if (iter != vec.end()) {
    *iter = 4;
  }

  if (iter != vec.end()) {
    return 0;
  }
  return 1;
}
```

ITR-end-cmp-bef (C++ only)

Synopsis

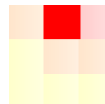
An iterator is compared with `end()` or `rend()`, then dereferenced.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

An iterator is compared with `end()` or `rend()`, then dereferenced. Although it is defined behavior for iterators to have a value of `end()` or `rend()`, dereferencing them at these values is undefined, and will most likely result in illegal memory access, creating a security vulnerability in the code. This error can occur if the programmer accidentally uses the wrong comparison operator, for example `==` instead of `!=`, or if the `then`- and `else`-clauses of an `if` statement have accidentally changed places.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <vector>
#include "iar.h"

int foo(){
  STD vector<int> a(5,6);
  STD vector<int>::iterator i;
  for (i = a.begin(); i != a.end(); ++i){
    ;
  }
  *i;  //here, i == a.end()
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <vector>
#include "iar.h"

int foo(){
  STD vector<int> a(5,6);
  STD vector<int>::iterator i;
  *i;
  for (i = a.begin(); i != a.end(); ++i){
    *i;  //OK - i will never be a.end()
  }
}
```

ITR-invalidated (C++ only)

Synopsis

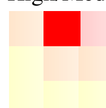
An iterator assigned to point into a container is used or dereferenced even though it might be invalidated.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

An iterator is assigned to point into a container, but later modifications to that container might have invalidated the iterator. The iterator is then used or dereferenced, which might be undefined behavior. Like pointers, iterators must point to a valid memory address to be used. When a container is modified by member functions such as `insert` or `erase`, some iterators might become invalidated and therefore risky to use. Any function that can remove elements, and some functions that add elements, might invalidate iterators. Iterators should be reassigned into a container after modifications are made and before they are used again, to ensure that they all point to a valid part of the container.

Coding standards

CERT ARR32-CPP

Do not use iterators invalidated by container modification

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 672

Operation on a Resource after Expiration or Release

Code examples

The following code example fails the check and will give a warning:

```
#include <vector>
#include "iar.h"

void example(){
  STD vector<int> a(5,6);
  STD vector<int>::iterator i;

  i = a.begin();
  while (i != a.end()){
    a.erase(i);
    ++i;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <vector>
#include "iar.h"

void example(){
  STD vector<int> a(5,6);
  STD vector<int>::iterator i;

  i = a.begin();
  while (i != a.end()){
    i = a.erase(a.begin());
  }
}
```

ITR-mismatch-alg (C++ only)

Synopsis
A pair of iterators passed to an STL algorithm function point to different containers.

Enabled by default
Yes

Severity/Certainty
High/Low

Full description
A pair of iterators passed to an STL algorithm function point to different containers. This can cause the application to access invalid memory, which might lead to a crash or a security vulnerability.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <vector>
#include <algorithm>
#include "iar.h"

void example(void) {

  #ifndef __embedded_cplusplus
    using namespace std;
  #endif

  vector<int> v, w;
  for (int i=0; i!= 10; ++i){
    v.push_back(random() % 100);
    w.push_back(random() % 100);
  }

  sort(v.begin(), w.end());  //v and w are different containers
}
#include <vector>
#include <algorithm>
#include "iar.h"

#define SIZE 10

void example(void) {
  int a[SIZE], b[SIZE];
  for (int i=0; i!= SIZE; ++i){
    a[i] = random() % 100;
    b[i] = random() % 100;
  }

  STD sort(a, b+SIZE);  //a and b are different arrays
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <vector>
#include <algorithm>
#include "iar.h"

void example(void) {
  STD vector<int> v;
  for (int i=0; i!= 10; ++i){
    v.push_back(random() % 100);
  }

  STD sort(v.begin(), v.end());  //OK
}
```

ITR-store (C++ only)

Synopsis

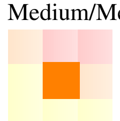
A container's `begin()` or `end()` iterator is stored and subsequently used.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

A container's `begin()` or `end()` iterator is stored and subsequently used. If the container is modified, these iterators will become invalidated. This could result in illegal memory access or a crash. Calling `begin()` and `end()` as these iterators are needed in loops and comparisons will ensure that only valid iterators are used.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <vector>
#include "iar.h"

void increment_all(STD vector<int>& v) {
  STD vector<int>::iterator b = v.begin();
  STD vector<int>::iterator e = v.end();
  //Storing these iterators is dangerous and unnecessary

  for (STD vector<int>::iterator i = b; i != e; ++i){
    ++(*i);
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <vector>
#include "iar.h"

void increment_all(STD vector<int>& v) {
  for (STD vector<int>::iterator i = v.begin();
       i != v.end(); ++i){
    ++(*i);  //OK
  }
}
```

ITR-uninit (C++ only)

Synopsis

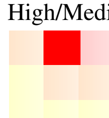
An iterator is dereferenced or incremented before it is assigned to point into a container.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

An iterator is dereferenced or incremented before it is assigned to point into a container. This will result in undefined behavior if the path that uses the uninitialized interator is executed, possibly causing illegal memory access or a crash.

Coding standards

CERT EXP33-C

Do not reference uninitialized memory

CWE 457

Use of Uninitialized Variable

Code examples

The following code example fails the check and will give a warning:

```
#include <map>
#include "iar.h"

void example(STD map<int, int>& m, bool maybe) {
  STD map<int, int>::iterator i;

  *i;  //i is uninitialized
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <map>
#include "iar.h"

void example(STD map<int, int>& m) {
  STD map<int, int>::iterator i;

  i=m.begin();  //i is initialized
  *i;
}
```

LIB-bsearch-overrun-pos

Synopsis

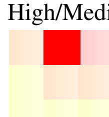
Arguments passed to `bsearch` might cause it to overrun.

Enabled by default

No

Severity/Certainty

High/Medium



Full description

A buffer overrun might be caused by a call to `bsearch`. This is because a buffer length being passed is greater than that of the buffer passed to either function as their first argument

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

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

int cmp(const void *a, const void *b) {
  return a == b;
}

void example(void) {
  int *a = malloc(sizeof(int) * 10);
  int *b = malloc(sizeof(int));
  bsearch(b, a, 20, sizeof(int), &cmp);
}
```

The following code example passes the check and will not give a warning about this issue:

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

int cmp(const void *a, const void *b) {
  return a == b;
}

void example(void) {
  int *a = malloc(sizeof(int) * 10);
  int *b = malloc(sizeof(int));
  bsearch(b, a, 10, sizeof(int), &cmp);
}
```

LIB-bsearch-overrun

Synopsis

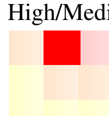
Arguments passed to `bsearch` cause it to overrun.

Enabled by default

No

Severity/Certainty

High/Medium



Full description

A buffer overrun is caused by a call to `bsearch`. This is because a buffer length being passed is greater than that of the buffer passed to either function as their first argument.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

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

int cmp(const void *a, const void *b) {
  return a == b;
}

void example(void) {
  int *a = malloc(sizeof(int) * 10);
  int *b = malloc(sizeof(int));
  bsearch(b, a, 20, sizeof(int), &cmp);
}
```

The following code example passes the check and will not give a warning about this issue:

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

int cmp(const void *a, const void *b) {
  return a == b;
}

void example(void) {
  int *a = malloc(sizeof(int) * 10);
  int *b = malloc(sizeof(int));
  bsearch(b, a, 10, sizeof(int), &cmp);
}
```

LIB-buf-size

Synopsis

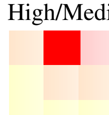
A call to a string function has a size argument larger than the size of the target buffer.

Enabled by default

No

Severity/Certainty

High/Medium



Full description

A call to a string function has a size argument larger than the size of the target buffer. This might indicate a buffer overflow or an illegal memory access, and might cause unexpected behavior or a crash. The target buffer must be able to store the number of elements as indicated by the size argument to the function. That is, the size argument must not be larger than the size of the destination buffer.

Coding standards

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

CWE 121

Stack-based Buffer Overflow

CWE 122

Heap-based Buffer Overflow

Code examples

There are no examples for this check.

LIB-fn-unsafe

Synopsis

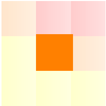
A potentially unsafe library function is used.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

A potentially unsafe library function is used, for which there is a safer alternative. This library function might create vulnerabilities like possible buffer overflow, because it does not check the size of a string before copying it into memory. The problem is that `strcpy()` and `gets()` functions are used. `strncpy()` should be used instead of `strcpy()`, and `fgets()` instead of `gets()`, because they include an additional argument in which the input's maximum allowed length is specified.

Coding standards

CWE 242

Use of Inherently Dangerous Function

CWE 252

Unchecked Return Value

CWE 394

Unexpected Status Code or Return Value

CWE 477

Use of Obsolete Functions

Code examples

The following code example fails the check and will give a warning:

```
#include <stdio.h>

void example(char* buf1) {
  scanf("%s", buf1);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdio.h>

void example(char* buf1, char* buf2) {
  strncpy(buf1, buf2, 5);
}
```

LIB-fread-overrun-pos

Synopsis

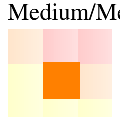
A call to `fread` might cause a buffer overrun.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

A call to `fread` might cause an overrun due to invalid arguments. `fread` takes an array as its first argument, the size of elements in the array as the second argument, and the number of elements in that array as the third. If `(size * count)` is greater than the allocated size of the array, an overrun will occur.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

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

void example(int b) {
  int *a = malloc(sizeof(int) * 10);
  int c;
  if (b) {
    c = 5;
  } else {
    c = 11;
  }
  fread(a, sizeof(int), c, NULL);
}
```

The following code example passes the check and will not give a warning about this issue:

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

void example(int b) {
  int *a = malloc(sizeof(int) * 10);
  int c;
  if (b) {
    c = 10;
  } else {
    c = 5;
  }
  fread(a, sizeof(int), c, NULL);
}
```

LIB-fread-overrun

Synopsis

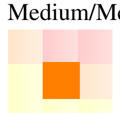
A call to `fread` causes a buffer overrun.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

A call to `fread` causes an overrun due to invalid arguments. `fread` takes an array as its first argument, the size of elements in the array as the second argument, and the number of elements in that array as the third. If `(size * count)` is greater than the allocated size of the array, an overrun will occur.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

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

void example(void) {
  int *a = malloc(sizeof(int) * 10);
  fread(a, sizeof(int), 11, NULL);
}
```

The following code example passes the check and will not give a warning about this issue:

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

void example(void) {
  int *a = malloc(sizeof(int) * 10);
  fread(a, sizeof(int), 10, NULL);
}
```

LIB-memchr-overrun-pos

Synopsis

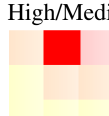
A call to `memchr` might cause a buffer overrun.

Enabled by default

No

Severity/Certainty

High/Medium



Full description

A call to `memchr` might cause a buffer overrun. If `memchr` is called with a size greater than the size of the allocated buffer, it will overrun and might cause a runtime error.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

void example(int b) {
  char *a = malloc(sizeof(char) * 20);
  int c;
  if (b) {
    c = 21;
  } else {
    c = 5;
  }
  memchr(a, 'a', c);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

void example(void) {
  char *a = malloc(sizeof(char) * 20);
  memchr(a, 'a', 10);
}
```

LIB-memchr-overrun

Synopsis

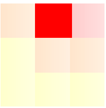
A call to `memchr` causes a buffer overrun.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

A call to `memchr` causes a buffer overrun. If `memchr` is called with a size greater than the size of the allocated buffer, it will overrun and might cause a runtime error.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

void example(void) {
  char *a = malloc(sizeof(char) * 20);
  memchr(a, 'a', 21);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

void example(void) {
  char *a = malloc(sizeof(char) * 20);
  memchr(a, 'a', 10);
}
```

LIB-memcpy-overrun-pos

Synopsis

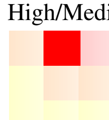
A call to `memcpy` might cause the memory to overrun.

Enabled by default

No

Severity/Certainty

High/Medium



Full description

A call to `memcpy` might cause the memory to overrun at either the destination or the source address.

Coding standards

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

CWE 121

Stack-based Buffer Overflow

CWE 122

Heap-based Buffer Overflow

CWE 124

Buffer Underwrite ('Buffer Underflow')

CWE 126

Buffer Over-read

CWE 127

Buffer Under-read

CWE 805

Buffer Access with Incorrect Length Value

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

void func(int b)
{
  int *p1;
  int *p2;
  if (b) {
    p1 = malloc(20);
    p2 = malloc(10);
  } else {
    p2 = malloc(20);
    p1 = malloc(10);
  }
  memcpy(p1, p2, 4);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

void func()
{
  int size = 10;
  int arr[size];
  int *ptr = malloc(size * sizeof(int));
  memcpy(ptr, arr, size);
}
```

LIB-memcpy-overrun

Synopsis

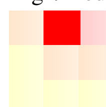
A call to `memcpy` or `memmove` causes the memory to overrun.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

A call to `memcpy` or `memmove` causes the memory to overrun at either the destination or the source address.

Coding standards

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

CWE 121

Stack-based Buffer Overflow

CWE 122

Heap-based Buffer Overflow

CWE 124

Buffer Underwrite ('Buffer Underflow')

CWE 126

Buffer Over-read

CWE 127

Buffer Under-read

CWE 805

Buffer Access with Incorrect Length Value

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

void func()
{
  int size = 10;
  int arr1[10];
  int arr2[11];
  memcpy(arr2, arr1, size + 1);
}
```

The following code example passes the check and will not give a warning about this issue:

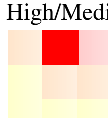
```
#include <stdlib.h>

void func()
{
  int size = 10;
  int arr[size];
  int *ptr = malloc(size * sizeof(int));
  memcpy(ptr, arr, size);
}
```

LIB-memset-overrun-pos

Synopsis
A call to `memset` might cause a buffer overrun.

Enabled by default
No

Severity/Certainty
High/Medium



Full description
A call to `memset` might cause a buffer overrun. If `memset` is called with a size greater than the size of the allocated buffer, it will overrun and might cause a runtime error.

Coding standards
This check does not correspond to any coding standard rules.

Code examples
The following code example fails the check and will give a warning:

```
#include <stdlib.h>

void example(int b) {
  char *a = malloc(sizeof(char) * 20);
  int c;
  if (b) {
    c = 21;
  } else {
    c = 5;
  }
  memset(a, 'a', c);
}
```

The following code example passes the check and will not give a warning about this issue:

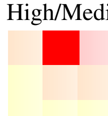
```
#include <stdlib.h>

void example(int b) {
  char *a = malloc(sizeof(char) * 20);
  int c;
  if (b) {
    c = 20;
  } else {
    c = 5;
  }
  memset(a, 'a', c);
}
```

LIB-memset-overrun

Synopsis A call to `memset` causes a buffer overrun.

Enabled by default Yes

Severity/Certainty High/Medium



Full description A call to `memset` causes a buffer overrun. If `memset` is called with a size greater than the size of the allocated buffer, it will overrun and might cause a runtime error.

Coding standards This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

void example(void) {
  char *a = malloc(sizeof(char) * 20);
  memset(a, 'a', 21);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

void example(void) {
  char *a = malloc(sizeof(char) * 20);
  memset(a, 'a', 10);
}
```

LIB-putenv

Synopsis

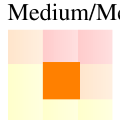
putenv used to set environment variable values.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

The POSIX function putenv() is used to set environment variable values. The putenv() function does not create a copy of the string supplied to it as an argument; instead it inserts a pointer to the string into the environment array. If a pointer to a buffer of automatic storage duration is supplied as an argument to putenv(), the memory allocated for that buffer might be overwritten when the containing function returns and stack memory is recycled.

Coding standards

CERT POS34-C

Do not call putenv() with a pointer to an automatic variable as the argument

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

int func(const char *var) {
  char env[1024];
  int retval = snprintf(env, sizeof(env),"TEST=%s", var);
  if (retval < 0 || (size_t)retval >= sizeof(env)) {
    /* Handle error */
  }

  return putenv(env);/* BUG: automatic storage is added to the
global environment */
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

int func(const char *var) {
  return setenv("TEST", var, 1);
}
```

LIB-qsort-overrun-pos

Synopsis

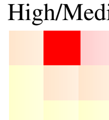
Arguments passed to `qsort` might cause it to overrun.

Enabled by default

No

Severity/Certainty

High/Medium



Full description

A buffer overrun might be caused by a call to `qsort`. This is because a buffer length being passed is greater than that of the buffer passed to either function as their first argument.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

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

int cmp(const void *a, const void *b) {
  return a == b;
}

void example(int b) {
  int *a = malloc(sizeof(int) * 10);
  int c;
  if (b) {
    c = 3;
  } else {
    c = 20;
  }
  qsort(a, c, sizeof(int), &cmp);
}
```

The following code example passes the check and will not give a warning about this issue:

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

int cmp(const void *a, const void *b) {
  return a == b;
}

void example(int b) {
  int *a = malloc(sizeof(int) * 10);
  int c;
  if (b) {
    c = 3;
  } else {
    c = 2;
  }
  qsort(a, c, sizeof(int), &cmp);
}
```

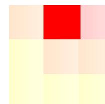
LIB-qsort-overrun

Synopsis Arguments passed to `qsort` cause it to overrun.

Enabled by default No

Severity/Certainty

High/Medium



Full description

A buffer overrun is caused by a call to `qsort`. This is because a buffer length being passed is greater than that of the buffer passed to either function as their first argument.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

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

int cmp(const void *a, const void *b) {
  return a == b;
}

void example(void) {
  int *a = malloc(sizeof(int) * 10);
  qsort(a, 11, sizeof(int), &cmp);
}
```

The following code example passes the check and will not give a warning about this issue:

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

int cmp(const void *a, const void *b) {
  return a == b;
}

void example(void) {
  int *a = malloc(sizeof(int) * 10);
  qsort(a, 3, sizeof(int), &cmp);
}
```

LIB-return-const

Synopsis

The return value of a `const` standard library function is not used.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

The return value of a `const` standard library function is not used. Because this function is defined as `const`, the call itself has no side effects; the only yield is the return value. If this return value is not used, the function call is redundant. These functions are inspected: `memchr()`, `strchr()`, `strpbrk()`, `strrchr()`, `strstr()`, `strtok()`, `gmtime()`, `getenv()`, and `bsearch()`. Discarding the return values of these functions is harmless but might indicate a misunderstanding of the application logic or purpose.

Coding standards

CERT EXP12-C

Do not ignore values returned by functions

CWE 252

Unchecked Return Value

CWE 394

Unexpected Status Code or Return Value

Code examples

The following code example fails the check and will give a warning:

```
#include <string.h>

void example(void) {
  strchr("Hello", 'h');  // No effect
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <string.h>

void example(void) {
  char* c = strchr("Hello", 'h');  //OK
}
```

LIB-return-error

Synopsis

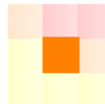
The return value for a library function that might return an error value is not used.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

The return value for a library function that might return an error value is not used. Because this function might fail, the programmer should inspect the return value to find any error values, to avoid a crash or unexpected behavior. These functions are isnpected: `malloc()`, `calloc()`, `realloc()`, and `mktime()`.

Coding standards

CWE 252

Unchecked Return Value

CWE 394

Unexpected Status Code or Return Value

MISRA C:2004 16.10

(Required) If a function returns error information, then that error information shall be tested.

MISRA C++ 2008 0-3-2

(Required) If a function generates error information, then that error information shall be tested.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  malloc(sizeof(int));  // This function could fail,
                        // and the return value is
                        // not checked
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

void example(void) {
  int *x = malloc(sizeof(int));  // OK - return value
                                 // is stored
}
```

LIB-return-leak

Synopsis

The return values from one or more library functions were not stored, returned, or passed as a parameter.

Enabled by default

Yes

Severity/Certainty

High/High

Full description

The return values from one or more library functions were not stored, returned, or passed as a parameter. If any of these functions return a pointer to newly allocated memory, and the return value is discarded, the memory is inaccessible and thus leaked. These functions are inspected: `malloc()`, `calloc()`, and `realloc()`.

Coding standards

CERT MEM31-C

Free dynamically allocated memory exactly once

CWE 252

Unchecked Return Value

CWE 394

Unexpected Status Code or Return Value

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

void example(void) {
  malloc(1);  //the return value of malloc is not
              // stored
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

void example(void) {
  int* x = malloc(1);  // OK - the return value of
                       // malloc is being stored in x
}
```

LIB-return-neg

Synopsis

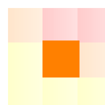
A variable assigned using a library function that can return -1 as an error value is subsequently used where the value must be non-negative.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

A variable assigned using a library function which can return -1 as an error value is subsequently used as a subscript or a size, both of which require the value to be non-negative. This might cause a crash or unpredictable behavior. These functions are inspected: `ftell()`, `clock()`, `time()`, `mktime()`, `fprintf()`, `printf()`, `sprintf()`, `vfprintf()`, `vprintf()`, `vsprintf()`, `mblen()`, `mbstowcs()`, `mbstowc()`, `wcstombs()`, and `wctomb()`.

Coding standards

CERT FIO04-C

Detect and handle input and output errors

CWE 252

Unchecked Return Value

CWE 394

Unexpected Status Code or Return Value

Code examples

The following code example fails the check and will give a warning:

```
#include <time.h>
#include <stdlib.h>

void example(void) {
  time_t time = clock();
  int *block = malloc(time); // time is used in a
                  // situation requiring it to be non-
                  // negative, but clock() may return -1
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <time.h>
#include <stdlib.h>

void example(void) {
  time_t time = clock();
  if (time>0){
    int *block = malloc(time); // OK - time is checked
  }
}
```

LIB-return-null

Synopsis

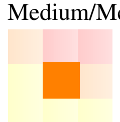
A pointer is assigned using a library function that can return `NULL` as an error value. This pointer is subsequently dereferenced without checking its value.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

A pointer is assigned using a library function that can return `NULL` as an error value. This pointer is subsequently dereferenced without checking its value, which might lead to a `NULL` dereference. Not inspecting the return value of any function returning a pointer before dereferencing it, might cause a crash. These functions are inspected: `malloc()`, `calloc()`, `realloc()`, `memchr()`, `strchr()`, `strpbrk()`, `strrchr()`, `strstr()`, `strtok()`, `gmtime()`, `getenv()`, and `bsearch()`.

Coding standards

CERT FIO04-C

Detect and handle input and output errors

CWE 252

Unchecked Return Value

CWE 394

Unexpected Status Code or Return Value

CWE 690

Unchecked Return Value to NULL Pointer Dereference

Code examples

The following code example fails the check and will give a warning:

```
#include <string.h>

void example(char c) {
  char* cp = strchr("Hello", c);
  printf("%c\n", *cp); // cp is dereferenced uncon-
                       // ditionally, but may be NULL
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <string.h>

void example(char c) {
  char* cp = strchr("Hello", c);
  if (cp){
    printf("%c\n", *cp); // OK - cp checked against
                         // NULL
  }
}
```

LIB-sprintf-overrun

Synopsis

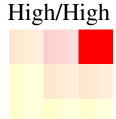
A call to `sprintf` causes a destination buffer overrun.

Enabled by default

No

Severity/Certainty

High/High



Full description

A call to the `sprintf` function causes a destination buffer overrun.

Coding standards

CERT STR31-C

Guarantee that storage for strings has sufficient space for character data and the null terminator

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

CWE 121

Stack-based Buffer Overflow

Code examples

The following code example fails the check and will give a warning:

```
#include <stdio.h>

char buf[5];

void example(void) {
  sprintf(buf, "Hello World!\n");
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdio.h>

char buf[14];

void example(void) {
  sprintf(buf, "Hello World!\n");
}
```

LIB-std-sort-overrun-pos (C++ only)

Synopsis

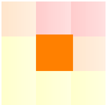
Using `std::sort` might cause buffer overrun.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

Using `std::sort` might cause a buffer overrun. `std::sort` can take a pointer to an array and a pointer to the end of the array as arguments, but if the pointer to the end of the array actually points beyond the end of the array being sorted, a buffer overrun might occur.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <algorithm>
#include "iar.h"

void example(void) {
  int a[10] = {0,1,2,3,4,5,6,7,8,9};
  STD sort(a, a+11);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <algorithm>
#include "iar.h"

void example(void) {
  int a[10] = {0,1,2,3,4,5,6,7,8,9};
  STD sort(a, a+5);
}
```

LIB-std-sort-overrun (C++ only)

Synopsis

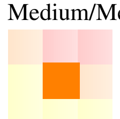
A buffer overrun is caused by use of `std::sort`.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

A buffer overrun is caused by use of `std::sort`. `std::sort` can take a pointer to an array and a pointer to the end of the array as arguments, but if the pointer to the end of the array actually points beyond the end of the array being sorted, a buffer overrun will occur.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <algorithm>
#include "iar.h"

void example(void) {
  int a[10] = {0,1,2,3,4,5,6,7,8,9};
  STD sort(a, a+11);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <algorithm>
#include "iar.h"

void example(void) {
  int a[10] = {0,1,2,3,4,5,6,7,8,9};
  STD sort(a, a+5);
}
```

LIB-strcat-overrun-pos

Synopsis

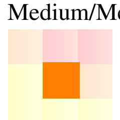
A call to `strcat` might cause destination buffer overrun.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

A call to the `strcat` function might cause a destination buffer overrun.

Coding standards

CERT STR31-C

Guarantee that storage for strings has sufficient space for character data and the null terminator

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

CWE 121

Stack-based Buffer Overflow

Code examples

The following code example fails the check and will give a warning:

```
#include <string.h>
#include <stdlib.h>

void example(void)
{
  char *str1 = "Hello World!\n";
  char *str2 = (char *)malloc(13);
  strcpy(str2,"");
  strcat(str2,str1);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <string.h>
#include <stdlib.h>

void example(void)
{
  char *str1 = "Hello World!\n";
  char *str2 = (char *)malloc(14);
  strcpy(str2, "");
  strcat(str2, str1);
}
```

LIB-strcat-overrun

Synopsis

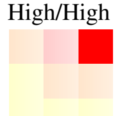
A call to `strcat` causes a destination buffer overrun.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

A call to the `strcat` function causes a destination buffer overrun.

Coding standards

CERT STR31-C

Guarantee that storage for strings has sufficient space for character data and the null terminator

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

CWE 121

Stack-based Buffer Overflow

Code examples

The following code example fails the check and will give a warning:

```
#include <string.h>
#include <stdlib.h>

void example(void)
{
  char *str1 = "Hello World!\n";
  char *str2 = (char *)malloc(13);
  strcpy(str2,"");
  strcat(str2,str1);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <string.h>
#include <stdlib.h>

void example(void)
{
  char *str1 = "Hello World!\n";
  char *str2 = (char *)malloc(14);
  strcpy(str2, "");
  strcat(str2, str1);
}
```

LIB-strcpy-overrun-pos

Synopsis

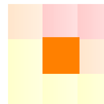
A call to `strcpy` might cause destination buffer overrun.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

A call to the `strcpy` function might cause a destination buffer overrun.

Coding standards

CERT STR31-C

Guarantee that storage for strings has sufficient space for character data and the null terminator

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

CWE 121

Stack-based Buffer Overflow

CWE 122

Heap-based Buffer Overflow

CWE 124

Buffer Underwrite ('Buffer Underflow')

CWE 126

Buffer Over-read

CWE 127

Buffer Under-read

Code examples

The following code example fails the check and will give a warning:

```
#include <string.h>
#include <stdlib.h>

void example(void)
{
  char *str1 = "Hello World!\n";
  char *str2 = (char *)malloc(13);
  strcpy(str2,str1);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <string.h>
#include <stdlib.h>

void example(void)
{
  char *str1 = "Hello World!\n";
  char *str2 = (char *)malloc(14);
  strcpy(str2,str1);
}
```

LIB-strcpy-overrun

Synopsis

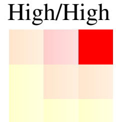
A call to `strcpy` causes a destination buffer overrun.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

A call to the `strcpy` function causes a destination buffer overrun.

Coding standards

CERT STR31-C

Guarantee that storage for strings has sufficient space for character data and the null terminator

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

CWE 121

Stack-based Buffer Overflow

CWE 122

Heap-based Buffer Overflow

CWE 124

Buffer Underwrite ('Buffer Underflow')

CWE 126

Buffer Over-read

CWE 127

Buffer Under-read

Code examples

The following code example fails the check and will give a warning:

```
#include <string.h>
#include <stdlib.h>

void example(void)
{
  char *str1 = "Hello World!\n";
  char *str2 = (char *)malloc(13);
  strcpy(str2,str1);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <string.h>
#include <stdlib.h>

void example(void)
{
  char *str1 = "Hello World!\n";
  char *str2 = (char *)malloc(14);
  strcpy(str2,str1);
}
```

LIB-strncat-overrun-pos

Synopsis

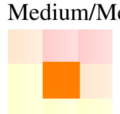
A call to `strncat` might cause a destination buffer overrun.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

Calling `strncat` with a destination buffer that is too small will cause a buffer overrun. `strncat` takes a destination buffer as its first argument. If the remaining space of this buffer is smaller than the number of characters to append, as determined by the position of the null terminator in the source buffer or the size passed as the third argument to `strncat`, an overflow might occur resulting in undefined behavior and runtime errors.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <string.h>
#include <stdlib.h>

void example(int d) {
  char * a = malloc(sizeof(char) * 5);
  char * b = malloc(sizeof(char) * 100);
  int c;
  if (d) {
    c = 10;
  } else {
    c = 5;
  }
  strcpy(a, "0123");
  strcpy(b, "45678901234");
  strncat(a, b, c);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <string.h>
#include <stdlib.h>

void example(int d) {
  char * a = malloc(sizeof(char) * 5);
  char * b = malloc(sizeof(char) * 100);
  int c;
  if (d) {
    c = 2;
  } else {
    c = 3;
  }
  strcpy(a, "0123");
  strcpy(b, "45678901234");
  strncat(b, a, c);
}
```

LIB-strncat-overrun

Synopsis

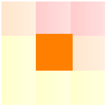
A call to `strncat` causes a destination buffer overrun.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

Calling `strncat` with a destination buffer that is too small will cause a buffer overrun. `strncat` takes a destination buffer as its first argument. If the remaining space of this buffer is smaller than the number of characters to append, as determined by the position of the null terminator in the source buffer or the size passed as the third argument to `strncat`, an overflow might occur resulting in undefined behavior and runtime errors.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <string.h>
#include <stdlib.h>

void example(void) {
  char * a = malloc(sizeof(char)*9);
  strcpy(a, "hello");
  strncat(a, "world", 4);
}
#include <string.h>
#include <stdlib.h>

void example(void) {
  char * a = malloc(sizeof(char)*9);
  strcpy(a, "hello");
  strncat(a, "world", 6);
}
```

The following code example passes the check and will not give a warning about this issue:

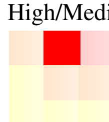
```
#include <string.h>
#include <stdlib.h>

void example(void) {
  char * a = malloc(sizeof(char)*11);
  strcpy(a, "hello");
  strncat(a, "world", 6);
}
#include <string.h>
#include <stdlib.h>

void example(void) {
  char * a = malloc(sizeof(char)*11);
  strcpy(a, "hello");
  strncat(a, "world", 4);
}
```

LIB-strncmp-overrun-pos

Synopsis
A call to `strncmp` might cause a buffer overrun.

Enabled by default
No

Severity/Certainty
High/Medium



Full description
An incorrect string length passed to `strncmp` might cause a buffer overrun. `strncmp` limits the number of characters it compares to the number passed as its third argument, to prevent buffer overruns with non-null-terminated strings. However, if a number is passed that is larger than the length of the two strings, and neither string is null-terminated, it will overrun.

Coding standards
This check does not correspond to any coding standard rules.

Code examples
The following code example fails the check and will give a warning:

```
#include <stdlib.h>
#include <string.h>

void example(int d) {
  char *a = malloc(sizeof(char) * 10);
  char *b = malloc(sizeof(char) * 10);
  int c;
  if (d) {
    c = 20;
  } else {
    c = 5;
  }
  strncmp(a, b, c);
}
```

The following code example passes the check and will not give a warning about this issue:

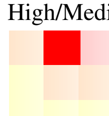
```
#include <stdlib.h>
#include <string.h>

void example(int d) {
  char *a = malloc(sizeof(char) * 10);
  char *b = malloc(sizeof(char) * 10);
  int c;
  if (d) {
    c = 8;
  } else {
    c = 5;
  }
  strncmp(a, b, c);
}
```

LIB-strncmp-overrun

Synopsis: A buffer overrun is caused by a call to `strncmp`.

Enabled by default: Yes

Severity/Certainty: High/Medium



Full description

A buffer overrun is caused by passing an incorrect string length to `strncmp`. `strncmp` limits the number of characters it compares to the number passed as its third argument, to prevent buffer overruns with non-null-terminated strings. However, if a number is passed that is larger than the length of the two strings, and neither string is null-terminated, it will overrun.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>
#include <string.h>

void example(void) {
  char *a = malloc(sizeof(char) * 10);
  char *b = malloc(sizeof(char) * 10);
  strncmp(a, b, 20);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>
#include <string.h>

void example(void) {
  char *a = malloc(sizeof(char) * 10);
  char *b = malloc(sizeof(char) * 10);
  strncmp(a, b, 5);
}
```

LIB-strncpy-overrun-pos

Synopsis

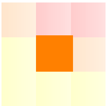
A call to `strncpy` might cause a destination buffer overrun.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

A call to `strncpy` might cause a destination buffer overrun.

Coding standards

CERT STR31-C

Guarantee that storage for strings has sufficient space for character data and the null terminator

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

CWE 121

Stack-based Buffer Overflow

CWE 122

Heap-based Buffer Overflow

CWE 124

Buffer Underwrite ('Buffer Underflow')

CWE 126

Buffer Over-read

CWE 127

Buffer Under-read

CWE 805

Buffer Access with Incorrect Length Value

Code examples

The following code example fails the check and will give a warning:

```
#include <string.h>
#include <stdlib.h>

void example(void)
{
  char *str1 = "Hello World!\n";
  char *str2 = (char *)malloc(13);
  strncpy(str2,str1,14);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <string.h>
#include <stdlib.h>

void example(void)
{
  char *str1 = "Hello World!\n";
  char *str2 = (char *)malloc(14);
  strncpy(str2, str1, 14);
}
```

LIB-strncpy-overrun

Synopsis

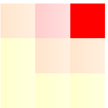
A call to `strncpy` causes a destination buffer overrun.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

A call to `strncpy` causes a destination buffer overrun.

Coding standards

CERT STR31-C

Guarantee that storage for strings has sufficient space for character data and the null terminator

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

CWE 121

Stack-based Buffer Overflow

CWE 122

Heap-based Buffer Overflow

CWE 124

Buffer Underwrite ('Buffer Underflow')

CWE 126

Buffer Over-read

CWE 127

Buffer Under-read

CWE 805

Buffer Access with Incorrect Length Value

Code examples

The following code example fails the check and will give a warning:

```
#include <string.h>
#include <stdlib.h>

void example(void)
{
  char *str1 = "Hello World!\n";
  char *str2 = (char *)malloc(13);
  strncpy(str2,str1,14);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <string.h>
#include <stdlib.h>

void example(void)
{
  char *str1 = "Hello World!\n";
  char *str2 = (char *)malloc(14);
  strncpy(str2, str1, 14);
}
```

LOGIC-overload (C++ only)

Synopsis

Overloaded && and || operators

Enabled by default

No

Severity/Certainty

Low/Low

Full description

There are overloaded versions of the comma and logical conjunction operators with the semantics of function calls, whose sequence point and ordering semantics are different from those of the built- in versions. It might not be clear at the point of use that these operators are overloaded, and which semantics that apply.

Coding standards

MISRA C++ 2008 5-2-11

(Required) The comma operator, && operator and the || operator shall not be overloaded.

Code examples

The following code example fails the check and will give a warning:

```
class C{
  bool x;
  bool operator||(bool other);
};

bool C::operator||(bool other){
  return x || other;
}
```

The following code example passes the check and will not give a warning about this issue:

```
class C{
  int x;
  int operator+(int other);
};

int C::operator+(int other){
  return x + other;
}
```

MEM-delete-array-op (C++ only)

Synopsis

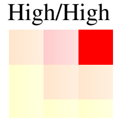
A memory location allocated with `new` is deleted with `delete[]`

Enabled by default

Yes

Severity/Certainty

High/High



Full description

A memory location is allocated with the `new` operator but deleted with the `delete []` operator. Use the `delete` operator instead.

Coding standards

CWE 762

Mismatched Memory Management Routines

Code examples

The following code example fails the check and will give a warning:

```
int main(void)
{
  int *p = new int;
  delete[] p; //should be delete, not delete[]

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int main(void)
{
  int *p = new int;
  delete p;

  return 0;
}
```

MEM-delete-op (C++ only)

Synopsis

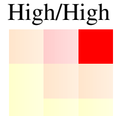
A memory location allocated with `new []` is deleted with `delete` or `free`.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

A memory location allocated with the `new []` operator is deleted with the `delete` operator. Use the `delete []` operator instead. The consequence of using `delete` is that only the array element directly pointed to will be deallocated, as if it were allocated with the singular `new` operator. This will most likely cause a memory leak. If `free` is used the resulting behavior will be undefined, because there is no guarantee that `new` invokes `malloc`.

Coding standards

CWE 762

Mismatched Memory Management Routines

Code examples

The following code example fails the check and will give a warning:

```
int main(void)
{
  int *p = new int[10];
  delete p; //should be delete[]

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int main(void)
{
  int *p = new int[10];
  delete [] p;

  return 0;
}
```

MEM-double-free-alias

Synopsis

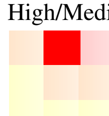
Freeing a memory location more than once.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

An attempt is made to free a memory location after it has already been freed. This will most likely cause an application crash. Unlike MEM-double-free, MEM-double-free-alias examines the location that pointers point to instead of the pointers themselves. You might see reports for code that looks like this (example of a linked list where each node has a pointer to an element, `elem`): for (; list != NULL; list

= list->next) {   free(list->elem); } The warning is issued because there is no guarantee that each list node's `elem` field is the same.

Coding standards

CERT MEM31-C

Free dynamically allocated memory exactly once

CWE 415

Double Free

MISRA C:2012 Rule-22.2

(Mandatory) A block of memory shall only be freed if it was allocated by means of a Standard Library function

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>
void f(int *p) {
  free(p);
  if(p) free(p);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

void example(void)
{
  int *p=malloc(4);
  free(p);
}
```

MEM-double-free-some

Synopsis

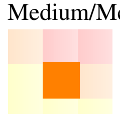
A memory location is freed more than once on some paths but not on others.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

There is a path through the code where a memory location is attempted to be freed after it has already been freed earlier. This will most likely cause an application crash on this path.

Coding standards

CERT MEM31-C

Free dynamically allocated memory exactly once

CWE 415

Double Free

MISRA C:2012 Rule-22.2

(Mandatory) A block of memory shall only be freed if it was allocated by means of a Standard Library function

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>
void example(void) {
    int *ptr = (int*)malloc(sizeof(int));
    free(ptr);
    if(rand() % 2 == 0)
    {
      free(ptr);
    }
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>
void example(void) {
    int *ptr = (int*)malloc(sizeof(int));
    if(rand() % 2 == 0)
    {
      free(ptr);
    }
    else
    {
      free(ptr);
    }
}
```

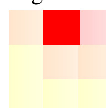
MEM-double-free

Synopsis

Freeing a memory location more than once.

Enabled by default Yes

Severity/Certainty High/Medium



Full description An attempt is made to free a memory location after it has already been freed. This will most likely cause an application crash.

Coding standards CERT MEM31-C

Free dynamically allocated memory exactly once

CWE 415

Double Free

MISRA C:2012 Rule-22.2

(Mandatory) A block of memory shall only be freed if it was allocated by means of a Standard Library function

Code examples The following code example fails the check and will give a warning:

```
#include <stdlib.h>
void f(int *p) {
  free(p);
  if(p) free(p);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

void example(void)
{
  int *p=malloc(4);
  free(p);
}
```

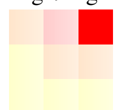
MEM-free-field

Synopsis A struct or a class field is possibly freed.

Enabled by default Yes

Severity/Certainty

High/High



Full description

A struct or a class field is possibly freed. Fields are located in the middle of memory objects and thus cannot be freed. Additionally, erroneously using `free()` on fields might corrupt `stdlib`'s memory bookkeeping, affecting heap memory.

Coding standards

CERT MEM34-C

> Only free memory allocated dynamically

CWE 590

> Free of Memory not on the Heap

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

struct C{
   int x;
};

int foo(struct C c) {
   int *p = &c.x;
   free(p);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

struct C{
   int *x;
};

int foo(struct C *c) {
   int *p = (c->x);
   free(p);
}
```

MEM-free-fptr

Synopsis

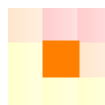
A function pointer is deallocated.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

A function pointer is deallocated. Function pointers are not dynamically allocated, and should thus not be deallocated. Freeing a function pointer will result in undefined behavior.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

int id(int a) {
  return a;
}

void example(void) {
  int (*f)(int);
  f = &id;
  free((void *)f);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

int id(int a) {
  return a;
}

void example(void) {
  int (*f)(int);
  f = &id;
}
```

MEM-free-no-alloc-struct

Synopsis

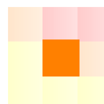
A struct field is deallocated without first having been allocated.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

A struct field is deallocated without first having been allocated. This might cause a runtime error.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

struct test {
  int *a;
};

void example(void) {
  struct test t;
  free(t.a);
}
```

The following code example passes the check and will not give a warning about this issue:

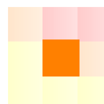
```
#include <stdlib.h>

struct test {
  int *a;
};

void example(void) {
  struct test t;
  t.a = malloc(sizeof(int));
  free(t.a);
}
```

MEM-free-no-alloc

Synopsis
A pointer is freed without having been allocated.

Enabled by default
Yes

Severity/Certainty
Medium/Medium



Full description
A pointer is freed without having been allocated.

Coding standards
This check does not correspond to any coding standard rules.

Code examples
The following code example fails the check and will give a warning:

```
#include <stdlib.h>

void example(void) {
  int *p;
  // Do stuff
  free(p);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

void example(void) {
  int *p = malloc(sizeof(int));
  // Do something
  free(p);
}
```

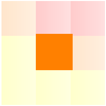
MEM-free-no-use

Synopsis
Memory is allocated and then freed without being used.

Enabled by default
Yes

Severity/Certainty

Medium/Medium



Full description

Memory is allocated and then freed without being used. This is probably unintentional and might indicate a copy-paste error.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>
void example(void) {
  int *p = malloc(sizeof(int));
  free(p);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>
void example(void) {
  int *p = malloc(sizeof(int));
  *p = 1;
  free(p);
}
```

MEM-free-op

Synopsis

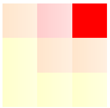
Memory allocated with `malloc` deallocated using `delete`.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

Memory allocated with `malloc()` or `calloc()` is deallocated using one of the `delete` operators instead of `free()`. This might cause a memory leak, or affect other heap memory due to corruption of `stdlib`'s memory bookkeeping.

Coding standards

CWE 404

Improper Resource Shutdown or Release

CWE 762

Mismatched Memory Management Routines

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>
void f()
{
  void *p = malloc(200);
  delete p;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>
void f() {
  void *p = malloc(200);
  free(p);
}
```

MEM-free-struct-field

Synopsis

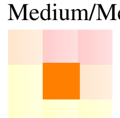
A struct's field is deallocated, but is not dynamically allocated.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

A struct's field is deallocated, but is not dynamically allocated. Regardless of whether a struct is allocated on the stack or on the heap, all non-dynamically allocated fields will be deallocated when the struct itself is deallocated (either through going out of scope or calling a function like `free()`). Explicitly freeing such fields might cause a crash, or corrupt surrounding memory. Incorrect use of `free()` might also corrupt stdlib's memory bookkeeping, affecting heap memory allocation.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

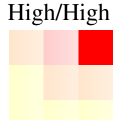
The following code example fails the check and will give a warning:

```
#include <stdlib.h>

struct test {
  int a;
};

void example(void) {
  struct test *t;
  free((void *)t->a);
}
#include <stdlib.h>

struct test {
  int a[10];
};

void example(void) {
  struct test t;
  free(t.a);
}
#include <stdlib.h>

struct test {
  int a;
};

void example(void) {
  struct test t;
  free((void *)t.a);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

struct test {
  int *a;
};

void example(void) {
  struct test *t;
  free(t->a);
}
#include <stdlib.h>

struct test {
  int *a;
};

void example(void) {
  struct test t;
  free(t.a);
}
```

MEM-free-variable-alias

Synopsis	A stack address might be freed.
Enabled by default	Yes
Severity/Certainty	High/High 
Full description	A stack address might be freed. Stack variables are automatically deallocated when they go out of scope. Consequently, explicitly freeing them might cause a crash or corrupt the surrounding stack data. Erroneously using `free()` on stack memory might also corrupt `stdlib`'s memory bookkeeping, affecting heap memory.
Coding standards	CERT MEM34-C Only free memory allocated dynamically CWE 590 Free of Memory not on the Heap

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>
void example(void){
  int x=0;
  free(&x);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int *p;
  p = (int *)malloc(sizeof( int));
  free(p);
}
```

MEM-free-variable

Synopsis

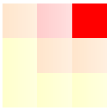
A stack address might be freed.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

A stack address might be freed. Stack variables are automatically deallocated when they go out of scope. Consequently, explicitly freeing them might cause a crash or corrupt the surrounding stack data. Erroneously using `free()` on stack memory might also corrupt `stdlib`'s memory bookkeeping, affecting heap memory.

Coding standards

CERT MEM34-C

Only free memory allocated dynamically

CWE 590

Free of Memory not on the Heap

MISRA C:2012 Rule-22.2

(Mandatory) A block of memory shall only be freed if it was allocated by means of a Standard Library function

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>
void example(void){
  int x=0;
  free(&x);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int *p;
  p = (int *)malloc(sizeof( int));
  free(p);
}
```

MEM-leak-alias

Synopsis

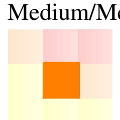
Incorrect deallocation causes memory leak.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

Memory is allocated, but then the pointer value is lost due to reassignment or its scope ending, without a guarantee of the value being propagated or the memory being freed. There must be no possible execution path during which the value is not freed, returned, or passed into another function as an argument, before it is lost. This is a memory leak. Note: If alias analysis is disabled, you must enable the non-alias version of this check, MEM-leak.

Coding standards

CERT MEM31-C

Free dynamically allocated memory exactly once

CWE 401

Improper Release of Memory Before Removing Last Reference ('Memory Leak')

CWE 772

Missing Release of Resource after Effective Lifetime

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

extern int rand();

void example(void) {
  int *ptr = malloc(sizeof(int));

  if (rand()){

    //losing reference to memory allocated
    //from the first malloc
    ptr = malloc(sizeof(int));
  }

  free(ptr);
}
#include <stdlib.h>

int main(void) {
    int *ptr = (int*)malloc(sizeof (int));
    if (rand() < 5) {
        free(ptr); // Not free() on all paths.
    }
    return 0;
}
#include <stdlib.h>

int main(void) {
  int *ptr = (int *)malloc(sizeof(int));

  ptr = NULL; //losing reference to the allocated memory

  free(ptr);

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

extern int rand();

void example(void) {
  int *ptr = malloc(sizeof(int));
  free(ptr);
}
```

MEM-leak

Synopsis

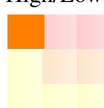
Incorrect deallocation causes memory leak.

Enabled by default

No

Severity/Certainty

High/Low



Full description

Memory is allocated, but then the pointer value is lost due to reassignment or its scope ending, without a guarantee of the value being propagated or the memory being freed. There must be no possible execution path during which the value is not freed, returned, or passed into another function as an argument, before it is lost. This is a memory leak.

Coding standards

CERT MEM31-C

Free dynamically allocated memory exactly once

CWE 401

Improper Release of Memory Before Removing Last Reference ('Memory Leak')

CWE 772

Missing Release of Resource after Effective Lifetime

MISRA C:2012 Rule-22.1

(Required) All resources obtained dynamically by means of Standard Library functions shall be explicitly released

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

extern int rand();

void example(void) {
  int *ptr = malloc(sizeof(int));

  if (rand()){

    //losing reference to memory allocated
    //from the first malloc
    ptr = malloc(sizeof(int));
  }

  free(ptr);
}
#include <stdlib.h>

int main(void) {
    int *ptr = (int*)malloc(sizeof (int));
    if (rand() < 5) {
        free(ptr); // Not free() on all paths.
    }
    return 0;
}
#include <stdlib.h>

int main(void) {
  int *ptr = (int *)malloc(sizeof(int));

  ptr = NULL; //losing reference to the allocated memory

  free(ptr);

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

int main(void) {
    int *ptr = (int*)malloc(sizeof(int));
    if (rand() < 5) {
        free(ptr);
    } else {
        free(ptr);
    }
    return 0;
}
#include <stdlib.h>

extern int rand();

void example(void) {
  int *ptr = malloc(sizeof(int));
  free(ptr);
}
```

MEM-malloc-arith

Synopsis

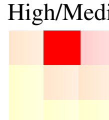
An assignment contains both a `malloc()` and pointer arithmetic on the right-hand side.

Enabled by default

No

Severity/Certainty

High/Medium



Full description

An assignment contains both a `malloc()` and pointer arithmetic on the right-hand side. If this is unintentional, the start of the allocated memory block might be lost, and a buffer overflow is possible.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

int example(void) {
  int *p;

  p = (int *)malloc(255) + 10;  //pointer arithmetic

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

int example(void) {
  int *p;

  p = (int *)malloc(255);

  return 0;
}
```

MEM-malloc-diff-type

Synopsis

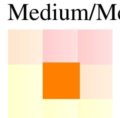
A call to `malloc` tries to allocate memory based on a `sizeof` operator, but the destination type of the call is of a different type.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

This might be an error, and will result in an allocated memory chunk that does not match the destination pointer or array. This might easily result in an invalid memory dereference, and crash the application.

Coding standards

CERT MEM35-C

Allocate sufficient memory for an object

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

int* foo(){
  return malloc(sizeof(char)*10);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

char* foo(){
  return malloc(sizeof(char)*10);
}
```

MEM-malloc-sizeof-ptr

Synopsis

`malloc(sizeof(p))`, where `p` is a pointer type, is assigned to a non-pointer variable.

Enabled by default

Yes

Severity/Certainty

High/Low

Full description

The argument given to `malloc()` is the size of a pointer, but the use of the return address does not suggest a double-indirection pointer. Allocating memory to an `int*`, for example, should use `sizeof(int)` rather than `sizeof(int*)`. Otherwise, the memory allocated might be smaller than expected, potentially leading to an application crash or corruption of other heap memory.

Coding standards

CERT EXP01-C

Do not take the size of a pointer to determine the size of the pointed-to type

CERT ARR01-C

Do not apply the sizeof operator to a pointer when taking the size of an array

CWE 467

Use of sizeof() on a Pointer Type

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>
void example(void) {
  int *p = (int*)malloc(sizeof(p)); //sizeof pointer
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>
void example(void) {
  int *p = (int*)malloc(sizeof(*p));
}
```

MEM-malloc-sizeof

Synopsis

Allocating memory with `malloc` without using `sizeof`.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

Memory was allocated with `malloc()` but the `sizeof` operator might not have been used. Using `sizeof` when allocating memory avoids any machine variations in the sizes of data types, and consequently avoids under-allocating. To pass this check, assign the address of the allocated memory to a `char` pointer, because `sizeof(char)` always returns 1.

Coding standards

CERT MEM35-C

Allocate sufficient memory for an object

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

void example(void) {
  int *x = malloc(4);  //no sizeof in malloc call
  free(x);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

void example(void) {
  int *x = malloc(sizeof(int));
  free(x);
}
```

MEM-malloc-strlen

Synopsis

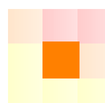
Dangerous arithmetic with `strlen` in argument to `malloc`.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

Dangerous arithmetic with `strlen` in an argument to `malloc`. It is usual to allocate a new string using `malloc(strlen(s)+1)`, to allow for the null terminator. However, it is easy to type `malloc(strlen(s+1))` by mistake, leading to `strlen` returning a length one less than the length of `s`, or if `s` is empty, exhibit undefined behavior.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>
#include <string.h>

void example(char *s) {
  char *a = malloc(strlen(s+1));
}
```

The following code example passes the check and will not give a warning about this issue:

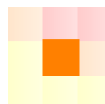
```
#include <stdlib.h>
#include <string.h>

void example(char *s) {
  char *a = malloc(strlen(s)+1);
}
```

MEM-realloc-diff-type

Synopsis
The variable that stores the result of `realloc` does not match the type of the first argument.

Enabled by default
Yes

Severity/Certainty
Medium/Medium



Full description
The variable that stores the result of `realloc` does not match the type of the first argument. Subsequent accesses to this memory might be misaligned and cause a runtime error.

Coding standards
This check does not correspond to any coding standard rules.

Code examples
The following code example fails the check and will give a warning:

```
#include <stdlib.h>

void example(int *a, int new_size) {
  unsigned int *b;
  b = realloc(a, sizeof(int) * new_size);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

void example(int *a, int new_size) {
  int *b;
  b = realloc(a, sizeof(int) * new_size);
}
```

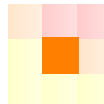
MEM-return-free

Synopsis
A function deallocates memory, then returns a pointer to that memory.

Enabled by default
Yes

Severity/Certainty

Medium/Medium



Full description

A function deallocates memory, then returns a pointer to that memory. If the callee of this function attempts to dereference the returned pointer, this will cause a runtime error.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

int *example(void) {
  int *a = malloc(sizeof(int));
  free(a);
  return a;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

int *example(void) {
  int *a = malloc(sizeof(int));
  return a;
}
```

MEM-return-no-assign

Synopsis

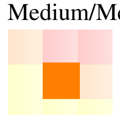
A function that allocates memory's return value is not stored.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

A function that allocates a memory's return value is not stored. Not storing the returned memory means that this memory cannot be tracked, and therefore deallocated. This will result in a memory leak.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

int *allocating_fn(void) {
  return malloc(sizeof(int));
}

void example(void) {
  allocating_fn();
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

int *allocating_fn(void) {
  return malloc(sizeof(int));
}

void example(void) {
  int *p = allocating_fn();
}
```

MEM-stack-alias

Synopsis

Might return address on the stack.

Enabled by default

Yes

Severity/Certainty

High/High

Full description

A local variable is defined in stack memory, then its address is potentially returned from the function. When the function exits, its stackframe will be considered illegal memory, and thus the address returned might be dangerous. This code and subsequent memory accesses might appear to work, but the operations are illegal and an application crash, or memory corruption, is very likely. To correct this problem, consider returning a copy of the object, using a global variable, or dynamically allocating memory.

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 562

Return of Stack Variable Address

MISRA C:2004 17.6

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

MISRA C++ 2008 7-5-1

(Required) A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.

Code examples

There are no examples for this check.

MEM-stack-global-alias

Synopsis

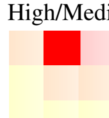
A stack address is stored in a global pointer.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

The address of a variable in stack memory is being stored in a global variable. When the relevant scope or function ends, the memory will become unused, and the externally stored address will point to junk data. This is particularly dangerous because the application might appear to run normally, when it is in fact accessing illegal memory. This might also lead to an application crash, or data changing unpredictably.

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 466

Return of Pointer Value Outside of Expected Range

MISRA C:2004 17.6

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

MISRA C++ 2008 7-5-2

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Code examples

There are no examples for this check.

MEM-stack-global-field

Synopsis

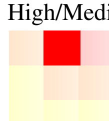
A stack address is stored in the field of a global struct.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

The address of a variable in stack memory is being stored in a global struct. When the relevant scope or function ends, the memory will become unused, and the externally stored address will point to junk data. This is particularly dangerous because the application might appear to run normally, when it is in fact accessing illegal memory. This might also lead to an application crash, or data changing unpredictably.

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 466

Return of Pointer Value Outside of Expected Range

MISRA C:2004 17.6

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

MISRA C:2012 Rule-18.6

(Required) 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

MISRA C++ 2008 7-5-2

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Code examples

The following code example fails the check and will give a warning:

```
struct S{
  int *px;
} s;

void example() {
  int i = 0;
  s.px = &i; //storing local address in global struct
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

struct S{
  int *px;
} s;

void example() {
  int i = 0;
  s.px = &i; //OK - the field is written to later
  s.px = NULL;
}
```

MEM-stack-global

Synopsis

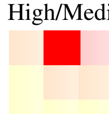
A stack address is stored in a global pointer.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

The address of a variable in stack memory is being stored in a global variable. When the relevant scope or function ends, the memory will become unused, and the externally stored address will point to junk data. This is particularly dangerous because the application might appear to run normally, when it is in fact accessing illegal memory. This might also lead to an application crash, or data changing unpredictably.

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 466

Return of Pointer Value Outside of Expected Range

MISRA C:2004 17.6

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

MISRA C:2012 Rule-18.6

(Required) 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

MISRA C++ 2008 7-5-2

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Code examples

The following code example fails the check and will give a warning:

```
int *px;
void example() {
  int i = 0;
  px = &i; // assigning the address of stack
           // variable a to the global px
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(int *pz) {
  int x; int *px = &x;
  int *py = px; /* local variable */
  pz = px; /* parameter */
}
```

MEM-stack-param-ref (C++ only)

Synopsis

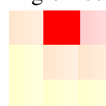
Stack address is stored via reference parameter.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

A stack address is stored outside a function via a parameter of reference type. The address of a local stack variable is assigned to a reference argument of its function. When the function ends, this memory address will become invalid. This is particularly dangerous because the application might appear to run normally, when it is in fact accessing illegal memory. This might also lead to an application crash, or data changing unpredictably.

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 466

Return of Pointer Value Outside of Expected Range

MISRA C++ 2008 7-5-2

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Code examples

The following code example fails the check and will give a warning:

```
void example(int *&pxx) {
  int x;
  pxx = &x;
}
```

The following code example passes the check and will not give a warning about this issue:

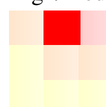
```
void example(int *p, int *&q) {
  int x;
  int *px= &x;
  p = px; // ok, pointer
  q = p; // ok, not local
}
```

MEM-stack-param

Synopsis
Stack address is stored outside function via parameter.

Enabled by default
Yes

Severity/Certainty
High/Medium



Full description
The address of a local stack variable is assigned to a location supplied by the caller via a parameter. When the function ends, this memory address will become invalid. This is particularly dangerous because the application might appear to run normally, when it is in fact accessing illegal memory. This might also lead to an application crash, or data changing unpredictably. Note that this check looks for any expression referring to the store located by the parameter, so the assignment `local[*parameter] = & local;` will trigger the check despite being OK.

Coding standards
CERT DCL30-C

Declare objects with appropriate storage durations

CWE 466

Return of Pointer Value Outside of Expected Range

MISRA C:2004 17.6

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

MISRA C:2012 Rule-18.6

(Required) 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

MISRA C++ 2008 7-5-2

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Code examples
The following code example fails the check and will give a warning:

```
void example(int **ppx) {
  int x;
  ppx[0] = &x;  //local address
}
```

The following code example passes the check and will not give a warning about this issue:

```
static int y = 0;
void example3(int **ppx){
  *ppx = &y;  //OK - static address
}
```

MEM-stack-pos

Synopsis

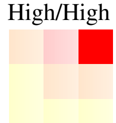
Might return address on the stack.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

A local variable is defined in stack memory, then its address is potentially returned from the function. When the function exits, its stackframe will be considered illegal memory, and thus the address returned might be dangerous. This code and subsequent memory accesses might appear to work, but the operations are illegal and an application crash, or memory corruption, is very likely. To correct this problem, consider returning a copy of the object, using a global variable, or dynamically allocating memory.

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 562

Return of Stack Variable Address

MISRA C:2004 17.6

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

MISRA C++ 2008 7-5-1

(Required) A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.

Code examples

The following code example fails the check and will give a warning:

```
int *example(int *a) {
    int i;
    int *p;
    if (a) {
  p = a;
    } else {
        p = &i;
    }
    return p;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {}
```

MEM-stack-ref (C++ only)

Synopsis

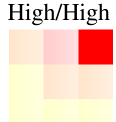
A stack object is returned from a function as a reference.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

A local variable is defined in stack memory, then it is returned from the function as a reference. When the function exits, its stackframe will be considered illegal memory, and thus the return value of the function will refer to an object that no longer exists. Operations on the return value are illegal and an application crash, or memory corruption, is very likely. A safe alternative is for the function to return a copy of the object.

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 562

Return of Stack Variable Address

MISRA C++ 2008 7-5-1

(Required) A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.

Code examples

The following code example fails the check and will give a warning:

```
int& example(void) {
  int x;
  return x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(void) {
  int x;
  return x;
}
```

MEM-stack

Synopsis

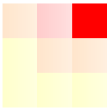
Might return address on the stack.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

A local variable is defined in stack memory, then its address is potentially returned from the function. When the function exits, its stackframe will be considered illegal memory, and thus the address returned might be dangerous. This code and subsequent memory accesses might appear to work, but the operations are illegal and an application crash, or memory corruption, is very likely. To correct this problem, consider returning a copy of the object, using a global variable, or dynamically allocating memory.

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 562

Return of Stack Variable Address

MISRA C:2004 17.6

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

MISRA C:2012 Rule-18.6

(Required) 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

MISRA C++ 2008 7-5-1

(Required) A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.

Code examples

The following code example fails the check and will give a warning:

```
int *f() {
  int x;
  return &x;  //x is a local variable
}
int *example(void) {
  int a[20];
  return a;  //a is a local array
}
```

The following code example passes the check and will not give a warning about this issue:

```
int* example(void) {
  int *p,i;
  p = (int *)malloc(sizeof(int));
  return p;  //OK - p is dynamically allocated

}
```

MEM-use-free-all

Synopsis

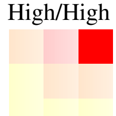
A pointer is used after it has been freed.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

Memory is being accessed after it has been deallocated. The application might appear to run normally, but the operation is illegal. The most likely result is a crash, but the application might keep running with erroneous or corrupt data.

Coding standards

CERT MEM30-C

Do not access freed memory

CWE 416

Use After Free

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

void example(void) {
  int *x;

  x = (int *)malloc(sizeof(int));

  free(x);

  *x++;  //x is dereferenced after it is freed
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

void example(void) {
  int *x;

  x = (int *)malloc(sizeof(int));

  free(x);

  x = (int *)malloc(sizeof(int));

  *x++;  //OK - x is reallocated
}
```

MEM-use-free-some

Synopsis

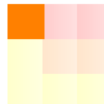
A pointer is used after it has been freed.

Enabled by default

Yes

Severity/Certainty

High/Low



Full description

A pointer is used after it has been freed. This might cause data corruption or an application crash.

Coding standards

CERT MEM30-C

Do not access freed memory

CWE 416

Use After Free

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

void example(void) {
  int *x;

  x = (int *)malloc(sizeof(int));
  free(x);

  if (rand()) {
    x = (int *)malloc(sizeof(int));
  }
  else {
    /* x not reallocated along this path */
  }

  (*x)++;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

void example(void) {
  int *x;

  x = (int *)malloc(sizeof(int));

  free(x);

  x = (int *)malloc(sizeof(int));

  *x++;
}
```

PTR-arith-field

Synopsis

Direct access to a field of a struct, using an offset from the address of the struct.

Enabled by default

Yes

Severity/Certainty

Medium/High

Full description

A field of a struct is accessed directly, using an offset from the address of the struct. Because a struct might in some cases be padded to maintain proper alignment of its fields, it can be very dangerous to access fields using only an offset from the address of the struct itself.

Coding standards

CERT ARR37-C

Do not add or subtract an integer to a pointer to a non-array object

CWE 188

Reliance on Data/Memory Layout

MISRA C:2004 17.1

(Required) Pointer arithmetic shall only be applied to pointers that address an array or array element.

Code examples

The following code example fails the check and will give a warning:

```
struct S{
  char c;
  int x;
};

void main(void) {
  struct S s;
  *(&s.c+1) = 10;
}
```

The following code example passes the check and will not give a warning about this issue:

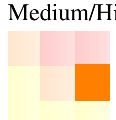
```
struct S{
  char c;
  int x;
};

void example(void) {
  struct S s;
  s.x = 10;
}
```

PTR-arith-stack

Synopsis
Pointer arithmetic applied to a pointer that references a stack address

Enabled by default
Yes

Severity/Certainty
Medium/High



Full description
A pointer is assigned a stack-based address and then used in pointer arithmetic.

Coding standards
CWE 120

> Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

MISRA C:2004 17.1

> (Required) Pointer arithmetic shall only be applied to pointers that address an array or array element.

MISRA C++ 2008 5-0-16

(Required) A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
    int i;
    int *p = &i;
    p++;
    *p = 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
    int i;
    int *p = &i;
    *p = 0;
}
```

PTR-arith-var

Synopsis

Invalid pointer arithmetic with an automatic variable that is neither an array nor a pointer.

Enabled by default

Yes

Severity/Certainty

Medium/High

Full description

The address of an automatic variable is taken, and arithmetic is performed on it. This should be avoided, because memory beyond the memory that was allocated for an automatic variable is invalid, and attempting to access it can lead to an application crash. This check handles local variables, parameters and globals, including structs.

Coding standards

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

MISRA C:2004 17.1

(Required) Pointer arithmetic shall only be applied to pointers that address an array or array element.

MISRA C++ 2008 5-0-16

(Required) A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.

Code examples

The following code example fails the check and will give a warning:

```
void example(int x) {
  *(&x+10) = 5;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(int *x) {
  *(x+10) = 5;
}
```

PTR-cmp-str-lit

Synopsis

A variable is tested for equality with a string literal.

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

A variable is tested for equality with a string literal. This compares the variable with the address of the literal, which is probably not the intended behavior. It is more likely that the intent is to compare the contents of strings at different addresses, for example with the `strcmp()` function.

Coding standards

CWE 597

Use of Wrong Operator in String Comparison

Code examples

The following code example fails the check and will give a warning:

```
#include <stdio.h>

int main (void) {
  char *p = "String";

  if (p == "String") {
    printf("They're equal.\n");
  }

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

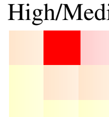
```
#include <stdio.h>
#include <string.h>

int main (void) {
  char *p = "String";

  //OK - using string comparison function
  if (strcmp(p,"String") == 0) {
    printf("They're equal.\n");
  }

  return 0;
}
```

PTR-null-assign-fun-pos

Synopsis: Possible `NULL` pointer dereferenced by a function.

Enabled by default: Yes

Severity/Certainty: High/Medium



Full description: A pointer variable is assigned `NULL`, either directly or as the result of a function call that can return `NULL`. This pointer is then dereferenced, either directly, or by being passed to a function that might dereference it without checking its value. This will cause an application crash.

Coding standards

CERT EXP34-C

Do not dereference null pointers

CWE 476

NULL Pointer Dereference

Code examples

The following code example fails the check and will give a warning:

```
#define NULL ((void*)  0)
void * malloc(unsigned long);

int * xmalloc(int size){

  int * res = malloc(sizeof(int)*size);
  if (res != NULL)
    return res;
  else
    return NULL;
}

void zeroout(int *xp, int i)
{
  xp[i] = 0;
}

int foo() {

  int * x;
  int i;

  x = xmalloc(45);

  // if (x)
  //  return -1;

  for(i = 0; i < 45; i++)
    zeroout(x, i);

}
```

The following code example passes the check and will not give a warning about this issue:

```
#define NULL ((void*)  0)
void * malloc(unsigned long);

int * xmalloc(int size){

  int * res = malloc(sizeof(int)*size);
  if (res != NULL)
    return res;
  else
    return NULL;
}

void zeroout(int *xp, int i)
{
  xp[i] = 0;
}

int foo() {

  int * x;
  int i;

  x = xmalloc(45);

  if (x == NULL)
    return -1;
  else {
    for(i = 0; i < 45; i++)
      zeroout(x, i);
  }
}
```

PTR-null-assign-pos

Synopsis	A pointer is assigned a value that might be `NULL`, and then dereferenced.
Enabled by default	Yes
Severity/Certainty	High/Low

Full description

A pointer is assigned a value that might be `NULL`, and then dereferenced. Often the source of the potential `NULL` pointer is a memory allocation function like `malloc()`, or a sentinel value provided in a user function.

Coding standards

CERT EXP34-C

Do not dereference null pointers

CWE 476

NULL Pointer Dereference

Code examples

The following code example fails the check and will give a warning:

```
#include <string.h>

char *
getenv(const char *name)
{
  return strcmp(name, "HOME")==0 ? "/" : NULL;
}

int
ex(void)
{
  char *p = getenv("USER");

  return *p;  //p might be NULL
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

int main(void)
{
  int *p = malloc(sizeof(int));

  if (p != 0) {
    *p = 4;
  }

  return (int)p;
}
```

PTR-null-assign

Synopsis

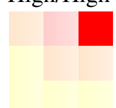
A pointer is assigned the value `NULL`, then dereferenced.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

A pointer is assigned the value `NULL`, then dereferenced. Assigning the pointer the value `NULL` might have been intentional to indicate that the pointer is no longer being used, but it is an error to subsequently dereference it, and will cause an application crash.

Coding standards

CERT EXP34-C

Do not dereference null pointers

CWE 476

NULL Pointer Dereference

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

int main(void) {
  int *p;

  p = NULL;

  return *p;  //dereference after
              //assignment to NULL
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

int main(void) {
  int *p;

  p = NULL;

  p = (int *)1;

  return *p;
}
```

PTR-null-cmp-aft

Synopsis

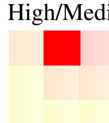
A pointer is dereferenced, then compared with `NULL`.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

A pointer is dereferenced, then compared with `NULL`. Dereferencing a pointer implicitly asserts that it is not `NULL`. Comparing it with `NULL` after this suggests that it might have been `NULL` when it was dereferenced.

Coding standards

CERT EXP34-C

Do not dereference null pointers

CWE 476

NULL Pointer Dereference

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

int example(void) {
  int *p;

  *p = 4;  //line 8 asserts that p may be NULL

  if (p != NULL) {
    return 0;
  }

  return 1;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

void example(int *p) {
  if (p == NULL) {
    return;
  }

  *p = 4;
}
#include <stdlib.h>

void main() {
  int y;
  int* x;
  x = malloc(sizeof(int));
  if (!x)
    return;
  y=*x;
  if (!x)
    return;
  y=*x;
  free(x);
}
```

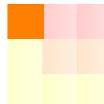
PTR-null-cmp-bef-fun

Synopsis A pointer is compared with `NULL`, then dereferenced by a function.

Enabled by default Yes

Severity/Certainty

High/Low



Full description

A pointer is compared with `NULL`, then passed as an argument to a function that might dereference it. This might occur if the wrong comparison operator is used, for example if `==` instead of `!=`, or if the then- and else- clauses of an if-statement are accidentally swapped. If the function does dereference the pointer, the application will crash. If it does not, the argument is unneeded.

Coding standards

CERT EXP34-C

Do not dereference null pointers

CWE 476

NULL Pointer Dereference

Code examples

The following code example fails the check and will give a warning:

```
#define NULL ((void *) 0)

int baz();

int bar(int *x, int *y, int *z){

  if (x != NULL) {
    *x = 0;
  }


  if (y != NULL) {
    *y = 0;
  }

  *z = 0;

  return 0;
}



int foo(int *x, int *y, int *z) {

  if (x != NULL && y != NULL && z != NULL) {
    *x = 0;
    *y = 0;
    *z = 0;

  }
  baz();

  bar(x,y,z);

}
```

```
#define NULL ((void *) 0)

int bar(int *x){


  *x = 3;

  return 0;
}
```

```
int foo(int *x) {

  if (x != NULL) {

    *x = 4;
  }

  bar(x);

}
```

The following code example passes the check and will not give a warning about this issue:

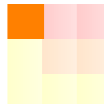
```
#define NULL ((void *) 0)

int bar(int *x){

  if (x != NULL)
    *x = 3;

  return 0;
}



int foo(int *x) {

  if (x != NULL) {

    *x = 4;
  }

  bar(x);

}
```

PTR-null-cmp-bef

Synopsis	A pointer is compared with `NULL`, then dereferenced.
Enabled by default	Yes

Severity/Certainty

High/Low



Full description

A pointer is compared with `NULL`, then dereferenced. This might occur if the wrong comparison operator is used, for example if `==` instead of `!=`, or if the then- and else-clauses of an if-statement are accidentally swapped. If the condition is evaluated and found to be true, the application will crash.

Coding standards

CERT EXP34-C

Do not dereference null pointers

CWE 476

NULL Pointer Dereference

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

int example(void) {
  int *p;

  if (p == NULL) {
    *p = 4;  //dereference after comparison with NULL
  }

  return 1;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

int example(void) {
  int *p;

  if (p != NULL) {
    *p = 4;  //OK - after comparison with non-NULL
  }

  return 1;
}
```

PTR-null-fun-pos

Synopsis

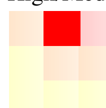
A possible `NULL` pointer is returned from a function, and immediately dereferenced without checking.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

A pointer that might be `NULL` is returned from a function, and immediately dereferenced without checking.

Coding standards

CERT EXP34-C

Do not dereference null pointers

CWE 476

NULL Pointer Dereference

Code examples

The following code example fails the check and will give a warning:

```
#include <string.h>

char *
getenv(const char *name)
{
  return strcmp(name, "HOME")==0 ? "/" : NULL;
}

int
ex(void)
{
  return *getenv("USER");  //getenv() might return NULL
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

int main(void)
{
  int *p = malloc(sizeof(int));

  if (p != 0) {
    *p = 4;
  }

  return (int)p;
}
```

PTR-null-literal-pos

Synopsis

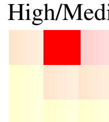
A literal pointer expression (like `NULL`) is dereferenced by a function call.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

A literal pointer expression (for example `NULL`) is passed as argument to a function that might dereference it. Pointer values are generally only useful if acquired at runtime, and thus dereferencing a literal address is usually unintentional, resulting in corrupted memory or an application crash.

Coding standards

CWE 476

NULL Pointer Dereference

Code examples

The following code example fails the check and will give a warning:

```
#define NULL ((void *) 0)

extern int sometimes;

int bar(int *x){
  if (sometimes)
    *x = 3;
  return 0;
}

int foo(int *x) {
  bar(NULL);
}
#define NULL ((void *) 0)

int bar(int *x){
  *x = 3;
  return 0;
}

int foo(int *x) {
  if (x != NULL) {
    *x = 4;
  }
  bar(NULL);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#define NULL ((void *) 0)

int bar(int *x){
  if (x != NULL)
    *x = 3;
  return 0;
}

int foo(int *x) {
  if (x != NULL) {
    *x = 4;
  }
  bar(x);
}
```

PTR-overload (C++ only)

Synopsis

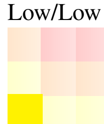
An `&` operator is overloaded.

Enabled by default

No

Severity/Certainty

Low/Low



Full description

The address of an object of incomplete type is taken. Because the complete type contains a user-declared `&` operator, this leads to undefined behavior.

Coding standards

MISRA C++ 2008 5-3-3

(Required) The unary & operator shall not be overloaded.

Code examples

The following code example fails the check and will give a warning:

```
class C{
  bool x;
  bool* operator&();
};

bool* C::operator&(){
  return &x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
class C{
  int x;
  int operator+(int other);
};

int C::operator+(int other){
  return x + other;
}
```

PTR-singleton-arith-pos

Synopsis

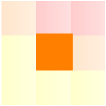
Pointer arithmetic might be performed on a pointer that points to a single object.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

Pointer arithmetic might be performed on a pointer that points to a single object. If this pointer is subsequently dereferenced, it could be pointing to invalid memory, causing a runtime error.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>
void example(int a) {
  int *p;
  if (a) {
    p = malloc(sizeof(int) * 10);
  } else {
    p = malloc(sizeof(int));

  }
  p = p + 1;
}
```

The following code example passes the check and will not give a warning about this issue:

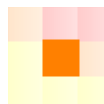
```
#include <stdlib.h>
void example(int a) {
  int *p;
  if (a) {
    p = malloc(sizeof(int) * 10);
  } else {
    p = malloc(sizeof(int) * 20);

  }
  p = p + 1;
}
```

PTR-singleton-arith

Synopsis
Pointer arithmetic is performed on a pointer that points to a single object.

Enabled by default
Yes

Severity/Certainty
Medium/Medium



Full description
Pointer arithmetic is performed on a pointer that points to a single object. If this pointer is subsequently dereferenced, it might be pointing to invalid memory, causing a runtime error.

Coding standards
This check does not correspond to any coding standard rules.

Code examples
The following code example fails the check and will give a warning:

```
#include <stdlib.h>

void example(void) {
  int *p = malloc(sizeof(int));
  p = p + 1;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

void example(void) {
  int *p = malloc(sizeof(int) * 10);
  p = p + 1;
}
```

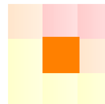
PTR-unchk-param-some

Synopsis
A pointer is dereferenced after being determined not to be `NULL` on some paths, but not checked on others.

Enabled by default
Yes

Severity/Certainty

Medium/Medium



Full description

On some execution paths a pointer is determined not to be `NULL` before being dereferenced, but is dereferenced on other paths without checking. Checking a pointer value indicates that its value might be `NULL`. It should thus be checked on all possible execution paths that result in a dereference.

Coding standards

CWE 822

Untrusted Pointer Dereference

Code examples

The following code example fails the check and will give a warning:

```
int deref(int *p,int q)
{
if(q)
  *p=q;
else{
  if(p == 0)
    return 0;
  else{
    *p=1;
    return 1;
  }
}
}
```

The following code example passes the check and will not give a warning about this issue:

```
#define NULL 0

int safe_deref(int *p)
{
  if (p == NULL) {
    return 0;
  } else {
    return *p;
  }
}
```

PTR-unchk-param

Synopsis
A pointer parameter is not compared to `NULL`

Enabled by default
No

Severity/Certainty
Low/High

Full description
A function dereferences a pointer argument, without first checking that it isn't equal to `NULL`. Dereferencing a `NULL` pointer will cause an application crash.

Coding standards
CWE 822

Untrusted Pointer Dereference

Code examples
The following code example fails the check and will give a warning:

```
int deref(int *p)
{
  return *p;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#define NULL 0

int safe_deref(int *p)
{
  if (p == NULL) {
    return 0;
  } else {
    return *p;
  }
}
```

PTR-uninit-pos

Synopsis
Possible dereference of an uninitialized or `NULL` pointer.

Enabled by default
Yes

Severity/Certainty

Low/High

Full description

On some execution paths, an uninitialized pointer value is dereferenced. This might cause memory corruption or an application crash. Pointer values must be initialized on all execution paths that result in a dereference.

Coding standards

CERT EXP33-C

Do not reference uninitialized memory

CWE 457

Use of Uninitialized Variable

CWE 824

Access of Uninitialized Pointer

MISRA C:2012 Rule-9.1

(Mandatory) The value of an object with automatic storage duration shall not be read before it has been set

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int *p;
  *p = 4;  //p is uninitialized
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int *p,a;
  p = &a;
  *p = 4;  //OK - p holds a valid address
}
```

PTR-uninit

Synopsis

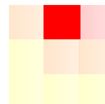
Dereference of an uninitialized or `NULL` pointer.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

An uninitialized pointer value is being dereferenced. This might cause memory corruption or an application crash. Pointer values must be initialized before being dereferenced.

Coding standards

CERT EXP33-C

Do not reference uninitialized memory

CWE 457

Use of Uninitialized Variable

CWE 824

Access of Uninitialized Pointer

MISRA C:2004 9.1

(Required) All automatic variables shall have been assigned a value before being used.

MISRA C++ 2008 8-5-1

(Required) All variables shall have a defined value before they are used.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int *p;
  *p = 4;  //p is uninitialized
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int *p,a;
  p = &a;
  *p = 4;  //OK - p holds a valid address
}
```

RED-case-reach

Synopsis

A case statement within a switch statement cannot be reached.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

A case statement within a switch statement cannot be reached, because the switch statement's expression cannot have the value of the case statement's label. This often occurs because literal values have been assigned to the switch condition. An unreachable case statement is not unsafe as such, but might indicate a programming error.

Coding standards

CERT MSC07-C

Detect and remove dead code

MISRA C:2012 Rule-2.1

(Required) A project shall not contain unreachable code

MISRA C++ 2008 0-1-2

(Required) A project shall not contain infeasible paths.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int x = 42;

  switch(2 * x) {
  case 42 :  //unreachable case, as x is 84
    ;
  default :
    ;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int x = 42;

  switch(2 * x) {
  case 84 :
    ;
  default :
    ;
  }
}
```

RED-cmp-always

Synopsis

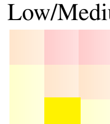
A comparison using ==, <, <=, >, or >= is always true.

Enabled by default

No

Severity/Certainty

Low/Medium



Full description

A comparison using ==, <, <=, >, or >= is always true, given the values of the arguments of the comparison operator. This often occurs because literal values or macros have been used on one or both sides of the operator. Double-check that the operands and the code logic are correct.

Coding standards

CWE 571

Expression is Always True

MISRA C:2004 13.7

(Required) Boolean operations whose results are invariant shall not be permitted.

Code examples

The following code example fails the check and will give a warning:

```
int example(void) {
  int x = 42;

  if (x == 42) {  //always true
    return 0;
  }

  return 1;

}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(void) {
  int x = 42;

  if (rand()) {
    x = 40;
  }

  if (x == 42) {  //OK - may not be true
    return 0;
  }

  return 1;

}
```

RED-cmp-never

Synopsis

A comparison using ==, <, <=, >, or >= is always false.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

A comparison using ==, <, <=, >, or >= is always false, based the values of the arguments of the comparison operator. This often occurs because literal values or macros have been used on one or both sides of the operator. Double-check that the operands and the code logic are correct.

Coding standards

CWE 570

Expression is Always False

MISRA C:2004 13.7

(Required) Boolean operations whose results are invariant shall not be permitted.

Code examples

The following code example fails the check and will give a warning:

```
int example(void) {
  int x = 10;

  if (x < 10) {  //never true
    return 1;
  }

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(int x) {

  if (x < 10) {  //OK - may be true
    return 1;
  }

  return 0;
}
```

RED-cond-always

Synopsis

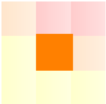
The condition in an if, for, while, do-while, or ternary operator will always be true.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

The condition in an if, for, while, do-while, or ternary operator will always be true. This might indicate a logical error that could result in unexpected runtime behavior.

Coding standards

CERT EXP17-C

Do not perform bitwise operations in conditional expressions

MISRA C:2012 Rule-14.3

(Required) Controlling expressions shall not be invariant

MISRA C++ 2008 0-1-2

(Required) A project shall not contain infeasible paths.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {

    int x = 5;

    for (x = 0; x < 6 && 1; x--) {
    }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {

    int x = 5;

    for (x = 0; x < 6 && 1; x++) {
    }
}
```

RED-cond-const-assign

Synopsis

A constant assignment in a conditional expression.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

An assignment of a constant to a variable is used in a conditional expression. It is most likely an accidental use of the assignment operator (=) instead of the comparison operator (==). The usual result of an assignment operation is the value of the right-hand operand, which in this case is a constant value. This constant value is being compared to zero in the condition, then an execution path is chosen. Any alternate paths are unreachable because of this constant condition.

Coding standards

CWE 481

Assigning instead of Comparing

CWE 570

Expression is Always False

CWE 571

Expression is Always True

Code examples

The following code example fails the check and will give a warning:

```
int * foo(int* y, int size){
  int counter = 100;
  int * orig = y;
  while (y = 0) {
    if (counter)
      continue;
    else
      return orig;

  };
}
```

The following code example passes the check and will not give a warning about this issue:

```
int * foo(int* y, int size){
  int counter = 100;
  int * orig = y;
  while (*y++ = 0) {
    if (++counter)
      continue;
    else
      return orig;

  };
}
```

RED-cond-const-expr

Synopsis

A conditional expression with a constant value

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

A non-trivial expression composed only of constants is used as the truth value in a conditional expression. The condition will either always or never be true, and thus program flow is deterministic, making the test redundant. This check assumes that trivial conditions, such as using a `const` variable or literal directly, are intentional. It is easy to see if they are indeed unintentional.

Coding standards

CWE 570

Expression is Always False

CWE 571

Expression is Always True

Code examples

The following code example fails the check and will give a warning:

```
int foo(int x){
  while (1+1){
  };
}

int foo2(int x){
  for(x = 0; 0 < 10; x++){
  };
}
```

The following code example passes the check and will not give a warning about this issue:

```
int foo(int x){

  while (foo(foo(3))){
    x++;
  }
  return x;
}


int foo2(int x){
  while (0){ // valid usage

  }
  return x;
}
```

RED-cond-const

Synopsis

A constant value is used as the condition for a loop or `if` statement.

Enabled by default

No

Severity/Certainty

Low/High

Full description

A constant value is used as the condition for a loop or `if` statement. This might be an error. If the condition is part of a `for` or `while` loop, it will never terminate.

Coding standards

CWE 570

Expression is Always False

CWE 571

Expression is Always True

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int x = 0;
  while (10){
    ++x;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int x = 0;
  while (x < 10){
    ++x;
  }
}
```

RED-cond-never

Synopsis

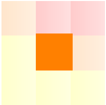
The condition in if, for, while, do-while, or ternary operator will never be true.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

The condition in an if, for, while, do-while, or ternary operator will never be true. This might indicate a logical error that could result in unexpected runtime behavior.

Coding standards

CERT EXP17-C

Do not perform bitwise operations in conditional expressions

MISRA C:2012 Rule-14.3

(Required) Controlling expressions shall not be invariant

MISRA C++ 2008 0-1-2

(Required) A project shall not contain infeasible paths.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {

    int x = 5;

    for (x = 0; x < 6 && x >= 1; x++) {
    }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {

    int x = 5;

    for (x = 0; x < 6 && x >= 0; x++) {
    }
}
```

RED-dead

Synopsis

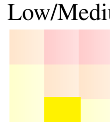
A part of the application is never executed.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

There are statements in the application that cannot be reached on at least some execution paths. Dead code might indicate problems with the application's branching structure.

Coding standards

CERT MSC07-C

Detect and remove dead code

CWE 561

Dead Code

MISRA C:2004 14.1

(Required) There shall be no unreachable code.

MISRA C:2012 Rule-2.1

(Required) A project shall not contain unreachable code

MISRA C++ 2008 0-1-1

(Required) A project shall not contain unreachable code.

MISRA C++ 2008 0-1-9

(Required) There shall be no dead code.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdio.h>

int f(int mode) {
    switch (mode) {
        case 0:
            return 1;
            printf("Hello!"); // This line cannot execute.
        default:
            return -1;
    }
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdio.h>

int f(int mode) {
    switch (mode) {
        case 0:
            printf("Hello!"); // This line can execute.
            return 1;
        default:
            return -1;
    }
}
```

RED-expr

Synopsis

Some expressions, such as `x & x` and `x | x`, are redundant.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

Using one or more variable does not result in a change in that variable, or another variable, or some other side-effect. Giving two identical operands to a bitwise `OR` operator, for example, yields nothing, because the result is equal to the original operands. This might indicate that one of the variables is not intended to be used where it is used. This use of the operator is redundant.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
int example(int x) {
  return x | x;
}
int example(int x) {
  return x & x;
}
void example(int x) {
  x = x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(int x) {
  x = x ^ x;  //OK - x is modified
}
```

RED-func-no-effect

Synopsis

A function is declared that has no return type and creates no side effects.

Enabled by default

No

Severity/Certainty

Low/Low

Full description

A function is declared that has no return type and creates no side effects. This function is meaningless.

Coding standards

MISRA C++ 2008 0-1-8

(Required) All functions with void return type shall have external side effect(s).

Code examples

The following code example fails the check and will give a warning:

```
void pointless (int i, char c)
{
  int local;
  local = 0;
  local = i;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void func(int i)
{
  int p;
  p = i;
  int *ptr;
  ptr = &i;
  i = p;
  i++;
}
```

RED-local-hides-global

Synopsis

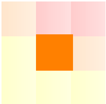
The definition of a local variable hides a global definition.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

A local variable is declared with the same name as a global variable, hiding the global variable from this scope, from this point onwards. This might be intentional, but it is better to use a different name for the local variable, so that a reference to the global variable does not accidentally change or return the local value.

Coding standards

CERT DCL01-C

Do not reuse variable names in subscopes

CERT DCL01-CPP

Do not reuse variable names in subscopes

MISRA C:2004 5.2

(Required) Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.

MISRA C:2012 Rule-5.3

(Required) An identifier declared in an inner scope shall not hide an identifier declared in an outer scope

MISRA C++ 2008 2-10-2

(Required) Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.

Code examples

The following code example fails the check and will give a warning:

```
int x;


int foo (int y ){
  int x=0;
  x++;
  return x+y;

}
```

The following code example passes the check and will not give a warning about this issue:

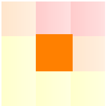
```
int x;


int foo (int y ){

  x++;
  return x+y;

}
```

RED-local-hides-local

Synopsis
The definition of a local variable hides a previous local definition.

Enabled by default
Yes

Severity/Certainty
Medium/Medium



Full description
A local variable is declared with the same name as another local variable, hiding the outer value from this scope, from this point onwards. This might be intentional, but it is better to use a different name for the second variable, so that a reference to the outer variable does not accidentally change or return the inner value.

Coding standards
CERT DCL01-C

Do not reuse variable names in subscopes

CERT DCL01-CPP

Do not reuse variable names in subscopes

MISRA C:2004 5.2

(Required) Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.

MISRA C:2012 Rule-5.3

(Required) An identifier declared in an inner scope shall not hide an identifier declared in an outer scope

MISRA C++ 2008 2-10-2

(Required) Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.

Code examples
The following code example fails the check and will give a warning:

```
int foo(int x ){

  for (int y= 0; y < 10 ; y++){

    for (int y = 0; y < 100; y ++){
      return x+y;
    }
  }
  return x;
}


int foo2(int x){
  int y = 10;

  for (int y= 0; y < 10 ; y++)
    x++;
    return x;
}

int foo3(int x){

  int y = 10;
  {
    int y = 100;
    return x + y;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

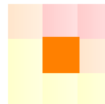
```
int foo(int x){

  for (int y=0; y < 10; y++)
    x++;
  for (int y=0; y < 10; y++)
    x++;
  return x;
}
```

RED-local-hides-member (C++ only)

Synopsis	The definition of a local variable hides a member of the class.
Enabled by default	Yes

Severity/Certainty

Medium/Medium



Full description

A local variable is declared in a class function with the same name as a member of the class, hiding the member from this scope, from this point onwards. This might be intentional, but it is better to use a different name for the variable, so that a reference to the class member does not accidentally change or return the local value.

Coding standards

CERT DCL01-C

Do not reuse variable names in subscopes

CERT DCL01-CPP

Do not reuse variable names in subscopes

MISRA C++ 2008 2-10-2

(Required) Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.

Code examples

The following code example fails the check and will give a warning:

```
class A {
  int x;

public:

  void foo(int y){

    for(int x = 0; x < 10 ; x++){
      y++;
    }

  }

  void foo2(int y){
    int x = 0;
    x+=y;
    return;

  }

  void foo3(int y){

    {
      int x = 0;
      x+=y;
      return;
    }
  }


};
```

The following code example passes the check and will not give a warning about this issue:

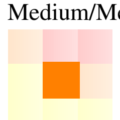
```
class A {
   int x;

};


class B{
   int y;
void foo();
};


void B::foo() {
   int x;
}
```

RED-local-hides-param

Synopsis A variable declaration hides a parameter of the function

Enabled by default Yes

Severity/Certainty Medium/Medium



Full description A local variable is declared in a function with the same name as an argument of the function, hiding the argument from this scope, from this point onwards. This might be intentional, but it is better to use a different name for the variable, so that a reference to the argument does not accidentally change or return the inner value.

Coding standards CERT DCL01-C

Do not reuse variable names in subscopes

CERT DCL01-CPP

Do not reuse variable names in subscopes

MISRA C:2004 5.2

(Required) Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.

MISRA C:2012 Rule-5.3

(Required) An identifier declared in an inner scope shall not hide an identifier declared in an outer scope

MISRA C++ 2008 2-10-2

(Required) Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.

Code examples

The following code example fails the check and will give a warning:

```
int foo(int x){


  for (int x = 0; x < 100; x++);

  return x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int foo(int x){
  int y;

  return x;
}
```

RED-no-effect

Synopsis

A statement potentially contains no side effects.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

A statement expression seems to have no side-effects and is redundant. For example, `5 + 6;` will add `5` and `6`, but will not use the result anywhere. Consequently the statement has no effect on the rest of the application, and should probably be deleted.

Coding standards

CERT MSC12-C

Detect and remove code that has no effect

CWE 482

Comparing instead of Assigning

MISRA C:2004 14.2

(Required) All non-null statements shall either have at least one side effect however executed, or cause control flow to change.

MISRA C:2012 Rule-2.2

(Required) There shall be no dead code

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int x = 1;
  x = 2;
  x < x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <string>
#include "iar.h"

void f();
template<class T>
struct X {
  int x;

  int get() const {
    return x;
  }

  X(int y) :
    x(y) {}

};

typedef X<int> intX;

void example(void) {
  /* everything below has a side-effect */
  int i=0;
  f();
  (void)f();
  ++i;
  i+=1;
  i++;
  char *p = "test";
  STD string s;
  s.assign(p);
  STD string *ps = &s;
  ps -> assign(p);
  intX xx(1);
  xx.get();
  intX(1);
}
```

RED-self-assign

Synopsis	In a C++ class member function, a variable is assigned to itself.
Enabled by default	Yes

Severity/Certainty

Low/High

Full description

In a C++ class member function, a variable is assigned to itself. This error might be harder to identify than in an ordinary C function, because variables might be qualified by `this`, and thus refer to class members.

Coding standards

CWE 480

Use of Incorrect Operator

Code examples

The following code example fails the check and will give a warning:

```
class A {
public :
  int x;
  void f(void) { this->x = x; }  //self-assignment
};

int main(void) {
  A *a = new A();

  a->f();

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
class A {
public :
  int x,y;
  void f(void) { this->x = y; }
};

int main(void) {
  A *a = new A();

  a->f();

  return 0;
}
```

RED-unused-assign

Synopsis

A variable is assigned a non-trivial value that is never used.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

A variable is assigned a non-trivial value that is never used. This is not unsafe as such, but might indicate a logical error.

Coding standards

CERT MSC13-C

Detect and remove unused values

CWE 563

Unused Variable

Code examples

The following code example fails the check and will give a warning:

```
int example(void) {
  int x;

  x = 20;

  x = 3;
  return 0;
}
#include <stdlib.h>

void ex(void) {
  int *p = 0;
  int *q = 0;
  p = malloc(sizeof(int));
  q = malloc(sizeof(int));
  p = q;  //p is not used after this assignment
  return;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

int *ex(void) {
  int *p = 0;
  p = malloc(sizeof(int));
  return p; //the value is returned
}
int example(void) {
  int x;

  x = 20;

  return x;
}
```

RED-unused-param

Synopsis

A function parameter is declared but not used.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

A function parameter is declared but not used. This might be intentional, and is not unsafe as such. For example, the function might need to follow a specific calling convention, or might be a virtual C++ function that does not need as much information from its arguments as other functions do. You should make sure that it is not an error, though.

Coding standards

CWE 563

Unused Variable

MISRA C:2012 Rule-2.7

(Advisory) There should be no unused parameters in functions

MISRA C++ 2008 0-1-11

(Required) There shall be no unused parameters (named or unnamed) in nonvirtual functions.

Code examples

The following code example fails the check and will give a warning:

```
int example(int x) {
  /* `x' is not used */
  return 20;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(int x) {
  return x + 20;
}
```

RED-unused-return-val

Synopsis

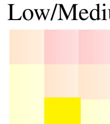
There are unused function return values (other than overloaded operators).

Enabled by default

No

Severity/Certainty

Low/Medium



Full description

There are unused function return values (other than overloaded operators). This might be an error. The return value of a function should always be used. Overloaded operators are excluded; they should behave like the built-in operators. You can discard the return value of a function by using a `(void)` cast.

Coding standards

CWE 252

Unchecked Return Value

MISRA C:2012 Rule-17.7

(Required) The value returned by a function having non-void return type shall be used

MISRA C++ 2008 0-1-7

(Required) The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.

Code examples

The following code example fails the check and will give a warning:

```
int func ( int para1 )
{
   return para1;
}

void discarded ( int para2 )
{
  func(para2);          // value discarded - Non-compliant
}
```

The following code example passes the check and will not give a warning about this issue:

```
int func ( int para1 )
{
   return para1;
}

int not_discarded ( int para2 )
{
  if (func(para2) > 5){
    return 1;
  }
  return 0;
}
```

RED-unused-val

Synopsis

A variable is assigned a value that is never used.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

A variable is initialized or assigned a value, and then another assignment destroys that value before it is used. This is not unsafe as such, but might indicate a logical error. This check does not detect when a value is simply lost when the function ends.

Coding standards

MISRA C:2012 Rule-2.2

(Required) There shall be no dead code

MISRA C++ 2008 0-1-4

(Required) A project shall not contain non-volatile POD variables having only one use.

MISRA C++ 2008 0-1-6

(Required) A project shall not contain instances of non-volatile variables being given values that are never subsequently used.

Code examples

The following code example fails the check and will give a warning:

```
int example(void) {
  int x;

  x = 20;

  x = 3;
  return 0;
}
#include <stdlib.h>

void ex(void) {
  int *p = 0;
  int *q = 0;
  p = malloc(sizeof(int));
  q = malloc(sizeof(int));
  p = q;  //p is not used after this assignment
  return;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

int *ex(void) {
  int *p;
  p = malloc(sizeof(int));
  return p; //the value is returned
}
int example(void) {
  int x;

  x = 20;

  return x;
}
```

RED-unused-var-all

Synopsis

A variable is neither read nor written for any execution path.

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

A variable is neither read nor written for any execution path. Writing includes initialization, and reading includes passing the variable as a parameter in a function call. This is not unsafe as such, but might indicate a logical error.

Coding standards

CERT MSC13-C

Detect and remove unused values

CWE 563

Unused Variable

MISRA C++ 2008 0-1-3

(Required) A project shall not contain unused variables.

Code examples

The following code example fails the check and will give a warning:

```
int example(void) {
  int x;  //this value is not used

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(void) {
  int x = 0;  //OK - x is returned

  return x;
}
```

RESOURCE-deref-file

Synopsis

A pointer to a FILE object is dereferenced.

Enabled by default No

Severity/Certainty Low/Medium

Full description A pointer to a FILE object is dereferenced.

Coding standards MISRA C:2012 Rule-22.5

(Mandatory) A pointer to a FILE object shall not be dereferenced

Code examples The following code example fails the check and will give a warning:

```
#include <stdio.h>

void example(void) {
   FILE *pf1;
   FILE f3;

   f3 = *pf1;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdio.h>

void example(void) {
  FILE *f1;
  FILE *f2;

  f1 = f2;
}
```

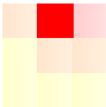
RESOURCE-double-close

Synopsis A file resource is closed multiple times

Enabled by default Yes

Severity/Certainty

High/Medium



Full description

An open file is closed multiple times without being re-opened in between. This will cause an application crash.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdio.h>

void example(void) {
  FILE *f1;
  f1 = fopen("test_file", "w");
  fclose(f1);
  fclose(f1);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdio.h>

void example(void) {
  FILE *f1;
  f1 = fopen("test_file", "w");
  fclose(f1);
}
```

RESOURCE-file-no-close-all

Synopsis

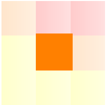
A file pointer is never closed.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

One or more file pointers are never closed. To avoid failure caused by resource exhaustion, all file pointers obtained dynamically by means of Standard Library functions must be explicitly released. Releasing them as soon as possible reduces the risk that exhaustion will occur.

Coding standards

CWE 404

Improper Resource Shutdown or Release

MISRA C:2012 Rule-22.1

(Required) All resources obtained dynamically by means of Standard Library functions shall be explicitly released

Code examples

The following code example fails the check and will give a warning:

```
#include <stdio.h>

void example(void) {
  FILE *fp = fopen("test.txt", "c");
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdio.h>

void example(void) {
  FILE *fp = fopen("test.txt", "c");
  fclose(fp);
}
#include <stdio.h>

void iCloseFilePointers(FILE *fp) {
  fclose(fp);
}

void example(void) {
  FILE *fp = fopen("text.txt", "w");
  iCloseFilePointers(fp);
}
```

RESOURCE-file-pos-neg

Synopsis

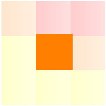
A file handler might be negative

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

A file handler might be negative. If `open()` cannot open a file, it will return a negative file descriptor. Using this file descriptor might cause a runtime error.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <fcntl.h>

void example(void) {
  int a = open("test.txt", O_WRONLY);
  write(a, "Hello", 5);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <fcntl.h>

void example(void) {
  int a = open("test.txt", O_WRONLY);
  if (a > 0) {
    write(a, "Hello", 5);
  }
}
```

RESOURCE-file-use-after-close

Synopsis

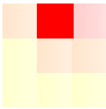
A file resource is used after it has been closed.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

A file resource is referred to after it has been closed. When a file has been closed, any reference to it is invalid. Using this reference might cause an application crash.

Coding standards

This check does not correspond to any coding standard rules.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdio.h>

void example(void) {
  FILE *f1;
  f1 = fopen("test_file", "w");
  fclose(f1);
  fprintf(f1, "Hello, World!\n");
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdio.h>

void example(void) {
  FILE *f1;
  f1 = fopen("test_file", "w");
  fprintf(f1, "Hello, World!\n");
  fclose(f1);
}
```

RESOURCE-implicit-deref-file

Synopsis

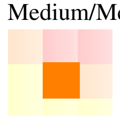
A file pointer is implicitly dereferenced by a library function.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

A file pointer is implicitly dereferenced by a library function.

Coding standards

MISRA C:2012 Rule-22.5

(Mandatory) A pointer to a FILE object shall not be dereferenced

Code examples

The following code example fails the check and will give a warning:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void example(void) {
  FILE *ptr1 = fopen("hello", "r");
  int *a;
  memcpy(ptr1, a, 10);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void example(void) {
  FILE *ptr1;
  int *a;
  memcpy(a, a, 0);
}
```

RESOURCE-write-ronly-file

Synopsis

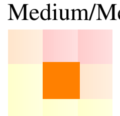
A file opened as read-only is written to.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

A file opened as read-only is written to. This will cause a runtime error in your application, either silently if the file exists, or as a crash if it does not exist.

Coding standards

MISRA C:2012 Rule-22.4

(Mandatory) There shall be no attempt to write to a stream which has been opened as read-only

Code examples

The following code example fails the check and will give a warning:

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

void example(void) {
  FILE *f1;
  f1 = fopen("test-file.txt", "r");
  fprintf(f1, "Hello, World!");
  fclose(f1);
}
```

The following code example passes the check and will not give a warning about this issue:

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

void example(void) {
  FILE *f1;
  f1 = fopen("test-file.txt", "r+");
  fprintf(f1, "Hello, World!");
  fclose(f1);
}
```

SIZEOF-side-effect

Synopsis

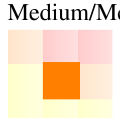
`sizeof` expressions containing side effects

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

The `sizeof` operator is used on an expression that contains side effects. Because `sizeof` only operates on the type of the expression, the expression itself is not evaluated, which it probably was meant to be.

Coding standards

CERT EXP06-C

Operands to the sizeof operator should not contain side effects

CERT EXP06-CPP

Operands to the sizeof operator should not contain side effects

MISRA C:2004 12.3

(Required) The sizeof operator shall not be used on expressions that contain side effects.

MISRA C++ 2008 5-3-4

(Required) Evaluation of the operand to the sizeof operator shall not contain side effects.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int i;
  int size = sizeof(i++);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int i;
  int size = sizeof(i);
  i++;
}
```

SPC-init-list

Synopsis

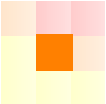
The initalization list of an array contains side effects.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

The initalization list of an array contains side effects.

Coding standards

MISRA C:2012 Rule-13.1

(Required) Initializer lists shall not contain persistent side effects

Code examples

The following code example fails the check and will give a warning:

```
volatile int v1;

extern void p ( int a[2] );

int x = 10;

void example(void) {
  int a[2] = { v1, 0 };

  p( (int[2]) { x++, x-- });

}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int a[2] = { 1, 2 };
}
```

SPC-order

Synopsis

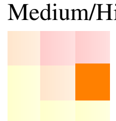
Expressions which depend on order of evaluation

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

One and the same variable is changed in different parts of an expression with an unspecified evaluation order, between two consecutive sequence points. Standard C does not specify an evaluation order for different parts of an expression. For this reason different compilers are free to perform their own optimizations regarding the evaluation order. Projects containing statements that violate this check are not easily ported to another architecture or compiler, and if they are they might be difficult to debug. Only four operators have a guaranteed order of evaluation: logical AND (`a && b`) evaluates the left operand, then the right operand only if the left is found to be true; logical OR (`a || b`) evaluates the left operand, then the right operand only if the left is found to be false; a ternary conditional (`a ? b : c`) evaluates the first operand, then either the second or the third, depending on whether the first is found to be true or false; and a comma (`a , b`) evaluates its left operand before its right.

Coding standards

CERT EXP10-C

Do not depend on the order of evaluation of subexpressions or the order in which side effects take place

CERT EXP30-C

Do not depend on order of evaluation between sequence points

CWE 696

Incorrect Behavior Order

MISRA C:2004 12.2

(Required) The value of an expression shall be the same under any order of evaluation that the standard permits.

MISRA C:2012 Rule-13.2

(Required) The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders

MISRA C++ 2008 5-0-1

(Required) The value of an expression shall be the same under any order of evaluation that the standard permits.

Code examples

The following code example fails the check and will give a warning:

```
int main(void) {
  int i = 0;

  i = i * i++;  //unspecified order of operations

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int main(void) {
  int i = 0;
  int x = i;

  i++;
  x = x * i;  //OK - statement is broken up

  return 0;
}
```

SPC-uninit-arr-all

Synopsis

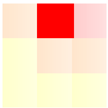
Reads from local buffers are not preceded by writes.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

A value is read from an array, without being explicitly stored in that array first. This check determines whether at least one element of an array has been written before any element of the array is read. If the check triggers, it generally means that an uninitialized value is read. This might cause incorrect behavior or an application crash.

Coding standards

CERT EXP33-C

Do not reference uninitialized memory

CWE 457

Use of Uninitialized Variable

MISRA C:2004 1.2

(Required) No reliance shall be placed on undefined or unspecified behavior.

MISRA C:2012 Rule-9.1

(Mandatory) The value of an object with automatic storage duration shall not be read before it has been set

Code examples

The following code example fails the check and will give a warning:

```
void example() {
  int x[20];
  x[0] = 1;
  int b = x[1]; /* bad read, x[0] was initialized but x[1] wasn't
*/
}
/* won't work until signature of memcpy is known */
#include <string.h>
void example() {
  int a[20];
  int b[20];
  memcpy(a,b,20);
}

/* read thru alias */
void example() {
  int x[20];
  int *a = x;
  int b = a[1]; /* read x thru alias a, but x not init */
}
void example() {
  int a[20];
  int b = a[1];
}
void example() {
  int x[20];
  *x = 1;
  int b = x[1]; /* bad read, x[0] was initialized but x[1] wasn't
*/
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example() {
  int x[20];
  int *p = x;
  x[0]=1;
  int k = *p; /* read thru alias */
}
void example() {
  int x[20];
  int *p = x;
  p[0]=1; /* write thru alias */
  int k = *x;
}
struct X { int e; };
void example() {
  struct X x[20];
  x->e = 1;
  { struct X b = x[0]; } /* x[0] has been initialized via x->e,
but Goanna currently doesn't have pointer alias analysis on
individual array elements */
}
void example() {
  int x[20];
  *(x+0) = 1;
  int b = x[1]; /* bad read but check can't detect which elements
*/
}
extern void f(int*);
void example() {
  int a[20];
  f(a);
  int b = a[1];
}
void example() {
  int a[20] =
{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
  int b = a[1];
}
void example() {
  int x[20];
  *x = 1;
  int b = x[1]; /* bad read but check can't detect which elements
*/
}
/* write thru alias */
void example() {
  int x[20];
  int *a = x;
```

```
  f(a); /* assumed init of x thru alias a */
  int b = x[1];
}
void example() {
  int x[20];
  x[0] = 1;
  int b = x[1]; /* bad read but check can't detect which elements
*/
}
```

SPC-uninit-struct-field-heap

Synopsis

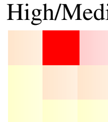
A field of a dynamically allocated struct is read before it is initialized.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

A field of a dynamically allocated struct is read before it is initialized. An uninitialized field might cause unexpected and unpredictable results. Uninitialized variables are easy to overlook, because they seldom cause problems.

Coding standards

CERT EXP33-C

Do not reference uninitialized memory

CWE 457

Use of Uninitialized Variable

Code examples

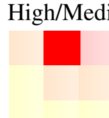
The following code example fails the check and will give a warning:

```
#include <stdlib.h>

struct st {
  int x;
  int y;
};

void example(void) {
  int a;
  struct st *str = malloc(sizeof(struct st));
  a = str->x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

struct st {
  int x;
  int y;
};

void example(void) {
  int a;
  struct st *str = malloc(sizeof(struct st));
  str->x = 0;
  a = str->x;
}
```

SPC-uninit-struct-field

Synopsis	A field of a local struct is read before it is initialized.
Enabled by default	Yes
Severity/Certainty	High/Medium 
Full description	A field of a local struct is read before it is initialized. An uninitialized field might cause unexpected and unpredictable results. Uninitialized variables are easy to overlook, because they seldom cause problems.

Coding standards

CERT EXP33-C

Do not reference uninitialized memory

CWE 457

Use of Uninitialized Variable

MISRA C:2012 Rule-9.1

(Mandatory) The value of an object with automatic storage duration shall not be read before it has been set

Code examples

The following code example fails the check and will give a warning:

```
struct st {
  int x;
  int y;
};

void example(void) {
  int a;
  struct st str;
  a = str.x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
struct st {
  int x;
  int y;
};

void example(void) {
  int a;
  struct st str;
  str.x = 0;
  a = str.x;
}
```

SPC-uninit-struct

Synopsis

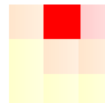
A struct has one or more fields read before they are initialized.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

A struct is read from before any of its fields are initialized. Using uninitialized values might cause unexpected results or unpredictable application behavior, particularly in the case of pointer fields.

Coding standards

CERT EXP33-C

Do not reference uninitialized memory

CWE 457

Use of Uninitialized Variable

MISRA C:2004 1.2

(Required) No reliance shall be placed on undefined or unspecified behavior.

MISRA C:2012 Rule-9.1

(Mandatory) The value of an object with automatic storage duration shall not be read before it has been set

Code examples

The following code example fails the check and will give a warning:

```
struct st {
  int x;
  int y;
};

void example(void) {
  int a;
  struct st str;
  a = str.x;
}
```

The following code example passes the check and will not give a warning about this issue:

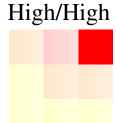
```
struct st {
  int x;
  int y;
};

void example(int i) {
  int a;
  struct st str;
  str.x = i;
  a = str.x;
}
```

SPC-uninit-var-all

Synopsis
A variable is read before it is assigned a value.

Enabled by default
Yes

Severity/Certainty
High/High



Full description
A variable is read before it is assigned a value. Different execution paths might result in a variable being read at different points in the execution. Because uninitialized data is read, application behavior might be unpredictable.

Coding standards
CERT EXP33-C

Do not reference uninitialized memory

CWE 457

Use of Uninitialized Variable

MISRA C:2004 9.1

(Required) All automatic variables shall have been assigned a value before being used.

MISRA C:2012 Rule-9.1

(Mandatory) The value of an object with automatic storage duration shall not be read before it has been set

MISRA C++ 2008 8-5-1

(Required) All variables shall have a defined value before they are used.

Code examples

The following code example fails the check and will give a warning:

```
int main(void) {
  int x;

  x++;  //x is uninitialized

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int main(void) {
  int x = 0;

  x++;

  return 0;
}
```

SPC-uninit-var-some

Synopsis

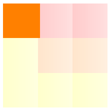
A variable is read before it is assigned a value.

Enabled by default

Yes

Severity/Certainty

High/Low



Full description

A variable is read before it is assigned a value. On some execution paths, the variable might be assigned a value before it is read. This might cause unpredictable application behavior.

Coding standards

CWE 457

Use of Uninitialized Variable

MISRA C:2004 9.1

(Required) All automatic variables shall have been assigned a value before being used.

MISRA C:2012 Rule-9.1

(Mandatory) The value of an object with automatic storage duration shall not be read before it has been set

MISRA C++ 2008 8-5-1

(Required) All variables shall have a defined value before they are used.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

int main(void) {
  int x, y;

  if (rand()) {
    x = 0;
  }

  y = x;  //x may not be initialized

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

int main(void) {
  int x;

  if (rand()) {
    x = 0;
  }

  /* x never read */

  return 0;
}
```

SPC-volatile-reads

Synopsis

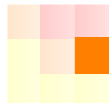
There are multiple read accesses with volatile-qualified type within one and the same sequence point.

Enabled by default

No

Severity/Certainty

Medium/High



Full description

There are multiple read accesses with volatile-qualified type within one and the same sequence point. There cannot be more than one read access with volatile-qualified type within a sequence point.

Coding standards

CERT EXP10-C

Do not depend on the order of evaluation of subexpressions or the order in which side effects take place

CERT EXP30-C

Do not depend on order of evaluation between sequence points

CWE 696

Incorrect Behavior Order

MISRA C:2004 12.2

(Required) The value of an expression shall be the same under any order of evaluation that the standard permits.

MISRA C:2012 Rule-13.2

(Required) The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders

MISRA C++ 2008 5-0-1

(Required) The value of an expression shall be the same under any order of evaluation that the standard permits.

Code examples

The following code example fails the check and will give a warning:

```
#include "mc2_types.h"
//#include "mc2_header.h"

void example(void) {
  uint16_t x;
  volatile uint16_t v;
  x = v + v;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int main(void) {
  int i = 0;
  int x = i;

  i++;
  x = x * i;  //OK - statement is broken up

  return 0;
}
```

SPC-volatile-writes

Synopsis

There are multiple write accesses with volatile-qualified type within one and the same sequence point.

Enabled by default

No

Severity/Certainty

Medium/High

Full description

There are multiple write accesses with volatile-qualified type within one and the same sequence point. There cannot be more than one write access with volatile-qualified type within a sequence point.

Coding standards

CERT EXP10-C

> Do not depend on the order of evaluation of subexpressions or the order in which side effects take place

CERT EXP30-C

> Do not depend on order of evaluation between sequence points

CWE 696

> Incorrect Behavior Order

MISRA C:2004 12.2

> (Required) The value of an expression shall be the same under any order of evaluation that the standard permits.

MISRA C:2012 Rule-13.2

(Required) The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders

MISRA C++ 2008 5-0-1

(Required) The value of an expression shall be the same under any order of evaluation that the standard permits.

Code examples

The following code example fails the check and will give a warning:

```
#include "mc2_types.h"
//#include "mc2_header.h"

void example(void) {
  uint16_t x;
  volatile uint16_t v, w;
  v = w = x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdbool.h>
void InitializeArray(int *);
const int *example(void)
{
   static volatile bool s_initialized = false;
   static int s_array[256];

   if (!s_initialized)
   {
      InitializeArray(s_array);
      s_initialized = true;
   }
   return s_array;
}
```

STR-trigraph

Synopsis

Trigraphs are used (in string literals only).

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

Trigraphs are used (in string literals only). Trigraphs can cause confusion with other uses of two question marks and should not be used.

Coding standards

MISRA C:2004 4.2

(Required) Tri-graphs shall not be used

MISRA C:2012 Rule-4.2

(Advisory) Trigraphs should not be used

MISRA C++ 2008 2-3-1

(Required) Trigraphs shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
void func()
{
  char * str = "abc??!def";
}
```

The following code example passes the check and will not give a warning about this issue:

```
void func()
{
  char * str = "abc??def";
}
```

STRUCT-signed-bit

Synopsis

There are signed single-bit fields (excluding anonymous fields).

Enabled by default

No

Severity/Certainty

Low/Low

Full description

There are signed single-bit fields (excluding anonymous fields). A signed bitfield should have size at least two, because one bit is required for the sign.

Coding standards

MISRA C:2004 6.5

(Required) Bitfields of signed type shall be at least 2 bits long.

MISRA C:2012 Rule-6.2

(Required) Single-bit named bit fields shall not be of a signed type

MISRA C++ 2008 9-6-4

(Required) Named bit-fields with signed integer type shall have a length of more than one bit.

Code examples

The following code example fails the check and will give a warning:

```
struct S
{
  signed int a : 1; // Non-compliant
};
```

The following code example passes the check and will not give a warning about this issue:

```
struct S
{
  signed int b : 2;
  signed int   : 0;
  signed int   : 1;
  signed int   : 2;
};
```

SWITCH-fall-through

Synopsis

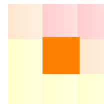
There are non-empty switch cases not terminated by break and without 'fallthrough' comment.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

There are non-empty switch cases not terminated by a break. A non-empty switch clause should be terminated by an unconditional break statement, unless explicitly commented as a 'fallthrough'.

Coding standards

CERT MSC17-C

> Finish every set of statements associated with a case label with a break statement

Code examples

The following code example fails the check and will give a warning:

```
void example(int input) {

  while (rand()) {
    switch(input) {
      case 0:
        if (rand()) {
          break;
        }
      default:
        break;
    }
  }
}
void example(int input) {

  switch(input) {
    case 0:
      if (rand()) {
        break;
      }
    default:
      break;
  }

}
```

The following code example passes the check and will not give a warning about this issue:

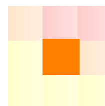
```
void example(int input) {

  switch(input) {
    case 0:
      if (rand()) {
        break;
      }
      break;
    case 1:
      if (rand()) {
        break;
      }
      // fallthrough
    case 2:
      // this should also fall through
      if (!rand()) {
        return;
      }
    default:
      break;
  }

}
void example(int input) {

  switch(input) {
    case 0:
      if (rand()) {
        break;
      } else {
        break;
      }
      // All paths above contain a break, therefore we do not
warn
    default:
      break;
  }

}
```

THROW-empty (C++ only)

Synopsis	Unsafe rethrow of exception.
Enabled by default	No

Severity/Certainty

Medium/Medium



Full description

A `throw` statement without an argument is used outside of a `catch` handler where there is no exception to rethrow. This is unsafe because a `throw` statement without an argument rethrows the temporary object that represents the current exception, to allow exception handling to be split over several handlers.

Coding standards

MISRA C++ 2008 15-1-3

> (Required) An empty throw (throw;) shall only be used in the compound-statement of a catch handler.

Code examples

The following code example fails the check and will give a warning:

```
void func()
{
  try
  {
    throw;
  }
  catch (...) {}
}
```

The following code example passes the check and will not give a warning about this issue:

```
void func()
{
  try
  {
    throw (42);
  }
  catch (int i)
  {
    if (i > 10)
    {
      throw;
    }
  }
}
```

THROW-main (C++ only)

Synopsis

No default exception handler for `try`.

Enabled by default

No

Severity/Certainty

Medium/Low

Full description

A top level `try` block does not have a default exception handler that will catch exceptions. Without this, an unhandled exception might lead to termination in an implementation-defined manner.

Coding standards

MISRA C++ 2008 15-3-2

> (Advisory) There should be at least one exception handler to catch all otherwise unhandled exceptions

Code examples

The following code example fails the check and will give a warning:

```
int main()
{
  try
  {
    throw (42);
  }
  catch (int i)
  {
    if (i > 10)
    {
      throw;
    }
  }
  return 1;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int main()
{
  try
  {
    throw;
  }
  catch (...) {}
  // spacer
  try {}
  catch (int i) {}
  catch (...) {}
  return 0;
}
```

THROW-null

Synopsis

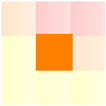
Throw of NULL integer constant

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

`throw(NULL)` (equivalent to `throw(0)`) is never a throw of the null-pointer-constant, which means it can only be caught by an integer handler. This might be undesired behavior, especially if your application only has handlers for pointer-to-type exceptions.

Coding standards

MISRA C++ 2008 15-1-2

(Required) NULL shall not be thrown explicitly.

Code examples

The following code example fails the check and will give a warning:

```
typedef intint32_t;
typedefsigned charchar_t;
#defineNULL0

void example(void)
{
  try {
    throw ( NULL );           // Non-compliant
  }
  catch ( int32_t i ) {       // NULL exception handled here
    // ...
  }
  catch ( const char_t * ) { // Developer may expect it to be
caught here
    // ...
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
typedef intint32_t;
typedefsigned charchar_t;
#defineNULL0

void example(void)
{
  char_t * p = NULL;
  try {
    throw ( p );              // Compliant
  }
  catch ( int32_t i ) {
    // ...
  }
  catch ( const char_t * ) { // Exception handled here
    // ...
  }
}
```

THROW-ptr

Synopsis

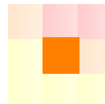
Throw of exceptions by pointer

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

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

Coding standards

CERT ERR09-CPP

Throw anonymous temporaries and catch by reference

MISRA C++ 2008 15-0-2

(Advisory) An exception object should not have pointer type.

Code examples

The following code example fails the check and will give a warning:

```
class Except {};

Except *new_except();

void example(void)
{
    throw new Except();
}
```

The following code example passes the check and will not give a warning about this issue:

```
class Except {};

void example(void)
{
    throw Except();
}
```

THROW-static (C++ only)

Synopsis

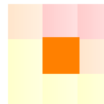
Exceptions thrown without a handler in some call paths that lead to that point.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

There are exceptions thrown without a handler in some call paths that lead to that point. If an application throws an unhandled exception, it terminates in an implementation-defined manner. In particular, it is implementation-defined whether the call stack is unwound before termination, so the destructors of any automatic objects might not be invoked. If an exception is thrown as an object of a derived class, a compatible type might be either the derived class or any of its bases. Make sure that the application catches all exceptions it is expected to throw.

Coding standards

MISRA C++ 2008 15-3-1

(Required) Exceptions shall be raised only after start-up and before termination of the program.

Code examples

The following code example fails the check and will give a warning:

```
class C {
public:
    C ( ) { throw ( 0 ); }  // Non-compliant – thrown before main
starts
    ~C ( ) { throw ( 0 ); } // Non-compliant – thrown after main
exits
};

C c;  // An exception thrown in C's constructor or destructor
will
      // cause the program to terminate, and will not be caught
by
      // the handler in main

int main( ... )
{
    try {
        // program code
        return 0;
    }
    // The following catch-all exception handler can only
    // catch exceptions thrown in the above program code
    catch ( ... ) {
        // Handle exception
        return 0;
    }
}
```

The following code example passes the check and will not give a warning about this issue:

```
class C {
public:
    C ( ) {  }  // Compliant - doesn't throw exceptions
    ~C ( ) {  } // Compliant - doesn't throw exceptions
};

C c;

int main( ... )
{
    try {
        // program code
        return 0;
    }
    // The following catch-all exception handler can only
    // catch exceptions thrown in the above program code
    catch ( ... ) {
        // Handle exception
        return 0;
    }
}
```

THROW-unhandled (C++ only)

Synopsis

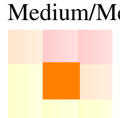
There are calls to functions explicitly declared to throw an exception type that is not handled (or declared as thrown) by the caller.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

There are calls to functions explicitly declared to throw an exception type that is not handled (or declared as thrown) by the caller. If an application throws an unhandled exception, it terminates in an implementation-defined manner. In particular, it is implementation-defined whether the call stack is unwound before termination, so the destructors of any automatic objects might not be invoked. If an exception is thrown as an object of a derived class, a compatible type might be either the derived class or any of its bases. Make sure that the application catches all exceptions it is expected to throw.

Coding standards

MISRA C++ 2008 15-3-4

(Required) Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.

Code examples

The following code example fails the check and will give a warning:

```
class E1{};

void foo(int i) throw (E1) {
  if (i<0)
    throw E1();
}

int bar() {
  foo(-3);
}
class E1{};

void foo(int i) throw (E1) {
  if (i<0)
    throw E1();
}

int bar() throw (E1) {  //warning about E1 because it is not
EXPLICITLY caught
  foo(-3);
}
```

The following code example passes the check and will not give a warning about this issue:

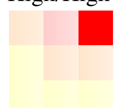
```
class E1{};

void foo(int i) throw (E1) {
  if (i<0)
    throw E1();
}

int bar() {
  try {
    foo(-3);
  }
  catch (E1){
  }
}
```

UNION-overlap-assign

Synopsis
Assignments from one field of a union to another.

Enabled by default
Yes

Severity/Certainty
High/High



Full description
There are assignments from one field of a union to another. Assignments between objects that are stored in the same physical memory causes undefined behavior.

Coding standards
MISRA C:2004 18.2

(Required) An object shall not be assigned to an overlapping object.

MISRA C:2012 Rule-19.1

(Mandatory) An object shall not be assigned or copied to an overlapping object

MISRA C++ 2008 0-2-1

(Required) An object shall not be assigned to an overlapping object.

Code examples
The following code example fails the check and will give a warning:

```
union cheat {
    char c[5];
    int i;
};

void example(union cheat *u)
{
    u->i = u->c[2];
}
union {
    char c[5];
    int i;
} u;

void example(void)
{
    u.i = u.c[2];
}
void example(void)
{
  union
  {
    char c[5];
    int i;
  } u;
  u.i = u.c[2];
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void)
{
  union
  {
    char c[5];
    int i;
  } u;
  int x;
  x = (int)u.c[2];
  u.i = x;
}
void example(void)
{
  struct
  {
    char c[5];
    int i;
  } u;
  u.i = u.c[2];
}
union cheat {
  char c[5];
  int i;
};

union cheat u;

void example(void)
{
  int x;
  x = (int)u.c[2];
  u.i = x;
}
```

UNION-type-punning

Synopsis

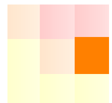
Writing to a field of a union after reading from a different field, effectively re-interpreting the bit pattern with a different type.

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

Writing to one field of a union and then silently reading from another field circumvents the type system. To reinterpret bit patterns deliberately, use an explicit cast.

Coding standards

CERT EXP39-C

Do not access a variable through a pointer of an incompatible type

CWE 188

Reliance on Data/Memory Layout

MISRA C:2004 12.12

(Required) The underlying bit representations of floating-point values shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
union name {
  int int_field;
  float float_field;
};

void example(void) {
  union name u;
  u.int_field = 10;
  float f = u.float_field;
}
```

The following code example passes the check and will not give a warning about this issue:

```
union name {
  int int_field;
  float float_field;
};

void example(void) {
  union name u;
  u.int_field = 10;
  float f = u.int_field;
}
```

MISRAC2004-1.2_a

Synopsis

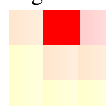
Checks reads from local buffers are preceded by writes.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

(Required) No reliance shall be placed on undefined or unspecified behavior. This is a semi-equivalent initialization check for arrays, which ensures that at least one element of the array has been written before any element is attempted to be read. A warning generally means that you have read an uninitialized value, and the program may behave erroneously or crash in some situations.

Coding standards

CERT EXP33-C

Do not reference uninitialized memory

CWE 457

Use of Uninitialized Variable

MISRA C:2004 1.2

(Required) No reliance shall be placed on undefined or unspecified behavior.

Code examples

The following code example fails the check and will give a warning:

```
void example() {
  int x[20];
  x[0] = 1;
  int b = x[1]; /* bad read but check can't detect which elements
*/
}
/* won't work until signature of memcpy is known */
#include <string.h>
void example() {
  int a[20];
  int b[20];
  memcpy(a,b,20);
}

/* read thru alias */
void example() {
  int x[20];
  int *a = x;
  int b = a[1]; /* read x thru alias a, but x not init */
}
void example() {
  int a[20];
  int b = a[1];
}
void example() {
  int x[20];
  *x = 1;
  int b = x[1]; /* bad read but check can't detect which elements
*/
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example() {
  int x[20];
  int *p = x;
  x[0]=1;
  int k = *p; /* read thru alias */
}
void example() {
  int x[20];
  int *p = x;
  p[0]=1; /* write thru alias */
  int k = *x;
}
struct X { int e; };
void example() {
  struct X x[20];
  x->e = 1;
  { struct X b = x[0]; } /* x[0] has been initialized via x->e,
but Goanna currently doesn't have pointer alias analysis on
individual array elements */
}
void example() {
  int x[20];
  *(x+0) = 1;
  int b = x[1]; /* bad read but check can't detect which elements
*/
}
extern void f(int*);
void example() {
  int a[20];
  f(a);
  int b = a[1];
}
void example() {
  int a[20] =
{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
  int b = a[1];
}
void example() {
  int x[20];
  *x = 1;
  int b = x[1]; /* bad read but check can't detect which elements
*/
}
/* write thru alias */
void example() {
  int x[20];
  int *a = x;
```

```
  f(a); /* assumed init of x thru alias a */
  int b = x[1];
}
void example() {
  int x[20];
  x[0] = 1;
  int b = x[1]; /* bad read but check can't detect which elements
*/
}
```

MISRAC2004-1.2_b

Synopsis

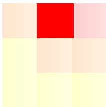
In all executions, a struct has one or more fields read before they are initialized.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

(Required) No reliance shall be placed on undefined or unspecified behavior. Using uninitialized values could lead to unexpected results or unpredictable program behavior, particularly in the case of pointer fields.

Coding standards

CERT EXP33-C

Do not reference uninitialized memory

CWE 457

Use of Uninitialized Variable

MISRA C:2004 1.2

(Required) No reliance shall be placed on undefined or unspecified behavior.

Code examples

The following code example fails the check and will give a warning:

```
struct st {
  int x;
  int y;
};

void example(void) {
  int a;
  struct st str;
  a = str.x;
}
```

The following code example passes the check and will not give a warning about this issue:

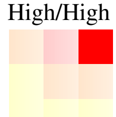
```
struct st {
  int x;
  int y;
};

void example(int i) {
  int a;
  struct st str;
  str.x = i;
  a = str.x;
}
```

MISRAC2004-1.2_c

Synopsis
An expression resulting in `0` is used as a divisor.

Enabled by default
Yes

Severity/Certainty
High/High



Full description
(Required) No reliance shall be placed on undefined or unspecified behavior. by interval analysis to be `0`, and it is used as a divisor. If this code executes, a `divide by zero' runtime error will occur.

Coding standards
CERT INT33-C

Ensure that division and modulo operations do not result in divide-by-zero errors

CWE 369

Divide By Zero

MISRA C:2004 1.2

(Required) No reliance shall be placed on undefined or unspecified behavior.

Code examples

The following code example fails the check and will give a warning:

```
int foo(void)
{
  int a = 3;
  a--;
  return 5 / (a-2);  // a-2 is 0
}
#include <stdlib.h>

int main (void)

{
 int *p = malloc( sizeof(int));
  int x = foo (p);
  /* foo(2) returns 8, so we have a division by zero below)*/
  x = 1 / (x - 8);              /*@@ZDV-RED@@ */

  return x;
}


int foo(int * p){
  return 8;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int foo(void)
{
  int a = 3;
  a--;
  return 5 / (a+2);  // OK - a+2 is 4
}
```

MISRAC2004-1.2_d

Synopsis

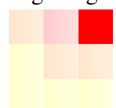
A variable is assigned the value `0`, then used as a divisor.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

(Required) No reliance shall be placed on undefined or unspecified behavior. If this code executes, a `divide by zero' runtime error will occur.

Coding standards

CERT INT33-C

> Ensure that division and modulo operations do not result in divide-by-zero errors

CWE 369

> Divide By Zero

MISRA C:2004 1.2

> (Required) No reliance shall be placed on undefined or unspecified behavior.

Code examples

The following code example fails the check and will give a warning:

```
int foo(void)
{
  int a = 20, b = 0, c;

  c = a / b;     /* Divide by zero */

  return c;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int foo(void)
{
  int a = 20, b = 5, c;

  c = a / b; /* b is not 0 */

  return c;
}
int main() {
    int totallen = 0;
    int i=0;
    float tmp=1;

   for( i=1; i<10; i++){
   totallen++;
   }

   foo(2/totallen);

   return 0;
}

int foo(int x){
   return x;
}
```

MISRAC2004-1.2_e

Synopsis

After a successful comparison with `0`, a variable is used as a divisor.

Enabled by default

Yes

Severity/Certainty

Medium/High

Full description

(Required) No reliance shall be placed on undefined or unspecified behavior. then used as a divisor without being written to beforehand. The presence of this comparison implies that the variable's value is `0` for the following statements. As such, its being used as a divisor afterwards would invoke a `divide by zero' runtime error.

Coding standards

CERT INT33-C

Ensure that division and modulo operations do not result in divide-by-zero errors

CWE 369

Divide By Zero

MISRA C:2004 1.2

(Required) No reliance shall be placed on undefined or unspecified behavior.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>
int foo(void)
{
  int a = 20;
  int p = rand();

  if (p == 0)   /* p is 0 */
    a = 34 / p;

  return a;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>
int foo(void)
{
  int a = 20;
  int p = rand();

  if (p != 0)   /* p is not 0 */
    a = 34 / p;

  return a;
}
```

MISRAC2004-1.2_f

Synopsis

A variable used as a divisor is subsequently compared with `0`.

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

(Required) No reliance shall be placed on undefined or unspecified behavior. This check will produce a warning if a variable is compared to `0` after it is used as a divisor, but before it is written to again. The comparison implies that the variable's value may be `0`, and thus may have been for the preceeding statements. As one of these statements is an operation using the variable as a divisor (which would invoke a `divide by zero' runtime error), the program's execution can never reach the comparison when the value is `0`, rendering it redundant.

Coding standards

CERT INT33-C

> Ensure that division and modulo operations do not result in divide-by-zero errors

CWE 369

> Divide By Zero

MISRA C:2004 1.2

> (Required) No reliance shall be placed on undefined or unspecified behavior.

Code examples

The following code example fails the check and will give a warning:

```
int foo(int p)
{
  int a = 20, b = 1;
  b = a / p;
  if (p == 0) // Checking the value of 'p' too late.
    return 0;
  return b;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int foo(int p)
{
  int a = 20, b;
  if (p == 0)
    return 0;
  b = a / p;    /* Here 'p' is non-zero. */
  return b;
}
```

MISRAC2004-1.2_g

Synopsis

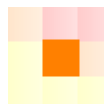
Interval analysis determines a value is `0`, then it is used as a divisor.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) No reliance shall be placed on undefined or unspecified behavior. by interval analysis to be `0`, and it is used as a divisor. The warning addresses the possibility that the division may invoke a `divide by zero' runtime error.

Coding standards

CERT INT33-C

> Ensure that division and modulo operations do not result in divide-by-zero errors

CWE 369

> Divide By Zero

MISRA C:2004 1.2

> (Required) No reliance shall be placed on undefined or unspecified behavior.

Code examples

The following code example fails the check and will give a warning:

```
int foo(void)
{
  int a = 1;
  a--;
  return 5 / a;   /* a is 0 */
}
```

The following code example passes the check and will not give a warning about this issue:

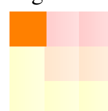
```
int foo(void)
{
  int a = 2;
  a--;
  return 5 / a;   /* OK - a is 1 */
}
```

MISRAC2004-1.2_h

Synopsis
An expression that may be `0` is used as a divisor.

Enabled by default
Yes

Severity/Certainty
High/Low



Full description
(Required) No reliance shall be placed on undefined or unspecified behavior. divisor, and its value, as determined by interval analysis contains `0`. If this code executes, a `divide by zero' runtime error may occur.

Coding standards
CERT INT33-C

Ensure that division and modulo operations do not result in divide-by-zero errors

CWE 369

Divide By Zero

MISRA C:2004 1.2

(Required) No reliance shall be placed on undefined or unspecified behavior.

Code examples
The following code example fails the check and will give a warning:

```
int main (void)

{

  int x = 2;

  int i;



  /* The second iteration leads to a division by zero*/

  for (i = 1; i < 3; i++) { x = x / (2 - i); }
/*@@ZDV-RED@@ */



  return x;

}

int foo(void)
{
  int a = 3;
  a--;
  return 5 / (a-2);  // a-2 is 0
}
```

The following code example passes the check and will not give a warning about this issue:

```
int foo(void)
{
  int a = 3;
  a--;
  return 5 / (a+2);  // OK - a+2 is 4
}
```

MISRAC2004-1.2_i

Synopsis A global variable is not checked against `0` before it is used as a divisor.

Enabled by default Yes

Severity/Certainty

Medium/Low

Full description

(Required) No reliance shall be placed on undefined or unspecified behavior. If the variable has a value of 0, then a `divide by zero' runtime error will occur.

Coding standards

CWE 369

Divide By Zero

MISRA C:2004 1.2

(Required) No reliance shall be placed on undefined or unspecified behavior.

Code examples

The following code example fails the check and will give a warning:

```
int x;

int example() {
  return 5/x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int x;

int example() {
  if (x != 0){
    return 5/x;
  }
}
```

MISRAC2004-1.2_j

Synopsis

A local variable is not checked against `0` before it is used as a divisor.

Enabled by default

Yes

Severity/Certainty

Medium/Low

Full description

(Required) No reliance shall be placed on undefined or unspecified behavior. If the variable has a value of 0, then a `divide by zero' runtime error will occur.

Coding standards

CWE 369

Divide By Zero

MISRA C:2004 1.2

(Required) No reliance shall be placed on undefined or unspecified behavior.

Code examples

The following code example fails the check and will give a warning:

```
int rand();

int example() {
    int x = rand();
    return 5/x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int rand();

int example() {
  int x = rand();
  if (x != 0){
    return 5/x;
  }
}
```

MISRAC2004-2.1

Synopsis

Inline asm statements that are not encapsulated in functions

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Assembler language shall be encapsulated and isolated.

Coding standards

MISRA C:2004 2.1

(Required) Assembler language shall be encapsulated and isolated.

Code examples

The following code example fails the check and will give a warning:

```
int ffs(int x)
{
        int r;
#if 0
#ifdef CONFIG_X86_64
        /*
         * AMD64 says BSFL won't clobber the dest reg if x==0;
Intel64 says the
         * dest reg is undefined if x==0, but their CPU architect
says its
         * value is written to set it to the same as before,
except that the
         * top 32 bits will be cleared.
         *
         * We cannot do this on 32 bits because at the very least
some
         * CPUs did not behave this way.
         */
        long tmp = -1;
        asm("bsfl %1,%0"
            : "=r" (r)
            : "rm" (x), "" (tmp));
#elif defined(CONFIG_X86_CMOV)
        asm("bsfl %1,%0\n\t"
            "cmovzl %2,%0"
            : "=&r" (r) : "rm" (x), "r" (-1));
#else
        asm("bsfl %1,%0\n\t"
            "jnz 1f\n\t"
            "movl $-1,%0\n"
            "1:" : "=r" (r) : "rm" (x));
#endif
#else
        asm("");
#endif
        return r + 1;
}
```

The following code example passes the check and will not give a warning about this issue:

```
unsigned int
bswap(unsigned int x)
{
    asm("");
    return x;
}
```

MISRAC2004-2.2

Synopsis

Uses of // comments

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

(Required) Source code shall only use /* ... */ style comments. These comments are not permitted by C90. The use of // in preprocessor directives (e.g. #define) can vary. The mixing of /* ... */ and // is inconsistent. In addition, different (pre C99) compilers may behave differently.

Coding standards

MISRA C:2004 2.2

(Required) Source code shall only use /* ... */ style comments.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  // an end of line comment
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  /* a terminated comment */
}
```

MISRAC2004-2.3

Synopsis

Appearances of /* inside comments

Enabled by default Yes

Severity/Certainty Low/High

Full description (Required) The character sequence /* shall not be used within a comment. Consider: /* A comment, end comment marker accidentally omitted <<New Page>> initialise(X); /* this comment is not compliant */ In this case, X will not be initialised because the code is hidden in a comment.

Coding standards MISRA C:2004 2.3

(Required) The character sequence /* shall not be used within a comment.

Code examples The following code example fails the check and will give a warning:

```
void example(void) {
  /* This comment starts here
  /* Nested comment starts here
  */
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  /* This comment starts here */
  /* Nested comment starts here
  */
}
```

MISRAC2004-2.4

Synopsis To allow comments to contain pseudo-code or code samples, only comments that end in ';', '{', or '}' characters are considered to be commented-out code.

Enabled by default No

Severity/Certainty Low/Medium

Full description

(Advisory) Sections of code should not be commented out. Code sections in comments are identified where the comment ends in ';', '{', or '}' characters.

Coding standards

MISRA C:2004 2.4

(Advisory) Sections of code should not be commented out.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
   /*
   int i;
   */
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
#if 0
   int i;
#endif
}
```

MISRAC2004-4.2

Synopsis

Uses of trigraphs (in string literals only)

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Tri-graphs shall not be used

Coding standards

MISRA C:2004 4.2

(Required) Tri-graphs shall not be used

Code examples

The following code example fails the check and will give a warning:

```
void func()
{
  char * str = "abc??!def";
}
```

The following code example passes the check and will not give a warning about this issue:

```
void func()
{
  char * str = "abc??def";
}
```

MISRAC2004-5.2_a

Synopsis

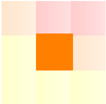
The definition of a local variable hides a global definition.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier. This may be intentional, but a different name should be used in case a reference to the global variable is attempted, and the local value changed or returned accidentally.

Coding standards

CERT DCL01-C

Do not reuse variable names in subscopes

CERT DCL01-CPP

Do not reuse variable names in subscopes

MISRA C:2004 5.2

(Required) Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.

Code examples

The following code example fails the check and will give a warning:

```
int x;


int foo (int y ){
  int x=0;
  x++;
  return x+y;

}
```

The following code example passes the check and will not give a warning about this issue:

```
int x;


int foo (int y ){

  x++;
  return x+y;

}
```

MISRAC2004-5.2_b

Synopsis

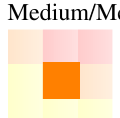
The definition of a local variable hides a previous local definition.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier. This may be intentional, but a different name should be used in case a reference to the outer variable is attempted, and the inner value changed or returned accidentally.

Coding standards

CERT DCL01-C

Do not reuse variable names in subscopes

CERT DCL01-CPP

Do not reuse variable names in subscopes

MISRA C:2004 5.2

(Required) Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.

Code examples

The following code example fails the check and will give a warning:

```
int foo(int x ){

  for (int y= 0; y < 10 ; y++){

    for (int y = 0; y < 100; y ++){
      return x+y;
    }
  }
  return x;
}


int foo2(int x){
  int y = 10;

  for (int y= 0; y < 10 ; y++)
    x++;
    return x;
}

int foo3(int x){

  int y = 10;
  {
    int y = 100;
    return x + y;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
int foo(int x){

  for (int y=0; y < 10; y++)
    x++;
  for (int y=0; y < 10; y++)
    x++;
  return x;
}
```

MISRAC2004-5.2_c

Synopsis

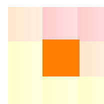
A variable declaration hides a parameter of the function

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier. This may be intentional, but a different name should be used in case a reference to the argument is attempted, and the inner value changed or returned accidentally.

Coding standards

CERT DCL01-C

> Do not reuse variable names in subscopes

CERT DCL01-CPP

> Do not reuse variable names in subscopes

MISRA C:2004 5.2

> (Required) Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.

Code examples

The following code example fails the check and will give a warning:

```
int foo(int x){


  for (int x = 0; x < 100; x++);

  return x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int foo(int x){
  int y;

  return x;
}
```

MISRAC2004-5.3

Synopsis

Typedef with this name already declared.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) A typedef name shall be a unique identifier.

Coding standards

MISRA C:2004 5.3

(Required) A typedef name shall be a unique identifier.

Code examples

The following code example fails the check and will give a warning:

```
typedef int WIDTH;
//dummy comment
void f1()
{
  WIDTH w1;
}

void f2()
{
  typedef float WIDTH;
  WIDTH w2;
  WIDTH w3;
}
```

The following code example passes the check and will not give a warning about this issue:

```
namespace NS1
{
  typedef int WIDTH;
}
// f2.cc
namespace NS2
{
  typedef float WIDTH; // Compliant - NS2::WIDTH is not the same 
as NS1::WIDTH
}
NS1::WIDTH w1;
NS2::WIDTH w2;
```

MISRAC2004-5.4

Synopsis

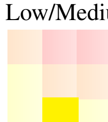
A class, struct, union or enum declaration that clashes with a previous declaration.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) A tag name shall be a unique identifier.

Coding standards

MISRA C:2004 5.4

(Required) A tag name shall be a unique identifier.

Code examples

The following code example fails the check and will give a warning:

```
void f1()
{
  class TYPE {};
}

void f2()
{
  float TYPE; // non-compliant
}
```

The following code example passes the check and will not give a warning about this issue:

```
enum ENS {ONE, TWO };

void f1()
{
  class TYPE {};
}

void f4()
{
  union GRRR {
    int i;
    float f;
  };
}
```

MISRAC2004-5.5

Synopsis A identifier is used that can clash with another static identifier.

Enabled by default No

Severity/Certainty Low/Medium

Full description (Advisory) No object or function identifier with static storage duration should be reused.

Coding standards MISRA C:2004 5.5

(Advisory) No object or function identifier with static storage duration should be reused.

Code examples The following code example fails the check and will give a warning:

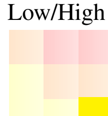
```
namespace NS1
{
  static int global = 0;
}

namespace NS2
{
  void fn()
  {
    int global; // Non-compliant
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
namespace NS1
{
  int global = 0;
}

namespace NS2
{
  void f1()
  {
    int global; // Non-compliant
  }
}

void f2()
{
  static int global;
}
```

MISRAC2004-6.1

Synopsis	Arithmetic on objects of type plain char, without an explicit signed or unsigned qualifier
Enabled by default	Yes
Severity/Certainty	Low/High



Full description

(Required) The plain char type shall be used only for the storage and use of character values. such types explicitly as "signed char" or "unsigned char", to avoid unportable behavior.

Coding standards

CERT INT07-C

Use only explicitly signed or unsigned char type for numeric values

MISRA C:2004 6.1

(Required) The plain char type shall be used only for the storage and use of character values.

Code examples

The following code example fails the check and will give a warning:

```
typedefsigned charINT8;
typedefunsigned charUINT8;

UINT8
toascii(INT8 c)
{
    return (UINT8)c & 0x7f;
}

int func(int x)
{
    char sc = 4;
    char *scp = &sc;
    UINT8 (*fp)(INT8 c) = &toascii;

    x = x + sc;
    x *= *scp;
    return (*fp)(x);
}
```

The following code example passes the check and will not give a warning about this issue:

```
typedefsigned charINT8;
typedefunsigned charUINT8;

UINT8
toascii(INT8 c)
{
    return (UINT8)c & 0x7f;
}

int func(int x)
{
    signed char sc = 4;
    signed char *scp = &sc;
    UINT8 (*fp)(INT8 c) = &toascii;

    x = x + sc;
    x *= *scp;
    return (*fp)(x);
}
```

MISRAC2004-6.3

Synopsis

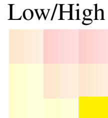
Uses of basic types char, int, short, long, double, and float without typedef

Enabled by default

No

Severity/Certainty

Low/High



Full description

(Advisory) typedefs that indicate size and signedness should be used in place of the basic types. Best practice is to use typedefs for portability.

Coding standards

MISRA C:2004 6.3

> (Advisory) typedefs that indicate size and signedness should be used in place of the basic types.

Code examples

The following code example fails the check and will give a warning:

```
typedef signed charSCHAR;
typedef intINT;
typedef floatFLOAT;

INT func(FLOAT f, INT *pi)
{
    INT x;
    INT (*fp)(const char *);
}
```

The following code example passes the check and will not give a warning about this issue:

```
typedef signed charSCHAR;
typedef intINT;
typedef floatFLOAT;

INT func(FLOAT f, INT *pi)
{
    INT x;
    INT (*fp)(const SCHAR *);
}
```

MISRAC2004-6.4

Synopsis

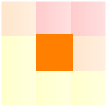
Bitfields with plain int type

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Bitfields shall only be defined to be of type unsigned int or signed int.

Coding standards

MISRA C:2004 6.4

> (Required) Bitfields shall only be defined to be of type unsigned int or signed int.

Code examples

The following code example fails the check and will give a warning:

```
struct bad {
  int x:3;
};
enum digs { ONE, TWO, THREE, FOUR };

struct bad {
  digs d:3;
};
```

The following code example passes the check and will not give a warning about this issue:

```
struct good {
  signed int x:3;
};
struct good {
  unsigned int x:3;
};
```

MISRAC2004-6.5

Synopsis

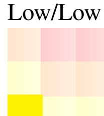
Signed single-bit fields (excluding anonymous fields)

Enabled by default

Yes

Severity/Certainty

Low/Low



Full description

(Required) Bitfields of signed type shall be at least 2 bits long.

Coding standards

MISRA C:2004 6.5

(Required) Bitfields of signed type shall be at least 2 bits long.

Code examples

The following code example fails the check and will give a warning:

```
struct S
{
  signed int a : 1; // Non-compliant
};
```

The following code example passes the check and will not give a warning about this issue:

```
struct S
{
  signed int b : 2;
  signed int   : 0;
  signed int   : 1;
  signed int   : 2;
};
```

MISRAC2004-7.1

Synopsis
Uses of octal integer constants

Enabled by default
Yes

Severity/Certainty
Low/Medium

Full description
(Required) Octal constants shall not be used. Zero is okay

Coding standards
MISRA C:2004 7.1

(Required) Octal constants shall not be used. Zero is okay

Code examples
The following code example fails the check and will give a warning:

```
void
func(void)
{
    int x = 077;
}
```

The following code example passes the check and will not give a warning about this issue:

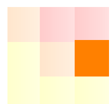
```
void
func(void)
{
    int x = 63;
}
```

MISRAC2004-8.1

Synopsis
Functions used without prototyping

Enabled by default
Yes

Severity/Certainty
Medium/High



Full description
(Required) Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.

Coding standards
CERT DCL31-C

Declare identifiers before using them

MISRA C:2004 8.1

(Required) Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.

Code examples
The following code example fails the check and will give a warning:

```
void func2(void)
{
    func();
}
```

The following code example passes the check and will not give a warning about this issue:

```
void func(void);
void func2(void)
{
    func();
}
```

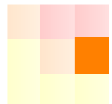
MISRAC2004-8.2

Synopsis
Whenever an object or function is declared or defined, its type shall be explicitly stated.

Enabled by default
Yes

Severity/Certainty

Medium/High



Full description

(Required) Whenever an object or function is declared or defined, its type shall be explicitly stated.

Coding standards

CERT DCL31-C

Declare identifiers before using them

MISRA C:2004 8.2

(Required) Whenever an object or function is declared or defined, its type shall be explicitly stated.

Code examples

The following code example fails the check and will give a warning:

```
void func(void)
{
    static y;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void func(void)
{
    int x;
}
```

MISRAC2004-8.5_a

Synopsis

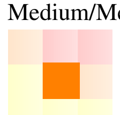
A header file shall not contain global variable.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) There shall be no definitions of objects or functions in a header file.

Coding standards

MISRA C:2004 8.5

(Required) There shall be no definitions of objects or functions in a header file.

Code examples

The following code example fails the check and will give a warning:

```
/*
global_def.h contains:
int global_variable;
 */
#include "global_def.h"
```

The following code example passes the check and will not give a warning about this issue:

```
/*
global_decl.h contains:
extern int global_variable;
*/
#include "global_decl.h"
```

MISRAC2004-8.5_b

Synopsis

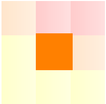
Non-inline functions defined in header files

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) There shall be no definitions of objects or functions in a header file. Header files should not be used to define functions. This makes it clear that only C source files contain executable code. A header file is defined to be any file that is included in a translation unit via the #include directive.

Coding standards

MISRA C:2004 8.5

(Required) There shall be no definitions of objects or functions in a header file.

Code examples

The following code example fails the check and will give a warning:

```
#include "definition.h"
/* Contents of definition.h:

void definition(void) {
}

*/

void example(void) {
  definition();
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include "declaration.h"
/* Contents of declaration.h:

void definition(void);

*/

void example(void) {
  definition();
}
```

MISRAC2004-8.12

Synopsis External arrays declared without size stated explicitly or defined implicitly by initialization.

Enabled by default Yes

Severity/Certainty Low/Medium

Full description (Required) When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.

Coding standards MISRA C:2004 8.12

> (Required) When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.

Code examples

The following code example fails the check and will give a warning:

```
extern int a[];
```

The following code example passes the check and will not give a warning about this issue:

```
extern int a[10];
extern int b[] = { 0, 1, 2 };
```

MISRAC2004-9.1_a

Synopsis

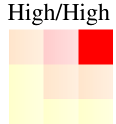
In all executions, a variable is read before it is assigned a value.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

(Required) All automatic variables shall have been assigned a value before being used. value. Different paths may result in reading a variable at different program points. Whichever path is executed, uninitialized data is read, and behavior may consequently be unpredictable.

Coding standards

CERT EXP33-C

Do not reference uninitialized memory

CWE 457

Use of Uninitialized Variable

MISRA C:2004 9.1

(Required) All automatic variables shall have been assigned a value before being used.

Code examples

The following code example fails the check and will give a warning:

```
int main(void) {
  int x;

  x++;  //x is uninitialized

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int main(void) {
  int x = 0;

  x++;

  return 0;
}
```

MISRAC2004-9.1_b

Synopsis

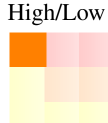
In some execution, a variable is read before it is assigned a value.

Enabled by default

Yes

Severity/Certainty

High/Low



Full description

(Required) All automatic variables shall have been assigned a value before being used. There may be some execution paths where the variable is assigned a value before it is read. In such cases behavior may be unpredictable.

Coding standards

CWE 457

Use of Uninitialized Variable

MISRA C:2004 9.1

(Required) All automatic variables shall have been assigned a value before being used.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

int main(void) {
  int x, y;

  if (rand()) {
    x = 0;
  }

  y = x;  //x may not be initialized

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

int main(void) {
  int x;

  if (rand()) {
    x = 0;
  }

  /* x never read */

  return 0;
}
```

MISRAC2004-9.1_c

Synopsis

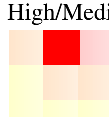
Dereference of an uninitialized or NULL pointer.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

(Required) All automatic variables shall have been assigned a value before being used. This will likely result in memory corruption or a program crash. Pointer values should always be initialized before being dereferenced, to avoid this.

Coding standards

CERT EXP33-C

Do not reference uninitialized memory

CWE 457

Use of Uninitialized Variable

CWE 824

Access of Uninitialized Pointer

MISRA C:2004 9.1

(Required) All automatic variables shall have been assigned a value before being used.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int *p;
  *p = 4;  //p is uninitialized
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int *p,a;
  p = &a;
  *p = 4;  //OK - p holds a valid address
}
```

MISRAC2004-9.2

Synopsis

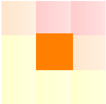
This check points out where a non-zero array initialisation does not exactly match the structure of the array declaration.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.

Coding standards

MISRA C:2004 9.2

> (Required) Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int y[3][4] = { { 1, 2, 3 }, { 4, 5, 6 } };
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int y[3][2] = { { 1, 2 }, { 3, 4 }, { 5, 6 } };
}
```

MISRAC2004-10.1_a

Synopsis

An expression of integer type is implicitly converted to a narrower or different sign underlying type

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The value of an expression of integer type shall not be implicitly converted to a different underlying type if: (a) it is not a conversion to a wider integer type of the same signedness.

Coding standards

MISRA C:2004 10.1

> (Required) The value of an expression of integer type shall not be implicitly converted to a different underlying type if: a. it is not a conversion to a wider integer type of the same signedness, or b. the expression is complex, or c. the expression is not constant and is a function argument, or d. the expression is not constant and is a return expression.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  long pc[10];
  // integer narrowing from int -> short
  short x = pc[5];
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int pc[10];
  long x = pc[5];
}
```

MISRAC2004-10.1_b

Synopsis

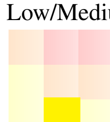
A complex expression of integer type is implicitly converted to a different underlying type.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) The value of an expression of integer type shall not be implicitly converted to a different underlying type if: (b) the expression is complex.

Coding standards

MISRA C:2004 10.1

> (Required) The value of an expression of integer type shall not be implicitly converted to a different underlying type if: a. it is not a conversion to a wider integer type of the same signedness, or b. the expression is complex, or c. the expression is not constant and is a function argument, or d. the expression is not constant and is a return expression.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int pc[10];
  // complex expression
  long long x = pc[5] + 5;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int pc[10];
  // complex expression without an implicit cast.
  int x = pc[5] + 5;
}
```

MISRAC2004-10.1_c

Synopsis

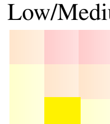
A non-constant expression of integer type is implicitly converted to a different underlying type in a function argument.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) The value of an expression of integer type shall not be implicitly converted to a different underlying type if: (c) the expression is not constant and is a function argument.

Coding standards

MISRA C:2004 10.1

> (Required) The value of an expression of integer type shall not be implicitly converted to a different underlying type if: a. it is not a conversion to a wider integer type of the same signedness, or b. the expression is complex, or c. the expression is not constant and is a function argument, or d. the expression is not constant and is a return expression.

Code examples

The following code example fails the check and will give a warning:

```
void function(long long argument);

void example(void) {
  int x = 4;
  function(x);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void function(long argument);

void example(void) {
   function(4);
}
```

MISRAC2004-10.1_d

Synopsis

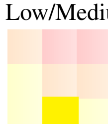
A non-constant expression of integer type is implicitly converted to a different underlying type in a return expression.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) The value of an expression of integer type shall not be implicitly converted to a different underlying type if: (d) the expression is not constant and is a return expression.

Coding standards

MISRA C:2004 10.1

> (Required) The value of an expression of integer type shall not be implicitly converted to a different underlying type if: a. it is not a conversion to a wider integer type of the same signedness, or b. the expression is complex, or c. the expression is not constant and is a function argument, or d. the expression is not constant and is a return expression.

Code examples

The following code example fails the check and will give a warning:

```
long long example(void) {
   int x = 4;
   return x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
long example(void) {
   return 4;
}
```

MISRAC2004-10.2_a

Synopsis
An expression of floating type is implicitly converted to a narrower underlying type

Enabled by default
Yes

Severity/Certainty
Low/Medium

Full description
(Required) The value of an expression of floating type shall not be implicitly converted to a different underlying type if: (a) it is not a conversion to a wider floating type.

Coding standards
MISRA C:2004 10.2

> (Required) The value of an expression of floating type shall not be implicitly converted to a different underlying type if: a. it is not a conversion to a wider floating type, or b. the expression is complex, or c. the expression is a function argument, or d. the expression is a return expression.

Code examples
The following code example fails the check and will give a warning:

```
#if __FLOAT_SIZE__ == __DOUBLE_SIZE__
  #error "IGNORE_TEST: double and float have same size"
#endif

void example(void) {
  double pc[10];
  // integer narrowing from double -> float
  float x = pc[5];
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  float pc[10];
  double x = pc[5];
}
```

MISRAC2004-10.2_b

Synopsis
An expression of floating type is implicitly converted to a narrower underlying type

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The value of an expression of floating type shall not be implicitly converted to a different underlying type if: (b) the expression is complex.

Coding standards

MISRA C:2004 10.2

> (Required) The value of an expression of floating type shall not be implicitly converted to a different underlying type if: a. it is not a conversion to a wider floating type, or b. the expression is complex, or c. the expression is a function argument, or d. the expression is a return expression.

Code examples

The following code example fails the check and will give a warning:

```
#if __FLOAT_SIZE__ == __DOUBLE_SIZE__
  #error "IGNORE_TEST: double and float have same size"
#endif

void example(void) {
   float pc[10];
   // complex expression
   double x = pc[5] + 5;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
   float pc[10];
   // complex expression without an implicit cast.
   float x = pc[5] + 5;
}
```

MISRAC2004-10.2_c

Synopsis

A non-constant expression of floating type is implicitly converted to a different underlying type in a function argument.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The value of an expression of floating type shall not be implicitly converted to a different underlying type if: (c) the expression is not constant and is a function argument.

Coding standards

MISRA C:2004 10.2

> (Required) The value of an expression of floating type shall not be implicitly converted to a different underlying type if: a. it is not a conversion to a wider floating type, or b. the expression is complex, or c. the expression is a function argument, or d. the expression is a return expression.

Code examples

The following code example fails the check and will give a warning:

```
#if __FLOAT_SIZE__ == __DOUBLE_SIZE__
  #error "IGNORE_TEST: double and float have same size"
#endif

void function(double argument);

void example(void) {
   float x = 4;
   function(x);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void function(double argument);

void example(void) {
   function(4.0);
}
```

MISRAC2004-10.2_d

Synopsis

A non-constant expression of floating type is implicitly converted to a different underlying type in a return expression.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The value of an expression of floating type shall not be implicitly converted to a different underlying type if: (d) the expression is not constant and is a return expression.

Coding standards

MISRA C:2004 10.2

> (Required) The value of an expression of floating type shall not be implicitly converted to a different underlying type if: a. it is not a conversion to a wider floating type, or b. the expression is complex, or c. the expression is a function argument, or d. the expression is a return expression.

Code examples

The following code example fails the check and will give a warning:

```
#if __FLOAT_SIZE__ == __DOUBLE_SIZE__
  #error "IGNORE_TEST: double and float have same size"
#endif

double example(void) {
  float x = 4;
  return x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
double example(void) {
  return 4.0;
}
```

MISRAC2004-10.3

Synopsis

A complex expression of integer type is cast to a wider or different sign underlying type.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The value of a complex expression of integer type shall only be cast to a type that is not wider and of the same signedness as the underlying type of the expression.

Coding standards

MISRA C:2004 10.3

> (Required) The value of a complex expression of integer type shall only be cast to a type that is not wider and of the same signedness as the underlying type of the expression.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int array[10];
  // complex expression cannot change sign
  unsigned int x = (unsigned int)(array[5] + 5);
}
void example(void) {
  int s16a = 3;
  int s16b = 3;

  // arithmetic makes it a complex expression
  long long x = (long long)(s16a + s16b);
}
void example(void) {
  int array[10];
  // complex expression cannot change type
  float x = (float)(array[5] + 5);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int array[10];
  // non-complex expression can change type
  float x = (float)(array[5]);
}
void example(void) {
  int array[10];

  // A non complex expression is considered safe
  long x = (long)(array[5]);
}
void example(void) {
  int array[10];

  // non-complex expressions can change sign
  unsigned int x = (unsigned int)(array[5]);
}
```

MISRAC2004-10.4

Synopsis

A complex expression of floating type is cast to a wider or different underlying type.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The value of a complex expression of floating type shall only be cast to a floating type which is narrower or of the same size.

Coding standards

MISRA C:2004 10.4

> (Required) The value of a complex expression of floating type shall only be cast to a floating type which is narrower or of the same size.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
   float array[10];
   // complex expression cannot change type
   int x = (int)(array[5] + 5.0f);
}
#if __FLOAT_SIZE__ == __DOUBLE_SIZE__
  #error "IGNORE_TEST: double and float have same size"
#endif

void example(void) {
   float array[10];

   // arithmetic makes it a complex expression
   double x = (double)(array[5] + 3.0f);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
   float array[10];

   // A non complex expression is considered safe
   double x = (double)(array[5]);
}
void example(void) {
   float array[10];
   // non-complex expression can change type
   int x = (int)(array[5]);
}
```

MISRAC2004-10.5

Synopsis	Bitwise operation on unsigned char or unsigned short, that are not immediately cast to this type to ensure consistent truncation
Enabled by default	Yes
Severity/Certainty	Low/Medium

Full description

(Required) If the bitwise operators ~ and << are applied to an operand of underlying type unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.

Coding standards

MISRA C:2004 10.5

(Required) If the bitwise operators ~ and << are applied to an operand of underlying type unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.

Code examples

The following code example fails the check and will give a warning:

```
typedef unsigned char uint8_t;
typedef unsigned short uint16_t;

void example(void) {
  uint8_t port = 0x5aU;
  uint8_t result_8;
  uint16_t result_16;
  uint16_t mode;

  result_8 = (~port) >> 4;
}
typedef unsigned char uint8_t;
typedef unsigned short uint16_t;

void example(void) {
  uint8_t port = 0x5aU;
  uint8_t result_8;
  uint16_t result_16;
  uint8_t mode;

  result_16 = ((port << 4) & mode) >> 6;
}
```

The following code example passes the check and will not give a warning about this issue:

```
typedef unsigned char uint8_t;
typedef unsigned short uint16_t;

void example(void) {
  uint8_t port = 0x5aU;
  uint8_t result_8;
  uint16_t result_16;
  uint16_t mode;

  result_16 = ((uint16_t)((uint16_t)port << 4) & mode) >> 6;
}
typedef unsigned char uint8_t;
typedef unsigned short uint16_t;

void example(void) {
  uint8_t port = 0x5aU;
  uint8_t result_8;
  uint16_t result_16;
  uint16_t mode;

  result_8 = ((uint8_t)(~port)) >> 4;
  result_16 = ((uint16_t)(~(uint16_t)port)) >> 4;
}
```

MISRAC2004-10.6

Synopsis

A U suffix shall be applied to all constants of unsigned type.

Enabled by default

Yes

Severity/Certainty

Low/Low

Full description

(Required) A U suffix shall be applied to all constants of unsigned type.

Coding standards

MISRA C:2004 10.6

(Required) A U suffix shall be applied to all constants of unsigned type.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  // 2147483648 -- does not fit in 31bits
  unsigned int x = 0x80000000;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  unsigned int x = 0x80000000u;
}
```

MISRAC2004-11.1

Synopsis

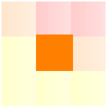
Conversions shall not be performed between a pointer to a function and any type other than an integral type.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Conversions shall not be performed between a pointer to a function and any type other than an integral type.

Coding standards

MISRA C:2004 11.1

> (Required) Conversions shall not be performed between a pointer to a function and any type other than an integral type.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

void example(void) {

  int (*fptr)(int,int);

  (int*)fptr;

}
```

The following code example passes the check and will not give a warning about this issue:

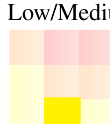
```
#include <stdlib.h>

void example(void) {

  int (*fptr)(int,int);

  (int )fptr;

}
```

MISRAC2004-11.3

Synopsis A cast should not be performed between a pointer type and an integral type.

Enabled by default No

Severity/Certainty Low/Medium



Full description (Advisory) A cast should not be performed between a pointer type and an integral type.

Coding standards MISRA C:2004 11.3

(Advisory) A cast should not be performed between a pointer type and an integral type.

Code examples The following code example fails the check and will give a warning:

```
void example(void) {

  int *p;
  int x;

  x = (int)p;

}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {

   int *p;
   int *x;

   x = p;

}
```

MISRAC2004-11.4

Synopsis

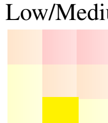
A pointer to object type is cast to a pointer to different object type

Enabled by default

No

Severity/Certainty

Low/Medium



Full description

(Advisory) A cast should not be performed between a pointer to object type and a different pointer to object type. Conversions of this type may be invalid if the new pointer type required a stricter alignment.

Coding standards

MISRA C:2004 11.4

> (Advisory) A cast should not be performed between a pointer to object type and a different pointer to object type.

Code examples

The following code example fails the check and will give a warning:

```
typedef unsigned int uint32_t;
typedef unsigned char uint8_t;

void example(void) {
   uint8_t * p1;
   uint32_t * p2;
   p2 = (uint32_t *)p1;
}
```

The following code example passes the check and will not give a warning about this issue:

```
typedef unsigned int uint32_t;
typedef unsigned char uint8_t;

void example(void) {
  uint8_t * p1;
  uint8_t * p2;
  p2 = (uint8_t *)p1;
}
```

MISRAC2004-11.5

Synopsis

Casts that remove any const or volatile qualification.

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

(Required) A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer. This violates the principle of type qualification. This check does not look for changes to the qualification of the pointer during the cast.

Coding standards

MISRA C:2004 11.5

> (Required) A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer.

Code examples

The following code example fails the check and will give a warning:

```
typedef unsigned short uint16_t;

void example(void) {

  uint16_t x;
  const uint16_t *    pci;      /* pointer to const int */
  uint16_t *          pi;       /* pointer to int */

  pi = (uint16_t *)pci; // not compliant

}
```

The following code example passes the check and will not give a warning about this issue:

```
typedef unsigned short uint16_t;

void example(void) {

  uint16_t x;
  uint16_t * const    cpi = &x; /* const pointer to int */
  uint16_t *          pi;       /* pointer to int */

  pi = cpi; // compliant - no cast required

}
```

MISRAC2004-12.1

Synopsis

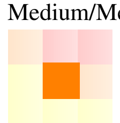
Add parentheses to avoid implicit operator precedence.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

(Advisory) Limited dependence should be placed on the C operator precedence rules in expressions.

Coding standards

MISRA C:2004 12.1

> (Advisory) Limited dependence should be placed on the C operator precedence rules in expressions.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
    int i;
    int j;
    int k;
    int result;

    result = i + j * k;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
    int i;
    int j;
    int k;
    int result;

    result = i + (j - k);
}
```

MISRAC2004-12.2_a

Synopsis

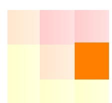
Expressions which depend on order of evaluation

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

(Required) The value of an expression shall be the same under any order of evaluation that the standard permits. expression with an unspecified evaluation order, between two consecutive sequence points. ANSI C does not specify an evaluation order for different parts of an expression. For this reason different compilers are free to perform their own optimizations regarding the evaluation order. Projects containing statements that violate this check are not readily ported between architectures or compilers, and their ports may prove difficult to debug. Only four operators have a guaranteed order of evaluation: logical AND (`a && b`) evaluates the left operand, then the right operand only if the left is found to be true; logical OR (`a || b`) evaluates the left operand, then the right operand only if the left is found to be false; a ternary conditional (`a ? b : c`) evaluates the first operand, then either the second or the third, depending on whether the first is found to be true or false; and a comma (`a , b`) evaluates its left operand before its right.

Coding standards

CERT EXP10-C

Do not depend on the order of evaluation of subexpressions or the order in which side effects take place

CERT EXP30-C

Do not depend on order of evaluation between sequence points

CWE 696

Incorrect Behavior Order

MISRA C:2004 12.2

(Required) The value of an expression shall be the same under any order of evaluation that the standard permits.

Code examples

The following code example fails the check and will give a warning:

```
int main(void) {
  int i = 0;

  i = i * i++;  //unspecified order of operations

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int main(void) {
  int i = 0;
  int x = i;

  i++;
  x = x * i;  //OK - statement is broken up

  return 0;
}
```

MISRAC2004-12.2_b

Synopsis

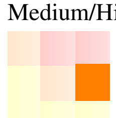
There shall be no more than one read access with volatile-qualified type within one sequence point

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

(Required) The value of an expression shall be the same under any order of evaluation that the standard permits.

Coding standards

CERT EXP10-C

Do not depend on the order of evaluation of subexpressions or the order in which side effects take place

CERT EXP30-C

Do not depend on order of evaluation between sequence points

CWE 696

Incorrect Behavior Order

MISRA C:2004 12.2

(Required) The value of an expression shall be the same under any order of evaluation that the standard permits.

Code examples

The following code example fails the check and will give a warning:

```
#include "mc2_types.h"
#include "mc2_header.h"

void example(void) {
  uint16_t x;
  volatile uint16_t v;
  x = v + v;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int main(void) {
  int i = 0;
  int x = i;

  i++;
  x = x * i;  //OK - statement is broken up

  return 0;
}
```

MISRAC2004-12.2_c

Synopsis

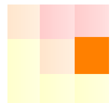
There shall be no more than one modification access with volatile-qualified type within one sequence point

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

(Required) The value of an expression shall be the same under any order of evaluation that the standard permits.

Coding standards

CERT EXP10-C

> Do not depend on the order of evaluation of subexpressions or the order in which side effects take place

CERT EXP30-C

> Do not depend on order of evaluation between sequence points

CWE 696

> Incorrect Behavior Order

MISRA C:2004 12.2

> (Required) The value of an expression shall be the same under any order of evaluation that the standard permits.

Code examples

The following code example fails the check and will give a warning:

```
#include "mc2_types.h"
#include "mc2_header.h"

void example(void) {
  uint16_t x;
  volatile uint16_t v, w;
  v = w = x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdbool.h>
void InitializeArray(int *);
const int *example(void)
{
  static volatile bool s_initialized = false;
  static int s_array[256];

  if (!s_initialized)
  {
    InitializeArray(s_array);
    s_initialized = true;
  }
  return s_array;
}
```

MISRAC2004-12.3

Synopsis

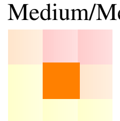
Sizeof expressions containing side effects

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The sizeof operator shall not be used on expressions that contain side effects. side effects. The expectation of the programmer might be that the expression will be evaluated. However because sizeof only operates on the type of the expression, the expression itself is not evaluated.

Coding standards

CERT EXP06-C

Operands to the sizeof operator should not contain side effects

CERT EXP06-CPP

Operands to the sizeof operator should not contain side effects

MISRA C:2004 12.3

(Required) The sizeof operator shall not be used on expressions that contain side effects.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int i;
  int size = sizeof(i++);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int i;
  int size = sizeof(i);
  i++;
}
```

MISRAC2004-12.4

Synopsis

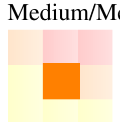
Right hand operands of && or || that contain side effects

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The right-hand operand of a logical && or || operator shall not contain side effects.

Coding standards

CWE 768

Incorrect Short Circuit Evaluation

MISRA C:2004 12.4

(Required) The right-hand operand of a logical && or || operator shall not contain side effects.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int i;
  int size = rand() && i++;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int i;
  int size = rand() && i;
}
```

MISRAC2004-12.6_a

Synopsis

Operands of logical operators (&&, ||, and !) that are not effectively Boolean.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

(Advisory) The operands of logical operators (&&, ||, and !) should be effectively boolean.

Coding standards

MISRA C:2004 12.6

> (Advisory) The operands of logical operators (&&, ||, and !) should be effectively boolean. Expressions that are effectively boolean should not be used as operands to operators other than (&&, ||, !, =, ==, !=, and ?:).

Code examples

The following code example fails the check and will give a warning:

```
void func(int * ptr)
{
  if (!ptr) {}
}
void func()
{
  if (!0) {}
}
void example(void) {
  int x = 0;
  int y = 1;
  int a = x || y << 2;
}
void example(void) {
  int x = 0;
  int y = 1;
  int a = 5;
  (a + (x || y)) ? example() : example();
}
void example(void) {
  int x = 5;
  int y = 11;
  if (x || y) {
  }
}
void example(void) {

  int d, c, b, a;

  d = ( c & a ) && b;

}
```

The following code example passes the check and will not give a warning about this issue:

```
bool test()
{
  return true;
}

void example(void) {
  if(test()) {}
}
typedef charboolean_t;/* Compliant: Boolean-by-enforcement */

void example(void)
{
    boolean_t d;
    boolean_t c = 1;
    boolean_t b = 0;
    boolean_t a = 1;

    d = ( c && a ) && b;

}
void func(bool * ptr)
{
  if (*ptr) {}
}
typedef intboolean_t;

void example(void) {
    boolean_t x = 0;
    boolean_t y = 1;
    boolean_t a = 0;
    if (a && (x || y)) {
    }
}
void example(void) {
  int x = 0;
  int y = 1;
  int a = x == y;
}
#include <stdbool.h>

void example(void) {
    bool x = false;
    bool y = true;
    if (x || y) {
    }
}
typedef charboolean_t;
```

```
void example(void) {
    boolean_t x = 0;
    boolean_t y = 1;
    boolean_t a = x || y;
    a ? example() : example();
}
```

MISRAC2004-12.6_b

Synopsis

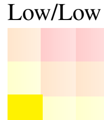
Uses of arithmetic operators on boolean operands.

Enabled by default

No

Severity/Certainty

Low/Low



Full description

(Advisory) Expressions that are effectively boolean should not be used as operands to operators other than (&&, ||, !, =, ==, !=, and ?:).

Coding standards

MISRA C:2004 12.6

> (Advisory) The operands of logical operators (&&, ||, and !) should be effectively boolean. Expressions that are effectively boolean should not be used as operands to operators other than (&&, ||, !, =, ==, !=, and ?:).

Code examples

The following code example fails the check and will give a warning:

```
void func(bool b)
{
  bool x;
  bool y;
  y = x % b;
}
void example(void) {
   int x = 0;
   int y = 1;
   int a = 5;
   (a + (x || y)) ? example() : example();
}
void example(void) {
   int x = 0;
   int y = 1;
   int a = (x == y) << 2;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int
isgood(int ch)
{
    return (ch & 0x80) == 0;
}

int example(int r, int f1, int f2)
{
  if (r && f1 == f2)
    return 1;
  else
    return 0;
}
bool test()
{
  return true;
}

void example(void) {
  if(test()) {}
}
typedef charboolean_t;/* Compliant: Boolean-by-enforcement */

void example(void)
{
    boolean_t d;
    boolean_t c = 1;
    boolean_t b = 0;
    boolean_t a = 1;

    d = ( c && a ) && b;

}
class foo {
  int val;
public:
  bool operator==(const foo &rhs) const { return val == rhs.val;
}
};

int example(bool r, const foo &f1, const foo &f2)
{
  if (r && f1 == f2)
    return 1;
  else
    return 0;
}
```

```
void func(bool * ptr)
{
  if (*ptr) {}
}
void func()
{
  bool x;
  bool y;
  y = x && y;
}
typedef intboolean_t;

void example(void) {
    boolean_t x = 0;
    boolean_t y = 1;
    boolean_t a = 0;
    if (a && (x || y)) {
    }
}
void example(void) {
   int x = 0;
   int y = 1;
   int a = x == y;
}
#include <stdbool.h>

void example(void) {
    bool x = false;
    bool y = true;
    if (x || y) {
    }
}
typedef charboolean_t;
void example(void) {
    boolean_t x = 0;
    boolean_t y = 1;
    boolean_t a = x || y;
    a ? example() : example();
}
```

MISRAC2004-12.7

Synopsis	Applications of bitwise operators to signed operands
Enabled by default	Yes

Severity/Certainty

Low/Medium

Full description

(Required) Bitwise operators shall not be applied to operands whose underlying type is signed.

Coding standards

CERT INT13-C

Use bitwise operators only on unsigned operands

MISRA C:2004 12.7

(Required) Bitwise operators shall not be applied to operands whose underlying type is signed.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int x = -(1U);

  x ^ 1;
  x & 0x7F;
  ((unsigned int)x) & 0x7F;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int x = -1;
  ((unsigned int)x) ^ 1U;
  2U ^ 1U;
  ((unsigned int)x) & 0x7FU;
  ((unsigned int)x) & 0x7FU;
}
```

MISRAC2004-12.8

Synopsis

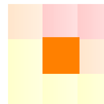
Out of range shifts

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand. In this case, the right-hand operand may be negative, or too large. This check is for all platforms. The behavior in this situation is undefined; the code may work as intended, or data could become erroneous.

Coding standards

CERT INT34-C

Do not shift a negative number of bits or more bits than exist in the operand

CWE 682

Incorrect Calculation

MISRA C:2004 12.8

(Required) The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand.

Code examples

The following code example fails the check and will give a warning:

```
unsigned int foo(unsigned long long x, unsigned int y)
{
  int shift = 65; // too big
  return 3ULL << shift;
}
unsigned int foo(unsigned int x, unsigned int y)
{
  int shift = 33; // too big
  return 3U << shift;
}
```

The following code example passes the check and will not give a warning about this issue:

```
unsigned int foo(unsigned int x)
{
  int y = 1;  // OK - this is within the correct range
  return x << y;
}
unsigned int foo(unsigned long long x)
{
  int y = 63; // ok
  return x << y;
}
```

MISRAC2004-12.9

Synopsis

Uses of unary - on unsigned expressions

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

Coding standards

MISRA C:2004 12.9

> (Required) The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  unsigned int max = -1U;
  // use max = ~0U;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int neg_one = -1;
}
```

MISRAC2004-12.10

Synopsis

Uses of the comma operator

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

(Required) The comma operator shall not be used.

Coding standards

MISRA C:2004 12.10

(Required) The comma operator shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
#include <string.h>

void reverse(char *string) {
  int i, j;
  j = strlen(string);
  for (i = 0; i < j; i++, j--) {
    char temp = string[i];
    string[i] = string[j];
    string[j] = temp;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <string.h>

void reverse(char *string) {
  int i;
  int length = strlen(string);
  int half_length = length / 2;
  for (i = 0; i < half_length; i++) {
    int opposite = length - i;
    char temp = string[i];
    string[i] = string[opposite];
    string[opposite] = temp;
  }
}
```

MISRAC2004-12.11

Synopsis

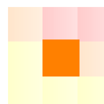
A constant unsigned integer expression overflows

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

(Advisory) Evaluation of constant unsigned integer expressions should not lead to wrap-around.

Coding standards

MISRA C:2004 12.11

> (Advisory) Evaluation of constant unsigned integer expressions should not lead to wrap-around.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
   (0xFFFFFFFF + 1u);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
   0x7FFFFFFF + 0;
}
```

MISRAC2004-12.12_a

Synopsis

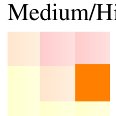
Reading from a field of a union following a write to a different field, effectively re-interpreting the bit pattern with a different type.

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

(Required) The underlying bit representations of floating-point values shall not be used. To reinterpret bit patterns deliberately, it is best to use an explicit cast.

Coding standards

CERT EXP39-C

Do not access a variable through a pointer of an incompatible type

CWE 188

Reliance on Data/Memory Layout

MISRA C:2004 12.12

(Required) The underlying bit representations of floating-point values shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
union name {
  int int_field;
  float float_field;
};

void example(void) {
  union name u;
  u.int_field = 10;
  float f = u.float_field;
}
```

The following code example passes the check and will not give a warning about this issue:

```
union name {
  int int_field;
  float float_field;
};

void example(void) {
  union name u;
  u.int_field = 10;
  float f = u.int_field;
}
```

MISRAC2004-12.12_b

Synopsis

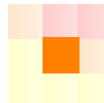
An expression provides access to the bit-representation of a floating point variable.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The underlying bit representations of floating-point values shall not be used.

Coding standards

MISRA C:2004 12.12

(Required) The underlying bit representations of floating-point values shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
void example(float f) {
  int * x = (int *)&f;
  int i = *x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(float f) {
  int i = (int)f;
}
```

MISRAC2004-12.13

Synopsis

Uses of increment (++) and decrement (--) operators mixed with other operators in an expression.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

(Advisory) The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.

Coding standards

MISRA C:2004 12.13

(Advisory) The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.

Code examples

The following code example fails the check and will give a warning:

```
void example(char *src, char *dst) {
  while ((*src++ = *dst++));
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(char *src, char *dst) {
  while (*src) {
    *dst = *src;
    src++;
    dst++;
  }
}
```

MISRAC2004-13.1

Synopsis

Assignment operators shall not be used in expressions that yield a boolean value.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Assignment operators shall not be used in expressions that yield a boolean value.

Coding standards

MISRA C:2004 13.1

(Required) Assignment operators shall not be used in expressions that yield a boolean value.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
   int result;
   if (result = condition()) {
   }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
   int result = condition();
   if (result) {
   }
}
```

MISRAC2004-13.2_a

Synopsis

Non-boolean termination conditions in do ... while statements.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

(Advisory) Tests of a value against zero should be made explicit, unless the operand is effectively boolean.

Coding standards

MISRA C:2004 13.2

> (Advisory) Tests of a value against zero should be made explicit, unless the operand is effectively boolean.

Code examples

The following code example fails the check and will give a warning:

```
typedefintint32_t;
int32_t func();

void example(void)
{
  do {
  } while (func());
}
```

The following code example passes the check and will not give a warning about this issue:

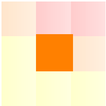
```
#include <stddef.h>

int * fn()
{
  int * ptr;
  return ptr;
}

int fn2()
{
  return 5;
}

bool fn3()
{
  return true;
}

void example(void)
{
  while (int *ptr = fn() )  // Compliant by exception
  {}

  do
  {
    int *ptr = fn();
    if ( NULL == ptr )
    {
      break;
    }
  }
  while (true); // Compliant

  while (int len = fn2() )  // Compliant by exception
  {}

  if (int *p = fn()) {}   // Compliant by exception
  if (int len = fn2() ) {} // Complioant by exception
  if (bool flag = fn3()) {} // Compliant
}
```

MISRAC2004-13.2_b

Synopsis
Non-boolean termination conditions in for loops.

Enabled by default
No

Severity/Certainty
Medium/Medium



Full description
(Advisory) Tests of a value against zero should be made explicit, unless the operand is effectively boolean.

Coding standards
MISRA C:2004 13.2

> (Advisory) Tests of a value against zero should be made explicit, unless the operand is effectively boolean.

Code examples
The following code example fails the check and will give a warning:

```
void example(void)
{
  for (int x = 10;x;--x) {}
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stddef.h>

int * fn()
{
  int * ptr;
  return ptr;
}

int fn2()
{
  return 5;
}

bool fn3()
{
  return true;
}

void example(void)
{
  for (fn(); fn3(); fn2())  // Compliant
  {}

  for (fn(); true; fn()) // Compliant
  {
    int *ptr = fn();
    if ( NULL == ptr )
    {
      break;
    }
  }

  for (int len = fn2(); len < 10; len++)  // Compliant
    ;
}
```

MISRAC2004-13.2_c

Synopsis	Non-boolean conditions in if statements.
Enabled by default	No

Severity/Certainty

Low/Medium

Full description

(Advisory) Tests of a value against zero should be made explicit, unless the operand is effectively boolean.

Coding standards

MISRA C:2004 13.2

> (Advisory) Tests of a value against zero should be made explicit, unless the operand is effectively boolean.

Code examples

The following code example fails the check and will give a warning:

```
void example(void)
{
  int u8;
  if (u8) {}
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stddef.h>

int * fn()
{
  int * ptr;
  return ptr;
}

int fn2()
{
  return 5;
}

bool fn3()
{
  return true;
}

void example(void)
{
  while (int *ptr = fn() )  // Compliant by exception
  {}

  do
  {
    int *ptr = fn();
    if ( NULL == ptr )
    {
      break;
    }
  }
  while (true); // Compliant

  while (int len = fn2() )  // Compliant by exception
  {}

  if (int *p = fn()) {}   // Compliant by exception
  if (int len = fn2() ) {} // Complioant by exception
  if (bool flag = fn3()) {} // Compliant
}
```

MISRAC2004-13.2_d

Synopsis	Non-boolean termination conditions in while statements.
Enabled by default	No

Severity/Certainty

Low/Medium

Full description

(Advisory) Tests of a value against zero should be made explicit, unless the operand is effectively boolean.

Coding standards

MISRA C:2004 13.2

> (Advisory) Tests of a value against zero should be made explicit, unless the operand is effectively boolean.

Code examples

The following code example fails the check and will give a warning:

```
void example(void)
{
  int u8;
  while (u8) {}
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stddef.h>

int * fn()
{
  int * ptr;
  return ptr;
}

int fn2()
{
  return 5;
}

bool fn3()
{
  return true;
}

void example(void)
{
  while (int *ptr = fn() )  // Compliant by exception
  {}

  do
  {
    int *ptr = fn();
    if ( NULL == ptr )
    {
      break;
    }
  }
  while (true); // Compliant

  while (int len = fn2() )  // Compliant by exception
  {}

  if (int *p = fn()) {}   // Compliant by exception
  if (int len = fn2() ) {} // Complioant by exception
  if (bool flag = fn3()) {} // Compliant
}
```

MISRAC2004-13.2_e

Synopsis	Non-boolean operands to the conditional (? :) operator
Enabled by default	No

Severity/Certainty

Low/Medium

Full description

(Advisory) Tests of a value against zero should be made explicit, unless the operand is effectively boolean.

Coding standards

MISRA C:2004 13.2

> (Advisory) Tests of a value against zero should be made explicit, unless the operand is effectively boolean.

Code examples

The following code example fails the check and will give a warning:

```
void example(int x) {
  int z;
  z = x ? 1 : 2;  //x is an int, not a bool
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(int x) {
  int z;
  z = x + 0 > 3 ? 1 : 2;  //OK - the condition is a comparison
}
void example(bool b) {
  int x;
  x = b ? 1 : 2;  //OK - b is a bool
}
```

MISRAC2004-13.3

Synopsis

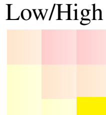
Floating point comparisons using == or !=

Enabled by default

Yes

Severity/Certainty

Low/High



Full description

(Required) Floating-point expressions shall not be tested for equality or inequality. The comparison will potentially be evaluated incorrectly, especially if either of the floats have been operated on arithmetically. In such a case, program logic will be compromised.

Coding standards

CERT FLP06-C

Understand that floating-point arithmetic in C is inexact

CERT FLP35-CPP

Take granularity into account when comparing floating point values

MISRA C:2004 13.3

(Required) Floating-point expressions shall not be tested for equality or inequality.

Code examples

The following code example fails the check and will give a warning:

```
int main(void)
{
  float f = 3.0;
  int i = 3;

  if (f == i) //comparison of a float and an int
    ++i;

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int main(void)
{
  int i = 60;
  char c = 60;

  if (i == c)
    ++i;

  return 0;
}
```

MISRAC2004-13.4

Synopsis

Floating-point values in the controlling expression of a for statement.

Enabled by default: Yes

Severity/Certainty: Low/Medium

Full description: (Required) The controlling expression of a for statement shall not contain any objects of floating type.

Coding standards: MISRA C:2004 13.4

(Required) The controlling expression of a for statement shall not contain any objects of floating type.

Code examples: The following code example fails the check and will give a warning:

```
void example(int input, float f) {
  int i;
  for (i = 0; i < input && f < 0.1f; ++i) {
  }
}
```

The following code example passes the check and will not give a warning about this issue:

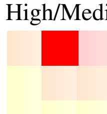
```
void example(int input, float f) {
  int i;
  int f_condition = f < 0.1f;
  for (i = 0; i < input && f_condition; ++i) {
    f_condition = f < 0.1f;
  }
}
```

MISRAC2004-13.5

Synopsis: A `for` loop counter variable is not initialized in the `for` loop.

Enabled by default: Yes

Severity/Certainty: High/Medium



Full description

(Required) The three expressions of a for statement shall be concerned only with loop control. been initialized in the `for` loop header. When a counter is used in a loop, it should be initialized. If not, the loop may iterate a very large number of times, or not at all. This check will not warn about uninitialized variables that are not used as counters.

Coding standards

MISRA C:2004 13.5

> (Required) The three expressions of a for statement shall be concerned only with loop control.

Code examples

The following code example fails the check and will give a warning:

```
int example(void) {
  int i, x = 10;

  /* 'i' used as a counter, not initialized */
  for ( ; i < 10; i++) {
    x++;
  }

  return x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(void) {
  int i, x = 10;

  /* 'i' initialized in loop header */
  for (i = 0; i < 10; i++) {
    x++;
  }

  return x;
}
```

MISRAC2004-13.6

Synopsis

A `for` loop counter variable is modified in the body of the loop.

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

(Required) Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop. statement) should not be assigned to in the body of the for loop. While it's legal to modify the loop counter within the body of a `for` loop (in place of a `while` loop), the conventional use of a `for` loop is to iterate over a predetermined range, incrementing the loop counter once per iteration. Modification of the loop counter within the `for` loop body is probably accidental, and could result in erroneous behavior or an infinite loop.

Coding standards

MISRA C:2004 13.6

> (Required) Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop.

Code examples

The following code example fails the check and will give a warning:

```
int main(void) {
  int i;

  /* i is incremented inside the loop body */
  for (i = 0; i < 10; i++) {
    i = i + 1;
  }

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int main(void) {
  int i;
  int x = 0;

  for (i = 0; i < 10; i++) {
    x = i + 1;
  }

  return 0;
}
```

MISRAC2004-13.7_a

Synopsis

A comparison using ==, <, <=, >, or >= is always true.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Boolean operations whose results are invariant shall not be permitted. the values of the arguments of the comparison operator. This often occurs because of literal values or macros having been used on one or both sides of the operator. Double-check that the operands and the code's logic are correct.

Coding standards

CWE 571

Expression is Always True

MISRA C:2004 13.7

(Required) Boolean operations whose results are invariant shall not be permitted.

Code examples

The following code example fails the check and will give a warning:

```
int example(void) {
  int x = 42;

  if (x == 42) {  //always true
    return 0;
  }

  return 1;

}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(void) {
  int x = 42;

  if (rand()) {
    x = 40;
  }

  if (x == 42) {  //OK - may not be true
    return 0;
  }

  return 1;

}
```

MISRAC2004-13.7_b

Synopsis

A comparison using ==, <, <=, >, or >= is always false.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Boolean operations whose results are invariant shall not be permitted. the values of the arguments of the comparison operator. This often occurs because of literal values or macros having been used on one or both sides of the operator. Double-check that the operands and the code's logic are correct.

Coding standards

CWE 570

Expression is Always False

MISRA C:2004 13.7

(Required) Boolean operations whose results are invariant shall not be permitted.

Code examples

The following code example fails the check and will give a warning:

```
int example(void) {
  int x = 10;

  if (x < 10) {  //never true
    return 1;
  }

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(int x) {

  if (x < 10) {  //OK - may be true
    return 1;
  }

  return 0;
}
```

MISRAC2004-14.1

Synopsis

In all executions, a part of the program is not executed.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) There shall be no unreachable code. Though not necessarily a problem, dead code can indicate programmer confusion about the program's branching structure.

Coding standards

CERT MSC07-C

Detect and remove dead code

CWE 561

Dead Code

MISRA C:2004 14.1

(Required) There shall be no unreachable code.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdio.h>

int f(int mode) {
    switch (mode) {
        case 0:
            return 1;
            printf("Hello!"); // This line cannot execute.
        default:
            return -1;
    }
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdio.h>

int f(int mode) {
    switch (mode) {
        case 0:
            printf("Hello!"); // This line can execute.
            return 1;
        default:
            return -1;
    }
}
```

MISRAC2004-14.2

Synopsis

A statement that potentially contains no side effects.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) All non-null statements shall either have at least one side effect however executed, or cause control flow to change.

Coding standards

CERT MSC12-C

Detect and remove code that has no effect

CWE 482

Comparing instead of Assigning

MISRA C:2004 14.2

(Required) All non-null statements shall either have at least one side effect however executed, or cause control flow to change.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int x = 1;
  x = 2;
  x < x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <string>
#include "iar.h"

void f();
template<class T>
struct X {
  int x;

  int get() const {
    return x;
  }

  X(int y) :
    x(y) {}

};

typedef X<int> intX;

void example(void) {
  /* everything below has a side-effect */
  int i=0;
  f();
  (void)f();
  ++i;
  i+=1;
  i++;
  char *p = "test";
  STD string s;
  s.assign(p);
  STD string *ps = &s;
  ps -> assign(p);
  intX xx(1);
  xx.get();
  intX(1);
}
```

MISRAC2004-14.3

Synopsis	Stray semicolons on the same line as other code
Enabled by default	Yes

Severity/Certainty

Low/Low

Full description

(Required) Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a whitespace character. by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character.

Coding standards

CERT EXP15-C

Do not place a semicolon on the same line as an if, for, or while statement

MISRA C:2004 14.3

(Required) Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a whitespace character.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int i;
  for (i=0; i!=10; ++i);  //Null statement as the
                          //body of this for loop
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int i;
  for (i=0; i!=10; ++i){  //An empty block is much
  }                       //more readable
}
```

MISRAC2004-14.4

Synopsis

Uses of goto.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The goto statement shall not be used.

Coding standards

MISRA C:2004 14.4

(Required) The goto statement shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {

  goto testin;

testin:
  printf("Reached by goto");

}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {

  printf ("Not reached by goto");

}
```

MISRAC2004-14.5

Synopsis

Uses of continue.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The continue statement shall not be used.

Coding standards

MISRA C:2004 14.5

(Required) The continue statement shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdio.h>

// Print the odd numbers between 0 and 99

void example(void) {
  int i;
  for (i = 0; i < 100; i++) {
    if (i % 2 == 0) {
      continue;
    }
    printf("%d", i);
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdio.h>

// Print the odd numbers between 0 and 99

void example(void) {
  int i;
  for (i = 0; i < 100; i++) {
    if (i % 2 != 0) {
      printf("%d", i);
    }
  }
}
```

MISRAC2004-14.6

Synopsis

Multiple break points from loop.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description
(Required) For any iteration statement, there shall be at most one break statement used for loop termination.

Coding standards
MISRA C:2004 14.6

(Required) For any iteration statement, there shall be at most one break statement used for loop termination.

Code examples
The following code example fails the check and will give a warning:

```
void func()
{
  int x = 1;
  for ( int i = 0; i < 10; i++ )
  {
    if ( x )
    {
      break;
    }
    else if ( i )
    {
      break;  // Non-compliant - second jump from loop
    }
    else
    {
      // Code
    }
  }
}
int fn(void);

void example(void) {
   int i = fn();
   int j;
   int counter = 0;
   switch (i) {
      case 1:
        break;
      case 2:
      case 3:
        counter++;
        if (i==3) {
              break;
        }
        counter++;
        break;
      case 4:
        for (j = 0; j < 10; j++) {
              if (j == i) {
                     break;
              }
              if (j == counter) {
                     break;
              }
        }
        counter--;
        break;
```

```
        default:
          break;
    }
}
int fn(int i);

void example(void) {
    int counter = 0;
    int i = 0;
    for (i = 0; i < 100; i++) {
        switch (i % 9) {
          case 8:
              counter++;
              break;
          default:
              break;
        }
        if (fn(i)) {
          break;
        }
        if (fn(i)) {
          break;
        }
    }
}

int test1(int);
int test2(int);

void example(void)
{
    int i = 0;
    for (i = 0; i < 10; i++) {
        if (test1(i)) {
          break;
        } else if (test2(i)) {
          break;
        }
    }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void)
{
  int i = 0;
  for (i = 0; i < 10 && i != 9; i++) {
    if (i == 9) {
      break;
    }
  }
}
void func()
{
  int x = 1;
  for ( int i = 0; i < 10; i++ )
  {
    if ( x )
    {
      break;
    }
    else if ( i )
    {
      while ( true )
      {
        if ( x )
        {
          break;
        }
        do
        {
          break;
        }
        while(true);
      }
    }
    else
    {
    }
  }
}
int fn(void);

void example(void) {
  int i = fn();
  int j;
  int counter = 0;
  switch (i) {
    case 1:
      break;
```

```
      case 2:
      case 3:
        counter++;
        if (i==3) {
            break;
        }
        counter++;
        break;
      case 4:
        for (j = 0; j < 10; j++) {
            if (j == i) {
                break;
            }
        }
        counter--;
        break;
      default:
        break;
    }
}
int fn(int i);

void example(void) {
  int counter = 0;
  int i = 0;
  int stop = 0;
  for (i = 0; i < 100 && !stop; i++) {
    switch (i % 9) {
      case 8:
          counter++;
          break;
      default:
          break;
    }
    stop = fn(i);
  }
}
```

MISRAC2004-14.7

Synopsis	A function shall have a single point of exit at the end of the function.
Enabled by default	Yes

Severity/Certainty

Low/Medium

Full description

(Required) A function shall have a single point of exit at the end of the function. function. This is required by IEC 61508, under good programming style.

Coding standards

MISRA C:2004 14.7

(Required) A function shall have a single point of exit at the end of the function.

Code examples

The following code example fails the check and will give a warning:

```
extern int errno;

void example(void) {
  if (errno) {
    return;
  }
  return;
}
```

The following code example passes the check and will not give a warning about this issue:

```
extern int errno;

void example(void) {
  if (errno) {
    goto end;
  }
end:
  {
    return;
  }
}
```

MISRAC2004-14.8_a

Synopsis

Missing braces in do ... while statements

Enabled by default

Yes

Severity/Certainty

Low/Low

Full description

(Required) The statement forming the body of a switch, while, do ... while, or for statement shall be a compound statement.

Coding standards

CERT EXP19-C

> Use braces for the body of an if, for, or while statement

CWE 483

> Incorrect Block Delimitation

MISRA C:2004 14.8

> (Required) The statement forming the body of a switch, while, do ... while, or for statement shall be a compound statement.

Code examples

The following code example fails the check and will give a warning:

```
int example(void) {
  do
    return 0;
  while (1);
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(void) {
  do {
    return 0;
  } while (1);
}
```

MISRAC2004-14.8_b

Synopsis

Missing braces in for statements

Enabled by default

Yes

Severity/Certainty

Low/Low

Full description

(Required) The statement forming the body of a switch, while, do ... while, or for statement shall be a compound statement.

Coding standards

CERT EXP19-C

Use braces for the body of an if, for, or while statement

CWE 483

Incorrect Block Delimitation

MISRA C:2004 14.8

(Required) The statement forming the body of a switch, while, do ... while, or for statement shall be a compound statement.

Code examples

The following code example fails the check and will give a warning:

```
int example(void) {
  for (;;)
    return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(void) {
  for (;;){
    return 0;
  }
}
```

MISRAC2004-14.8_c

Synopsis

Missing braces in switch statements

Enabled by default

Yes

Severity/Certainty

Low/Low

Full description

(Required) The statement forming the body of a switch, while, do ... while, or for statement shall be a compound statement.

Coding standards

CERT EXP19-C

Use braces for the body of an if, for, or while statement

CWE 483

Incorrect Block Delimitation

MISRA C:2004 14.8

(Required) The statement forming the body of a switch, while, do ... while, or for statement shall be a compound statement.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  while(1);
  for(;;);
  do ;
  while (0);
  switch(0);
}
```

The following code example passes the check and will not give a warning about this issue:

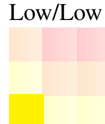
```
void example(void) {
  while(1) {
  }
  for(;;) {
  }
  do {
  } while (0);
  switch(0) {
  }
}
```

MISRAC2004-14.8_d

Synopsis Missing braces in while statements

Enabled by default Yes

Severity/Certainty Low/Low



Full description (Required) The statement forming the body of a switch, while, do ... while, or for statement shall be a compound statement.

Coding standards

CERT EXP19-C

Use braces for the body of an if, for, or while statement

CWE 483

Incorrect Block Delimitation

MISRA C:2004 14.8

(Required) The statement forming the body of a switch, while, do ... while, or for statement shall be a compound statement.

Code examples The following code example fails the check and will give a warning:

```
int example(void) {
  while (1)
    return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(void) {
  while (1){
    return 0;
  }
}
```

MISRAC2004-14.9

Synopsis Missing braces in if, else, and else if statements

Enabled by default

Yes

Severity/Certainty

Low/Low

Full description

(Required) An if expression construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement or another if statement.

Coding standards

CERT EXP19-C

> Use braces for the body of an if, for, or while statement

CWE 483

> Incorrect Block Delimitation

MISRA C:2004 14.9

> (Required) An if expression construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement or another if statement.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  if (random());
  if (random());
  else;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  if (random()) {
  }
  if (random()) {
  } else {
  }
  if (random()) {
  } else if (random()) {
  }
}
```

MISRAC2004-14.10

Synopsis
If ... else if constructs that are not terminated with an else clause.

Enabled by default
Yes

Severity/Certainty
Low/High

Full description
(Required) All if ... else if constructs shall be terminated with an else clause.

Coding standards
MISRA C:2004 14.10

(Required) All if ... else if constructs shall be terminated with an else clause.

Code examples
The following code example fails the check and will give a warning:

```
void example(void) {
  if (!rand()) {
    printf("The first random number is 0");
  } else if (!rand()) {
    printf("The second random number is 0");
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  if (!rand()) {
    printf("The first random number is 0");
  } else if (!rand()) {
    printf("The second random number is 0");
  } else {
    printf("Neither random number was 0");
  }
}
```

MISRAC2004-15.0

Synopsis
Switch statements that do not conform to the MISRA C switch syntax.

Enabled by default
Yes

Severity/Certainty

Low/High

Full description

(Required) The MISRA C switch syntax shall be used. switch-statement : switch '(' expression ')' '{' case-label-clause-list default-label-clause? '}' case-label-clause-list: case-label case-clause? case-label-clause-list case-label case-clause? case-label: case constant-expression ':' case-clause: statement-list? break ';' '{' declaration-list? statement-list? break ';' '}' default-label-clause : default-label default-clause default-label: default ':' default-clause: case-clause

Coding standards

MISRA C:2004 15.0

(Required) The MISRA C switch syntax shall be used.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  switch(expr()) {
    // at least one case label
    case 1:
      // statement list
      stmt();
      stmt();
      // WARNING: missing break at end of statement list
    default:
      break; // statement list ends in a break
  }

  switch(expr()) {
    // WARNING: missing at least one case label
    default:
      break; // statement list ends in a break
  }

  switch(expr()) {
    // at least one case label
    case 1:
      // statement list
      stmt();
      stmt();
      break; // statement list ends in a break
    case 0:
      stmt();
      // WARNING: declaration list without block
      int decl = 0;
      int x;
      // statement list
      stmt();
      stmt();
      break; // statement list ends in a break
    default:
      break; // statement list ends in a break
  }

  switch(expr()) {
    // at least one case label
    case 1: {
      // statement list
      stmt();
      // WARNING: Additional block inside of the case clause
block
      {
      stmt();
```

```
      }
      break;
    }
    default:
      break; // statement list ends in a break
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  switch(expr()) {
    // at least one case label
    case 1:
      // statement list (no declarations)
      stmt();
      stmt();
      break; // statement list ends in a break
    case 0: {
      // one level of block is allowed
      // declaration list
      int decl = 0;
      // statement list
      stmt();
      stmt();
      break; // statement list ends in a break
    }
    case 2: // empty cases are allowed
    default:
      break; // statement list ends in a break
  }
}
```

MISRAC2004-15.1

Synopsis

Switch labels in nested blocks.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.

Coding standards

MISRA C:2004 15.1

> (Required) A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {

  switch(rand()) {
    {case 1:}
    case 2:
    case 3:
    default:
  }

}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {

  switch(rand()) {
    case 1:
    case 2:
    case 3:
    default:
  }

}
```

MISRAC2004-15.2

Synopsis

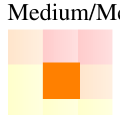
Non-empty switch cases not terminated by break

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) An unconditional break statement shall terminate every non-empty switch clause.

Coding standards

CERT MSC17-C

Finish every set of statements associated with a case label with a break statement

CWE 484

Omitted Break Statement in Switch

MISRA C:2004 15.2

(Required) An unconditional break statement shall terminate every non-empty switch clause.

Code examples

The following code example fails the check and will give a warning:

```
void example(int input) {

  while (rand()) {
    switch(input) {
      case 0:
        if (rand()) {
          break;
        }
      default:
        break;
    }
  }
}
void example(int input) {

  switch(input) {
    case 0:
      if (rand()) {
        break;
      }
    default:
      break;
  }

}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(int input) {

  switch(input) {
    case 0:
      if (rand()) {
        break;
      }
      break;
    default:
      break;
  }

}
void example(int input) {

  switch(input) {
    case 0:
      if (rand()) {
        break;
      } else {
        break;
      }
      // All paths above contain a break, therefore we do not
warn
    default:
      break;
  }

}
```

MISRAC2004-15.3

Synopsis

Switch statements with no default clause, or a default clause that is not the final clause.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The final clause of a switch statement shall be the default clause.

Coding standards

CWE 478

Missing Default Case in Switch Statement

MISRA C:2004 15.3

(Required) The final clause of a switch statement shall be the default clause.

Code examples

The following code example fails the check and will give a warning:

```
int example(int x) {
  switch(x){
    default:
      return 2;
      break;
    case 0:
      return 0;
      break;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(int x) {
  switch(x){
    case 3:
      return 0;
      break;
    case 5:
      return 1;
      break;
    default:
      return 2;
      break;
  }
}
```

MISRAC2004-15.4

Synopsis

A switch expression shall not represent a value that is effectively boolean.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) A switch expression shall not represent a value that is effectively boolean.

Coding standards

MISRA C:2004 15.4

> (Required) A switch expression shall not represent a value that is effectively boolean.

Code examples

The following code example fails the check and will give a warning:

```
void example(int x) {
  switch(x == 0) {
    case 0:
    case 1:
    default:
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(int x) {
  switch(x) {
    case 1:
    case 0:
    default:
  }
}
```

MISRAC2004-15.5

Synopsis

Switch statements with no cases.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Every switch statement shall have at least one case clause.

Coding standards

MISRA C:2004 15.5

> (Required) Every switch statement shall have at least one case clause.

Code examples

The following code example fails the check and will give a warning:

```
int example(int x) {
  switch(x){
    default:
      return 2;
      break;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(int x) {
  switch(x){
    case 3:
      return 0;
      break;
    case 5:
      return 1;
      break;
    default:
      return 2;
      break;
  }
}
```

MISRAC2004-16.1

Synopsis

Functions defined using ellipsis (...) notation

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

(Required) Functions shall not be defined with a variable number of arguments. Additionally, passing an argument with non-POD class type leads to undefined behaviour. Note that the rule specifies defined (and not declared) so as to permit the use of existing library functions.

Coding standards

MISRA C:2004 16.1

(Required) Functions shall not be defined with a variable number of arguments.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdarg.h>
int putchar(int c);

void
minprintf(const char *fmt, ...)
{
    va_list    ap;
    const char *p, *s;

    va_start(ap, fmt);
    for (p = fmt; *p != '\0'; p++) {
        if (*p != '%') {
            putchar(*p);
            continue;
        }
        switch (*++p) {
        case 's':
            for (s = va_arg(ap, const char *); *s != '\0'; s++)
                putchar(*s);
            break;
        }
    }
    va_end(ap);
}
```

The following code example passes the check and will not give a warning about this issue:

```
int puts(const char *);

void
func(void)
{
    puts("Hello, world!");
}
```

MISRAC2004-16.2_a

Synopsis

Functions that call themselves directly.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Functions shall not call themselves, either directly or indirectly.

Coding standards

MISRA C:2004 16.2

(Required) Functions shall not call themselves, either directly or indirectly.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  example();
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC2004-16.2_b

Synopsis

Functions that call themselves indirectly.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Functions shall not call themselves, either directly or indirectly.

Coding standards

MISRA C:2004 16.2

(Required) Functions shall not call themselves, either directly or indirectly.

Code examples

The following code example fails the check and will give a warning:

```
void example(void);
void callee(void) {
    example();
}
void example(void) {
    callee();
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void);
void callee(void) {
    // example();
}
void example(void) {
    callee();
}
```

MISRAC2004-16.3

Synopsis

Function prototypes must name all parameters

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

(Required) Identifiers shall be given for all of the parameters in a function prototype declaration.

Coding standards

MISRA C:2004 16.3

> (Required) Identifiers shall be given for all of the parameters in a function prototype declaration.

Code examples

The following code example fails the check and will give a warning:

```
char *strchr(const char *, int c);

void func(void)
{
    strchr("hello, world!\n", '!');
}
```

The following code example passes the check and will not give a warning about this issue:

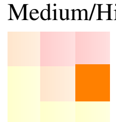
```
char *strchr(const char *s, int c);

void func(void)
{
    strchr("hello, world!\n", '!');
}
```

MISRAC2004-16.5

Synopsis
Functions declared with an empty () parameter list that does not form a valid prototype

Enabled by default
Yes

Severity/Certainty
Medium/High



Full description
(Required) Functions with no parameters shall be declared and defined with the parameter list void.

Coding standards
CERT DCL20-C

Always specify void even if a function accepts no arguments

MISRA C:2004 16.5

(Required) Functions with no parameters shall be declared and defined with the parameter list void.

Code examples
The following code example fails the check and will give a warning:

```
void func();/* not a valid prototype in C */
void func2(void)
{
    func();
}
```

The following code example passes the check and will not give a warning about this issue:

```
void func(void);
void func2(void)
{
    func();
}
```

MISRAC2004-16.7

Synopsis

A function does not modify one of its parameters.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.

Coding standards

MISRA C:2004 16.7

> (Required) A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.

Code examples

The following code example fails the check and will give a warning:

```
int example(int* x) {  //x should be const
  if (*x > 5){
    return *x;
  } else {
    return 5;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(const int* x) {  //OK
  if (*x > 5){
    return *x;
  } else {
    return 5;
  }
}
```

MISRAC2004-16.8

Synopsis

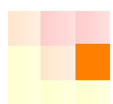
For some execution, no return statement is executed in a function with a non-void return type

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

(Required) All exit paths from a function with non-void return type shall have an explicit return statement with an expression. Checks whether all execution paths in non-void functions contain a return statement before they exit. If a non-void function has no return statement, it will return an undefined value. This will not pose a problem if the function is used as a void function, however, if the function return value is used it will cause unpredictable behavior.   Note: This is a weaker check than the one performed by gcc. Its check allows more aggressive coding without violating the rule. However, a rule violation in gcc means there is no path leading to a return statement. non-`void` return type.

Coding standards

CERT MSC37-C

> Ensure that control never reaches the end of a non-void function

MISRA C:2004 16.8

> (Required) All exit paths from a function with non-void return type shall have an explicit return statement with an expression.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdio.h>

int example(void) {
  int x;

  scanf("%d",&x);

  if (x > 10) {
    return 10;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdio.h>

int example(void) {
  int x;

  scanf("%d",&x);

  if (x > 10) {
    return 10;
  }

  return 0;
}
```

MISRAC2004-16.9

Synopsis

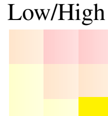
Function addresses taken without explicit &

Enabled by default

Yes

Severity/Certainty

Low/High



Full description

(Required) A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty.

Coding standards

MISRA C:2004 16.9

(Required) A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty.

Code examples

The following code example fails the check and will give a warning:

```
void func(void);

void
example(void)
{
    void (*pf)(void) = func;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void func(void);

void
example(void)
{
    void (*pf)(void) = &func;
}
```

MISRAC2004-16.10

Synopsis

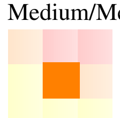
The return value for a library function that may return an error value is not used.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) If a function returns error information, then that error information shall be tested.

Coding standards

CWE 252

Unchecked Return Value

CWE 394

Unexpected Status Code or Return Value

MISRA C:2004 16.10

(Required) If a function returns error information, then that error information shall be tested.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  malloc(sizeof(int));  // This function could fail,
                        // and the return value is
                        // not checked
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

void example(void) {
  int *x = malloc(sizeof(int));  // OK - return value
                                 // is stored
}
```

MISRAC2004-17.1_a

Synopsis

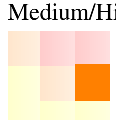
Direct access to a field of a struct using an offset from the address of the struct.

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

(Required) Pointer arithmetic shall only be applied to pointers that address an array or array element.

Coding standards

CERT ARR37-C

Do not add or subtract an integer to a pointer to a non-array object

CWE 188

Reliance on Data/Memory Layout

MISRA C:2004 17.1

(Required) Pointer arithmetic shall only be applied to pointers that address an array or array element.

Code examples

The following code example fails the check and will give a warning:

```
struct S{
  char c;
  int x;
};

void main(void) {
  struct S s;
  *(&s.c+1) = 10;
}
```

The following code example passes the check and will not give a warning about this issue:

```
struct S{
  char c;
  int x;
};

void example(void) {
  struct S s;
  s.x = 10;
}
```

MISRAC2004-17.1_b

Synopsis

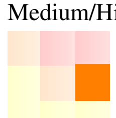
Pointer arithmetic applied to a pointer that references a stack address

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

(Required) Pointer arithmetic shall only be applied to pointers that address an array or array element.

Coding standards

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

MISRA C:2004 17.1

(Required) Pointer arithmetic shall only be applied to pointers that address an array or array element.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
    int i;
    int *p = &i;
    p++;
    *p = 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
    int i;
    int *p = &i;
    *p = 0;
}
```

MISRAC2004-17.1_c

Synopsis

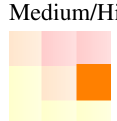
Invalid pointer arithmetic with an automatic variable that is neither an array nor a pointer.

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

(Required) Pointer arithmetic shall only be applied to pointers that address an array or array element. This check warns when the address of an automatic variable is taken, and arithmetic is performed on it, as this behavior indicates that an invalid memory access attempt may occur. It handles local variables, parameters and globals, including structs.

Coding standards

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

MISRA C:2004 17.1

(Required) Pointer arithmetic shall only be applied to pointers that address an array or array element.

Code examples

The following code example fails the check and will give a warning:

```
void example(int x) {
  *(&x+10) = 5;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(int *x) {
  *(x+10) = 5;
}
```

MISRAC2004-17.4_a

Synopsis

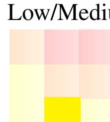
Array indexing shall be the only allowed form of pointer arithmetic.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) Array indexing shall be the only allowed form of pointer arithmetic.

Coding standards

MISRA C:2004 17.4

(Required) Array indexing shall be the only allowed form of pointer arithmetic.

Code examples

The following code example fails the check and will give a warning:

```
typedef int INT32;

void example(INT32 array[]) {
   INT32 *pointer = array;
   INT32 *end = array + 10;
   for (; pointer != end; pointer += 1) {
      *pointer = 0;
   }
}
```

The following code example passes the check and will not give a warning about this issue:

```
typedef int INT32;

void example(INT32 array[]) {
  INT32 index = 0;
  INT32 end = 10;
  for (; index != end; index += 1) {
    array[index] = 0;
  }
}
```

MISRAC2004-17.4_b

Synopsis

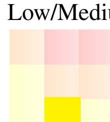
Array indexing shall only be applied to objects defined as an array type.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) Array indexing shall be the only allowed form of pointer arithmetic.

Coding standards

MISRA C:2004 17.4

(Required) Array indexing shall be the only allowed form of pointer arithmetic.

Code examples

The following code example fails the check and will give a warning:

```
typedef unsigned charUINT8;
typedefunsigned intUINT;

void example(UINT8 *p, UINT size) {
  UINT i;
  for (i = 0; i < size; i++) {
    p[i] = 0;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
typedef unsigned charUINT8;
typedef unsigned intUINT;

void example(void) {
  UINT8 p[10];
  UINT  i;
  for (i = 0; i < 10; i++) {
    p[i] = 0;
  }
}
```

MISRAC2004-17.5

Synopsis

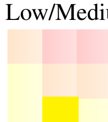
The declaration of objects should contain no more than two levels of pointer indirection.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) The declaration of objects should contain no more than two levels of pointer indirection.

Coding standards

MISRA C:2004 17.5

> (Required) The declaration of objects should contain no more than two levels of pointer indirection.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
    int ***p;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
    int **p;
}
```

MISRAC2004-17.6_a

Synopsis

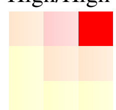
May return address on the stack.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist. Depending on the circumstances, this code and subsequent memory accesses could appear to work, but the operations are illegal and a program crash, or memory corruption, is very likely. Returning a copy of the object, using a global variable, or dynamically allocating memory, are possible alternatives.

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 562

Return of Stack Variable Address

MISRA C:2004 17.6

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Code examples

The following code example fails the check and will give a warning:

```
int *f() {
  int x;
  return &x;  //x is a local variable
}
int *example(void) {
  int a[20];
  return a;  //a is a local array
}
```

The following code example passes the check and will not give a warning about this issue:

```
int* example(void) {
  int *p,i;
  p = (int *)malloc(sizeof(int));
  return p;  //OK - p is dynamically allocated

}
```

MISRAC2004-17.6_b

Synopsis

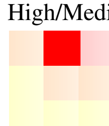
Store a stack address in a global pointer.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist. This is particularly dangerous because the program may appear to work normally, while it is in fact accessing illegal memory. Other results of this are a program crash, or the data changing unpredictably.

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 466

Return of Pointer Value Outside of Expected Range

MISRA C:2004 17.6

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Code examples

The following code example fails the check and will give a warning:

```
int *px;
void example() {
  int i = 0;
  px = &i; // assigning the address of stack
           // variable a to the global px
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(int *pz) {
  int x; int *px = &x;
  int *py = px; /* local variable */
  pz = px; /* parameter */
}
```

MISRAC2004-17.6_c

Synopsis

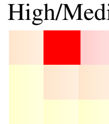
Store a stack address in the field of a global struct.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist. This is particularly dangerous because the program may appear to work normally, while it is in fact accessing illegal memory. Other results of this are a program crash, or the data changing unpredictably.

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 466

Return of Pointer Value Outside of Expected Range

MISRA C:2004 17.6

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Code examples

The following code example fails the check and will give a warning:

```
struct S{
  int *px;
} s;

void example() {
  int i = 0;
  s.px = &i; //storing local address in global struct
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

struct S{
  int *px;
} s;

void example() {
  int i = 0;
  s.px = &i; //OK - the field is written to later
  s.px = NULL;
}
```

MISRAC2004-17.6_d

Synopsis

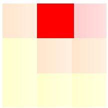
Store stack address outside function via parameter.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist. This is particularly dangerous because the program may appear to work normally, while it is in fact accessing illegal memory. Other results of this are a program crash, or the data changing unpredictably. Known false positives: this test checks for any expression refering to the store located by the parameter and so the assignment 'local[*parameter] = & local;' will invoke a warning.

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 466

Return of Pointer Value Outside of Expected Range

MISRA C:2004 17.6

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Code examples

The following code example fails the check and will give a warning:

```
void example(int **ppx) {
  int x;
  ppx[0] = &x;  //local address
}
```

The following code example passes the check and will not give a warning about this issue:

```
static int y = 0;
void example3(int **ppx){
  *ppx = &y;  //OK - static address
}
```

MISRAC2004-18.1

Synopsis

Structs and unions that are used without being defined.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) All structure and union types shall be complete at the end of the translation unit.

Coding standards

MISRA C:2004 18.1

(Required) All structure and union types shall be complete at the end of the translation unit.

Code examples

The following code example fails the check and will give a warning:

```
struct incomplete;

void example(struct incomplete *p)
{
}
```

The following code example passes the check and will not give a warning about this issue:

```
struct complete {
    int x;
};

void example(struct complete *p)
{
}
```

MISRAC2004-18.2

Synopsis

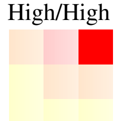
Assignments from one field of a union to another.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

(Required) An object shall not be assigned to an overlapping object.

Coding standards

MISRA C:2004 18.2

(Required) An object shall not be assigned to an overlapping object.

Code examples

The following code example fails the check and will give a warning:

```
union cheat {
    char c[5];
    int i;
};

void example(union cheat *u)
{
    u->i = u->c[2];
}
union {
    char c[5];
    int i;
} u;

void example(void)
{
    u.i = u.c[2];
}
void example(void)
{
  union
  {
    char c[5];
    int i;
  } u;
  u.i = u.c[2];
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void)
{
  union
  {
    char c[5];
    int i;
  } u;
  int x;
  x = (int)u.c[2];
  u.i = x;
}
void example(void)
{
  struct
  {
    char c[5];
    int i;
  } u;
  u.i = u.c[2];
}
union cheat {
  char c[5];
  int i;
};

union cheat u;

void example(void)
{
  int x;
  x = (int)u.c[2];
  u.i = x;
}
```

MISRAC2004-18.4

Synopsis	All unions
Enabled by default	Yes
Severity/Certainty	Low/Medium

Full description

(Required) Unions shall not be used.

Coding standards

MISRA C:2004 18.4

(Required) Unions shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
union cheat {
  int   i;
  float f;
};

int example(float f) {
  union cheat u;
  u.f = f;
  return u.i;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(int x) {
  return x;
}
```

MISRAC2004-19.2

Synopsis

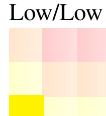
Illegal characters in header file names

Enabled by default

No

Severity/Certainty

Low/Low



Full description

(Advisory) Non-standard characters should not occur in header file names in #include directives. ', \, /*, or // characters are used between the " delimiters in a header name preprocessing token.

Coding standards

MISRA C:2004 19.2

(Advisory) Non-standard characters should not occur in header file names in #include directives.

Code examples

The following code example fails the check and will give a warning:

```
#include "fi'le.h"/* Non-compliant */
void example(void) {}
```

The following code example passes the check and will not give a warning about this issue:

```
#include "header.h"
void example(void) {}
```

MISRAC2004-19.6

Synopsis

All #undef's

Enabled by default

Yes

Severity/Certainty

Low/Low

Full description

(Required) #undef shall not be used. or meaning of a macro when it is used in the code.

Coding standards

MISRA C:2004 19.6

(Required) #undef shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
#defineSYM
#undef SYM
void example(void) {}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {}
```

MISRAC2004-19.7

Synopsis

Function-like macros

Enabled by default

No

Severity/Certainty

Low/Low

Full description

(Advisory) A function should be used in preference to a function-like macro. robust mechanism. This is particularly true with respect to the type checking of parameters, and the problem of function-like macros potentially evaluating parameters multiple times. Inline functions should be used instead.

Coding standards

MISRA C:2004 19.7

(Advisory) A function should be used in preference to a function-like macro.

Code examples

The following code example fails the check and will give a warning:

```
#defineABS(x)((x) < 0 ? -(x) : (x))

void example(void) {
  int a;
  ABS (a);
}
```

The following code example passes the check and will not give a warning about this issue:

```
template <typename T>
inline T ABS(T x) { return x < 0 ? -x : x; }
```

MISRAC2004-19.12

Synopsis

Multiple # or ## operators in a macro definition

Enabled by default

Yes

Severity/Certainty

Medium/Low

Full description

(Required) There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition. This problem can be avoided by having only one occurrence of either operator in any single macro definition (i.e. one #, or one ## or neither).

Coding standards MISRA C:2004 19.12

(Required) There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.

Code examples The following code example fails the check and will give a warning:

```
#defineD(x, y, z, yz)x ## y ## z/* Non-compliant */
#define C(x, y)# x ## y/* Non-compliant */
```

The following code example passes the check and will not give a warning about this issue:

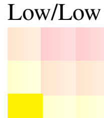
```
#define A(x)#x/* Compliant */
#defineB(x, y)x ## y/* Compliant */
```

MISRAC2004-19.13

Synopsis The # and ## operators should not be used

Enabled by default No

Severity/Certainty Low/Low



Full description (Advisory) The # and ## preprocessor operators should not be used. Compilers have been known to implement these operators inconsistently, therefore, to avoid these problems, do not use them.

Coding standards MISRA C:2004 19.13

(Advisory) The # and ## preprocessor operators should not be used.

Code examples The following code example fails the check and will give a warning:

```
#defineA(X,Y)X##Y/* Non-compliant */

#define A(Y)#Y/* Non-compliant */
```

The following code example passes the check and will not give a warning about this issue:

```
#define A(x)(x)/* Compliant */
```

MISRAC2004-19.15

Synopsis

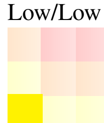
Header files without #include guards

Enabled by default

Yes

Severity/Certainty

Low/Low



Full description

(Required) Precautions shall be taken in order to prevent the contents of a header file being included twice. particular header file to be included more than once. This can be, at best, a source of confusion. If this multiple inclusion leads to multiple or conflicting definitions, then this can result in undefined or erroneous behaviour.

Coding standards

MISRA C:2004 19.15

> (Required) Precautions shall be taken in order to prevent the contents of a header file being included twice.

Code examples

The following code example fails the check and will give a warning:

```
#include "unguarded_header.h"
void example(void) {}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>
#include "header.h"/* contains #ifndef HDR #define HDR ... #endif
*/
void example(void) {}
```

MISRAC2004-20.1

Synopsis

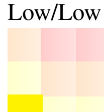
#define or #undef of a reserved identifier in the standard library

Enabled by default

Yes

Severity/Certainty

Low/Low



Full description

(Required) Reserved identifiers, macros, and functions in the standard library shall not be defined, redefined, or undefined. practice to #define a macro name that is a C/C++ reserved identifier, or C/C++ keyword or the name of any macro, object or function in the standard library. For example, there are some specific reserved words and function names that are known to give rise to undefined behaviour if they are redefined or undefined, including defined, __LINE__, __FILE__, __DATE__, __TIME__, __STDC__, errno and assert.

Coding standards

MISRA C:2004 20.1

> (Required) Reserved identifiers, macros, and functions in the standard library shall not be defined, redefined, or undefined.

Code examples

The following code example fails the check and will give a warning:

```
#define__TIME__11111111/* Non-compliant */
```

The following code example passes the check and will not give a warning about this issue:

```
#define A(x)(x)/* Compliant */
```

MISRAC2004-20.4

Synopsis

All uses of malloc, calloc, realloc, and free

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Dynamic heap memory allocation shall not be used.

Coding standards

MISRA C:2004 20.4

> (Required) Dynamic heap memory allocation shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

void *example(void) {
  return malloc(100);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC2004-20.5

Synopsis

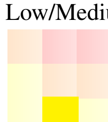
All uses of errno

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) The error indicator errno shall not be used.

Coding standards

MISRA C:2004 20.5

(Required) The error indicator errno shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
#include <errno.h>
#include <stdlib.h>
//int errno;

int example(char buf[]) {
  int i;
  errno = 0;
  i = atoi(buf);
  return (errno == 0) ? i : 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC2004-20.6

Synopsis

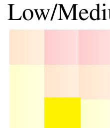
All uses of the offsetof built-in function

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) The macro offsetof in the stddef.h library shall not be used.

Coding standards

MISRA C:2004 20.6

(Required) The macro offsetof in the stddef.h library shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
#include <stddef.h>
//#include <sys/stat.h>
struct stat { int st_size; };

int example(void) {
  return offsetof(struct stat, st_size);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC2004-20.7

Synopsis

All uses of <setjmp.h>

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The setjmp macro and the longjmp function shall not be used.

Coding standards

CERT ERR34-CPP

Do not use longjmp

MISRA C:2004 20.7

(Required) The setjmp macro and the longjmp function shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
#include <setjmp.h>

jmp_buf ex;

void example(void) {
  setjmp(ex);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC2004-20.8

Synopsis

All uses of <signal.h>

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The signal handling facilities of signal.h shall not be used.

Coding standards

MISRA C:2004 20.8

(Required) The signal handling facilities of signal.h shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
#include <signal.h>
#include <stddef.h>

void example(void) {
  signal(SIGFPE, NULL);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC2004-20.9

Synopsis

All uses of <stdio.h>

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The input/output library stdio.h shall not be used in production code.

Coding standards

MISRA C:2004 20.9

(Required) The input/output library stdio.h shall not be used in production code.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdio.h>

void example(void) {
  printf("Hello, world!\n");
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC2004-20.10

Synopsis

All uses of atof, atoi, atol and atoll

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The functions atof, atoi, and atol from the library stdlib.h shall not be used.

Coding standards

CERT INT06-C

Use strtol() or a related function to convert a string token to an integer

MISRA C:2004 20.10

(Required) The functions atof, atoi, and atol from the library stdlib.h shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

int example(char buf[]) {
  return atoi(buf);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC2004-20.11

Synopsis

All uses of abort, exit, getenv, and system

Enabled by default

Yes

Severity/Certainty | Low/Medium

Full description | (Required) The functions abort, exit, getenv, and system from the library stdlib.h shall not be used.

Coding standards | MISRA C:2004 20.11

> (Required) The functions abort, exit, getenv, and system from the library stdlib.h shall not be used.

Code examples | The following code example fails the check and will give a warning:

```
#include <stdlib.h>

void example(void) {
  abort();
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC2004-20.12

Synopsis | All uses of <time.h> functions: asctime, clock, ctime, difftime, gmtime, localtime, mktime, strftime, and time

Enabled by default | Yes

Severity/Certainty | Low/Medium

Full description | (Required) The time handling functions of time.h shall not be used.

Coding standards | MISRA C:2004 20.12

> (Required) The time handling functions of time.h shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
#include <stddef.h>
#include <time.h>

time_t example(void) {
  return time(NULL);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC2012-Dir-4.3

Synopsis

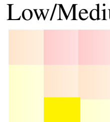
Inline asm statements that are not encapsulated in functions

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) Assembly language shall be encapsulated and isolated

Coding standards

MISRA C:2012 Dir-4.3

(Required) Assembly language shall be encapsulated and isolated

Code examples

The following code example fails the check and will give a warning:

```
int ffs(int x)
{
        int r;
#if 0
#ifdef CONFIG_X86_64
        /*
         * AMD64 says BSFL won't clobber the dest reg if x==0;
Intel64 says the
         * dest reg is undefined if x==0, but their CPU architect
says its
         * value is written to set it to the same as before,
except that the
         * top 32 bits will be cleared.
         *
         * We cannot do this on 32 bits because at the very least
some
         * CPUs did not behave this way.
         */
        long tmp = -1;
        asm("bsfl %1,%0"
            : "=r" (r)
            : "rm" (x), "" (tmp));
#elif defined(CONFIG_X86_CMOV)
        asm("bsfl %1,%0\n\t"
            "cmovzl %2,%0"
            : "=&r" (r) : "rm" (x), "r" (-1));
#else
        asm("bsfl %1,%0\n\t"
            "jnz 1f\n\t"
            "movl $-1,%0\n"
            "1:" : "=r" (r) : "rm" (x));
#endif
#else
        asm("");
#endif
        return r + 1;
}
```

The following code example passes the check and will not give a warning about this issue:

```
unsigned int
bswap(unsigned int x)
{
    asm("");
    return x;
}
```

MISRAC2012-Dir-4.4

Synopsis

To allow comments to contain pseudo-code or code samples, only comments that end in ';', '{', or '}' characters are considered to be commented-out code.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

(Advisory) Sections of code should not be "commented out" Code sections in comments are identified where the comment ends in ';', '{', or '}' characters.

Coding standards

MISRA C:2012 Dir-4.4

(Advisory) Sections of code should not be "commented out"

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  /*
  int i;
  */
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
#if 0
  int i;
#endif
}
```

MISRAC2012-Dir-4.6_a

Synopsis

Uses of basic types char, int, short, long, double, and float without typedef

Enabled by default

No

Severity/Certainty

Low/High

Full description

(Advisory) typedefs that indicate size and signedness should be used in place of the basic numerical types Best practice is to use typedefs for portability.

Coding standards

MISRA C:2012 Dir-4.6

(Advisory) typedefs that indicate size and signedness should be used in place of the basic numerical types

Code examples

The following code example fails the check and will give a warning:

```
typedef signed charSCHAR;
typedef intINT;
typedef floatFLOAT;

INT func(FLOAT f, INT *pi)
{
    INT x;
    INT (*fp)(const char *);
}
```

The following code example passes the check and will not give a warning about this issue:

```
typedef signed charSCHAR;
typedef intINT;
typedef floatFLOAT;

INT func(FLOAT f, INT *pi)
{
    INT x;
    INT (*fp)(const SCHAR *);
}
```

MISRAC2012-Dir-4.9

Synopsis

Function-like macros

Enabled by default

No

Severity/Certainty

Low/Low

Full description

(Advisory) A function should be used in preference to a function-like macro where they are interchangeable robust mechanism. This is particularly true with respect to the type checking of parameters, and the problem of function-like macros potentially evaluating parameters multiple times. Inline functions should be used instead.

Coding standards

MISRA C:2012 Dir-4.9

> (Advisory) A function should be used in preference to a function-like macro where they are interchangeable

Code examples

The following code example fails the check and will give a warning:

```
#defineABS(x)((x) < 0 ? -(x) : (x))

void example(void) {
  int a;
  ABS (a);
}
```

The following code example passes the check and will not give a warning about this issue:

```
template <typename T>
inline T ABS(T x) { return x < 0 ? -x : x; }
```

MISRAC2012-Dir-4.10

Synopsis

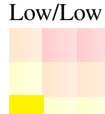
Header files without #include guards

Enabled by default

Yes

Severity/Certainty

Low/Low



Full description

(Required) Precautions shall be taken in order to prevent the contents of a header file being included more than once particular header file to be included more than once. This can be, at best, a source of confusion. If this multiple inclusion leads to multiple or conflicting definitions, then this can result in undefined or erroneous behaviour.

Coding standards

MISRA C:2012 Dir-4.10

(Required) Precautions shall be taken in order to prevent the contents of a header file being included more than once

Code examples

The following code example fails the check and will give a warning:

```
#include "unguarded_header.h"
void example(void) {}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>
#include "header.h"/* contains #ifndef HDR #define HDR ... #endif
*/
void example(void) {}
```

MISRAC2012-Rule-1.3_a

Synopsis

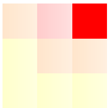
An expression resulting in `0` is used as a divisor.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

(Required) There shall be no occurrence of undefined or critical unspecified behaviour by interval analysis to be `0`, and it is used as a divisor. If this code executes, a `divide by zero' runtime error will occur.

Coding standards

CERT INT33-C

Ensure that division and modulo operations do not result in divide-by-zero errors

CWE 369

Divide By Zero

MISRA C:2012 Rule-1.3

(Required) There shall be no occurrence of undefined or critical unspecified behaviour

Code examples

The following code example fails the check and will give a warning:

```
int foo(void)
{
  int a = 3;
  a--;
  return 5 / (a-2);  // a-2 is 0
}
#include <stdlib.h>

int main (void)

{
 int *p = malloc( sizeof(int));
  int x = foo (p);
  /* foo(2) returns 8, so we have a division by zero below)*/
  x = 1 / (x - 8);              /*@@ZDV-RED@@ */

  return x;
}


int foo(int * p){
  return 8;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int foo(void)
{
  int a = 3;
  a--;
  return 5 / (a+2);  // OK - a+2 is 4
}
```

MISRAC2012-Rule-1.3_b

Synopsis

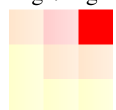
A variable is assigned the value `0`, then used as a divisor.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

(Required) There shall be no occurrence of undefined or critical unspecified behaviour If this code executes, a `divide by zero' runtime error will occur.

Coding standards

CERT INT33-C

> Ensure that division and modulo operations do not result in divide-by-zero errors

CWE 369

> Divide By Zero

MISRA C:2012 Rule-1.3

> (Required) There shall be no occurrence of undefined or critical unspecified behaviour

Code examples

The following code example fails the check and will give a warning:

```
int foo(void)
{
  int a = 20, b = 0, c;

  c = a / b;     /* Divide by zero */

  return c;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int foo(void)
{
  int a = 20, b = 5, c;

  c = a / b; /* b is not 0 */

  return c;
}
int main() {
    int totallen = 0;
    int i=0;
    float tmp=1;

   for( i=1; i<10; i++){
   totallen++;
   }

   foo(2/totallen);

   return 0;
}

int foo(int x){
   return x;
}
```

MISRAC2012-Rule-1.3_c

Synopsis After a successful comparison with `0`, a variable is used as a divisor.

Enabled by default Yes

Severity/Certainty Medium/High

Full description (Required) There shall be no occurrence of undefined or critical unspecified behaviour then used as a divisor without being written to beforehand. The presence of this comparison implies that the variable's value is `0` for the following statements. As such, its being used as a divisor afterwards would invoke a `divide by zero' runtime error.

Coding standards

CERT INT33-C

Ensure that division and modulo operations do not result in divide-by-zero errors

CWE 369

Divide By Zero

MISRA C:2012 Rule-1.3

(Required) There shall be no occurrence of undefined or critical unspecified behaviour

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>
int foo(void)
{
  int a = 20;
  int p = rand();

  if (p == 0)   /* p is 0 */
    a = 34 / p;

  return a;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>
int foo(void)
{
  int a = 20;
  int p = rand();

  if (p != 0)   /* p is not 0 */
    a = 34 / p;

  return a;
}
```

MISRAC2012-Rule-1.3_d

Synopsis

A variable used as a divisor is subsequently compared with `0`.

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

(Required) There shall be no occurrence of undefined or critical unspecified behavior This check will produce a warning if a variable is compared to `0` after it is used as a divisor, but before it is written to again. The comparison implies that the variable's value may be `0`, and thus may have been for the preceding statements. As one of these statements is an operation using the variable as a divisor (which would invoke a `divide by zero' runtime error), the program's execution can never reach the comparison when the value is `0`, rendering it redundant.

Coding standards

CERT INT33-C

> Ensure that division and modulo operations do not result in divide-by-zero errors

CWE 369

> Divide By Zero

MISRA C:2012 Rule-1.3

> (Required) There shall be no occurrence of undefined or critical unspecified behavior

Code examples

The following code example fails the check and will give a warning:

```
int foo(int p)
{
  int a = 20, b = 1;
  b = a / p;
  if (p == 0) // Checking the value of 'p' too late.
    return 0;
  return b;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int foo(int p)
{
  int a = 20, b;
  if (p == 0)
    return 0;
  b = a / p;    /* Here 'p' is non-zero. */
  return b;
}
```

MISRAC2012-Rule-1.3_e

Synopsis

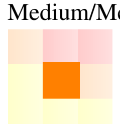
Interval analysis determines a value is `0`, then it is used as a divisor.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) There shall be no occurrence of undefined or critical unspecified behaviour by interval analysis to be `0`, and it is used as a divisor. The warning addresses the possibility that the division may invoke a `divide by zero' runtime error.

Coding standards

CERT INT33-C

Ensure that division and modulo operations do not result in divide-by-zero errors

CWE 369

Divide By Zero

MISRA C:2012 Rule-1.3

(Required) There shall be no occurrence of undefined or critical unspecified behaviour

Code examples

The following code example fails the check and will give a warning:

```
int foo(void)
{
  int a = 1;
  a--;
  return 5 / a;  /* a is 0 */
}
```

The following code example passes the check and will not give a warning about this issue:

```
int foo(void)
{
  int a = 2;
  a--;
  return 5 / a;   /* OK - a is 1 */
}
```

MISRAC2012-Rule-1.3_f

Synopsis

An expression that may be `0` is used as a divisor.

Enabled by default

Yes

Severity/Certainty

High/Low

Full description

(Required) There shall be no occurrence of undefined or critical unspecified behaviour divisor, and its value, as determined by interval analysis contains `0`. If this code executes, a `divide by zero' runtime error may occur.

Coding standards

CERT INT33-C

Ensure that division and modulo operations do not result in divide-by-zero errors

CWE 369

Divide By Zero

MISRA C:2012 Rule-1.3

(Required) There shall be no occurrence of undefined or critical unspecified behaviour

Code examples

The following code example fails the check and will give a warning:

```
int main (void)

{

  int x = 2;

  int i;



  /* The second iteration leads to a division by zero*/

  for (i = 1; i < 3; i++) { x = x / (2 - i); }
/*@@ZDV-RED@@ */



  return x;

}

int foo(void)
{
  int a = 3;
  a--;
  return 5 / (a-2);  // a-2 is 0
}
```

The following code example passes the check and will not give a warning about this issue:

```
int foo(void)
{
  int a = 3;
  a--;
  return 5 / (a+2);  // OK - a+2 is 4
}
```

MISRAC2012-Rule-1.3_g

Synopsis — A global variable is not checked against `0` before it is used as a divisor.

Enabled by default — Yes

Severity/Certainty

Medium/Low

Full description

(Required) There shall be no occurrence of undefined or critical unspecified behaviour If the variable has a value of 0, then a `divide by zero' runtime error will occur.

Coding standards

CWE 369

Divide By Zero

MISRA C:2012 Rule-1.3

(Required) There shall be no occurrence of undefined or critical unspecified behaviour

Code examples

The following code example fails the check and will give a warning:

```
int x;

int example() {
  return 5/x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int x;

int example() {
  if (x != 0){
    return 5/x;
  }
}
```

MISRAC2012-Rule-1.3_h

Synopsis

A local variable is not checked against `0` before it is used as a divisor.

Enabled by default

Yes

Severity/Certainty

Medium/Low

Full description

(Required) There shall be no occurrence of undefined or critical unspecified behaviour If the variable has a value of 0, then a `divide by zero' runtime error will occur.

Coding standards

CWE 369

Divide By Zero

MISRA C:2012 Rule-1.3

(Required) There shall be no occurrence of undefined or critical unspecified behaviour

Code examples

The following code example fails the check and will give a warning:

```
int rand();

int example() {
    int x = rand();
    return 5/x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int rand();

int example() {
  int x = rand();
  if (x != 0){
    return 5/x;
  }
}
```

MISRAC2012-Rule-2.1_a

Synopsis

A case statement within a switch statement is unreachable.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) A project shall not contain unreachable code the switch's expression cannot have the value of the case's label. This often occurs because of literal values having been assigned to the switch condition. An unreachable case is not inherently dangerous, but may represent a misunderstanding of program behavior on the programmer's part.

Coding standards

CERT MSC07-C

Detect and remove dead code

MISRA C:2012 Rule-2.1

(Required) A project shall not contain unreachable code

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int x = 42;

  switch(2 * x) {
  case 42 :  //unreachable case, as x is 84
    ;
  default :
    ;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int x = 42;

  switch(2 * x) {
  case 84 :
    ;
  default :
    ;
  }
}
```

MISRAC2012-Rule-2.1_b

Synopsis

In all executions, a part of the program is not executed.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) A project shall not contain unreachable code Though not necessarily a problem, dead code can indicate programmer confusion about the program's branching structure.

Coding standards

CERT MSC07-C

Detect and remove dead code

CWE 561

Dead Code

MISRA C:2012 Rule-2.1

(Required) A project shall not contain unreachable code

Code examples

The following code example fails the check and will give a warning:

```
#include <stdio.h>

int f(int mode) {
    switch (mode) {
        case 0:
            return 1;
            printf("Hello!"); // This line cannot execute.
        default:
            return -1;
    }
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdio.h>

int f(int mode) {
    switch (mode) {
        case 0:
            printf("Hello!"); // This line can execute.
            return 1;
        default:
            return -1;
    }
}
```

MISRAC2012-Rule-2.2_a

Synopsis

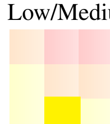
A statement that potentially contains no side effects.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) There shall be no dead code

Coding standards

CERT MSC12-C

Detect and remove code that has no effect

CWE 482

Comparing instead of Assigning

MISRA C:2012 Rule-2.2

(Required) There shall be no dead code

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int x = 1;
  x = 2;
  x < x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <string>
#include "iar.h"

void f();
template<class T>
struct X {
  int x;

  int get() const {
    return x;
  }

  X(int y) :
    x(y) {}

};

typedef X<int> intX;

void example(void) {
  /* everything below has a side-effect */
  int i=0;
  f();
  (void)f();
  ++i;
  i+=1;
  i++;
  char *p = "test";
  STD string s;
  s.assign(p);
  STD string *ps = &s;
  ps -> assign(p);
  intX xx(1);
  xx.get();
  intX(1);
}
```

MISRAC2012-Rule-2.2_c

Synopsis	A variable is assigned a value that is never used.
Enabled by default	Yes

Severity/Certainty

Low/Medium

Full description

(Required) There shall be no dead code destroys that value before it is used. This check does not detect situations where the value is simply lost when the function ends. This is not inherently dangerous, but may indicate an oversight or lack of understanding of the program behavior.

Coding standards

MISRA C:2012 Rule-2.2

(Required) There shall be no dead code

Code examples

The following code example fails the check and will give a warning:

```
int example(void) {
  int x;

  x = 20;

  x = 3;
  return 0;
}
#include <stdlib.h>

void ex(void) {
  int *p = 0;
  int *q = 0;
  p = malloc(sizeof(int));
  q = malloc(sizeof(int));
  p = q;  //p is not used after this assignment
  return;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

int *ex(void) {
  int *p;
  p = malloc(sizeof(int));
  return p; //the value is returned
}
int example(void) {
  int x;

  x = 20;

  return x;
}
```

MISRAC2012-Rule-2.7

Synopsis

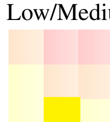
A function parameter is declared but not used.

Enabled by default

No

Severity/Certainty

Low/Medium



Full description

(Advisory) There should be no unused parameters in functions For example, the function may need to observe some calling protocol, or in C++ it may be a virtual function which doesn't need as much information from its arguments as related classes' equivalent functions do. Often, though, the warning indicates a genuine error.

Coding standards

CWE 563

Unused Variable

MISRA C:2012 Rule-2.7

(Advisory) There should be no unused parameters in functions

Code examples

The following code example fails the check and will give a warning:

```
int example(int x) {
  /* `x' is not used */
  return 20;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(int x) {
  return x + 20;
}
```

MISRAC2012-Rule-3.1

Synopsis

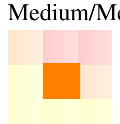
The character sequences /* and // shall not be used within a comment

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The character sequences /* and // shall not be used within a comment

Coding standards

MISRA C:2012 Rule-3.1

(Required) The character sequences /* and // shall not be used within a comment

Code examples

The following code example fails the check and will give a warning:

```
// This is /* a comment
```

The following code example passes the check and will not give a warning about this issue:

```
// This is a comment
```

MISRAC2012-Rule-4.2

Synopsis

Uses of trigraphs (in string literals only)

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

(Advisory) Trigraphs should not be used

Coding standards

MISRA C:2012 Rule-4.2

(Advisory) Trigraphs should not be used

Code examples

The following code example fails the check and will give a warning:

```
void func()
{
  char * str = "abc??!def";
}
```

The following code example passes the check and will not give a warning about this issue:

```
void func()
{
  char * str = "abc??def";
}
```

MISRAC2012-Rule-5.1

Synopsis

An external identifier is not unique for the first 31 characters but not identical

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) External identifiers shall be distinct

Coding standards

MISRA C:2012 Rule-5.1

(Required) External identifiers shall be distinct

Code examples

The following code example fails the check and will give a warning:

```
int ABC;

void example (void) {

}
```

The following code example passes the check and will not give a warning about this issue:

```
int a;

void example (void) {

}
```

MISRAC2012-Rule-5.3_a

Synopsis

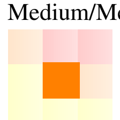
The definition of a local variable hides a global definition.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) An identifier declared in an inner scope shall not hide an identifier declared in an outer scope This may be intentional, but a different name should be used in case a reference to the global variable is attempted, and the local value changed or returned accidentally.

Coding standards

CERT DCL01-C

Do not reuse variable names in subscopes

CERT DCL01-CPP

Do not reuse variable names in subscopes

MISRA C:2012 Rule-5.3

(Required) An identifier declared in an inner scope shall not hide an identifier declared in an outer scope

Code examples

The following code example fails the check and will give a warning:

```
int x;


int foo (int y ){
  int x=0;
  x++;
  return x+y;

}
```

The following code example passes the check and will not give a warning about this issue:

```
int x;


int foo (int y ){

  x++;
  return x+y;

}
```

MISRAC2012-Rule-5.3_b

Synopsis

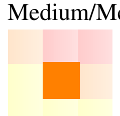
The definition of a local variable hides a previous local definition.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) An identifier declared in an inner scope shall not hide an identifier declared in an outer scope This may be intentional, but a different name should be used in case a reference to the outer variable is attempted, and the inner value changed or returned accidentally.

Coding standards

CERT DCL01-C

Do not reuse variable names in subscopes

CERT DCL01-CPP

Do not reuse variable names in subscopes

MISRA C:2012 Rule-5.3

(Required) An identifier declared in an inner scope shall not hide an identifier declared in an outer scope

Code examples

The following code example fails the check and will give a warning:

```
int foo(int x ){

  for (int y= 0; y < 10 ; y++){

    for (int y = 0; y < 100; y ++){
      return x+y;
    }
  }
  return x;
}


int foo2(int x){
  int y = 10;

  for (int y= 0; y < 10 ; y++)
    x++;
    return x;
}

int foo3(int x){

  int y = 10;
  {
    int y = 100;
    return x + y;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

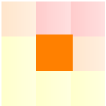
```
int foo(int x){

  for (int y=0; y < 10; y++)
    x++;
  for (int y=0; y < 10; y++)
    x++;
  return x;
}
```

MISRAC2012-Rule-5.3_c

Synopsis
A variable declaration hides a parameter of the function

Enabled by default
Yes

Severity/Certainty
Medium/Medium



Full description
(Required) An identifier declared in an inner scope shall not hide an identifier declared in an outer scope This may be intentional, but a different name should be used in case a reference to the argument is attempted, and the inner value changed or returned accidentally.

Coding standards
CERT DCL01-C

Do not reuse variable names in subscopes

CERT DCL01-CPP

Do not reuse variable names in subscopes

MISRA C:2012 Rule-5.3

(Required) An identifier declared in an inner scope shall not hide an identifier declared in an outer scope

Code examples
The following code example fails the check and will give a warning:

```
int foo(int x){


  for (int x = 0; x < 100; x++);

  return x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int foo(int x){
  int y;

  return x;
}
```

MISRAC2012-Rule-5.4_c89

Synopsis

Macro names that are not distinct in their first 31 characters from their macro parameters or other macro names

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Macro identifiers shall be distinct

Coding standards

MISRA C:2012 Rule-5.4

(Required) Macro identifiers shall be distinct

Code examples

The following code example fails the check and will give a warning:

```
/* MISRA C 2012 Rule 5.4 Example */

/*       1234567890123456789012345678901*********
Characters */
#define  engine_exhaust_gas_temperature_raw     egt_r
#define  engine_exhaust_gas_temperature_scaled  egt_s   /*
Non-compliant */
```

The following code example passes the check and will not give a warning about this issue:

```
/* MISRA C 2012 Rule 5.4 Example */

/*       1234567890123456789012345678901*********
Characters */
#define  engine_exhaust_gas_temp_raw            egt_r
#define  engine_exhaust_gas_temp_scaled         egt_s   /*
Compliant     */
```

MISRAC2012-Rule-5.4_c99

Synopsis

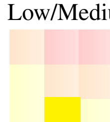
Macro names that are not distinct in their first 63 characters from their macro parameters or other macro names

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) Macro identifiers shall be distinct

Coding standards

MISRA C:2012 Rule-5.4

(Required) Macro identifiers shall be distinct

Code examples

The following code example fails the check and will give a warning:

```
/* MISRA C 2012 Rule 5.4 Example */

/*
123456789012345678901234567890123456789012345678901234567890123**
******             Characters */
#define
engine_exhaust_gas_temperature_blablablablablablablablablabla_
raw    egt_r
#define
engine_exhaust_gas_temperature_blablablablablablablablablabla_
scaled egt_s /* Non-compilant */
```

The following code example passes the check and will not give a warning about this issue:

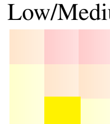
```
/* MISRA C 2012 Rule 5.4 Example */

/*
1234567890123456789012345678901234567890123456789012345678901234567890123**
******              Characters */
#define
engine_exhaust_gas_temperature_raw_blablablablablablablablablabla
bla    egt_r
#define
engine_exhaust_gas_temperature_scaled_blablablablablablablablabla
blabla egt_s /* Compilant     */
```

MISRAC2012-Rule-5.5_c89

Synopsis
Non-macro identifiers that are not distinct in their first 31 characters from macro names

Enabled by default
Yes

Severity/Certainty
Low/Medium



Full description
(Required) Identifiers shall be distinct from macro names

Coding standards
MISRA C:2012 Rule-5.5

(Required) Identifiers shall be distinct from macro names

Code examples
The following code example fails the check and will give a warning:

```
/* MISRA C 2012 Rule 5.5 Example */

#include "mc2_types.h"

#define Sum(x, y) ( ( x )  + ( y ) )

int16_t Sum;
```

The following code example passes the check and will not give a warning about this issue:

```
/* MISRA C 2012 Rule 5.5 Example */

#include "mc2_types.h"

#define Sum(x, y) ( ( x )  + ( y ) )

int16_t x = Sum ( 1, 2 );
```

MISRAC2012-Rule-5.5_c99

Synopsis

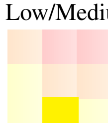
Non-macro identifiers that are not distinct in their first 63 characters from macro names

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) Identifiers shall be distinct from macro names

Coding standards

MISRA C:2012 Rule-5.5

(Required) Identifiers shall be distinct from macro names

Code examples

The following code example fails the check and will give a warning:

```
/* MISRA C 2012 Rule 5.5 Example */

#include "mc2_types.h"

#define Sum(x, y) ( ( x )  + ( y ) )

int16_t Sum;
```

The following code example passes the check and will not give a warning about this issue:

```
/* MISRA C 2012 Rule 5.5 Example */

#include "mc2_types.h"

#define Sum(x, y) ( ( x )  + ( y ) )

int16_t x = Sum ( 1, 2 );
```

MISRAC2012-Rule-5.6

Synopsis
Typedef with this name already declared.

Enabled by default
Yes

Severity/Certainty
Low/Medium

Full description
(Required) A typedef name shall be a unique identifier

Coding standards
MISRA C:2012 Rule-5.6

(Required) A typedef name shall be a unique identifier

Code examples
The following code example fails the check and will give a warning:

```
typedef int WIDTH;
//dummy comment
void f1()
{
  WIDTH w1;
}

void f2()
{
  typedef float WIDTH;
  WIDTH w2;
  WIDTH w3;
}
```

The following code example passes the check and will not give a warning about this issue:

```
namespace NS1
{
  typedef int WIDTH;
}
// f2.cc
namespace NS2
{
  typedef float WIDTH; // Compliant - NS2::WIDTH is not the same
as NS1::WIDTH
}
NS1::WIDTH w1;
NS2::WIDTH w2;
```

MISRAC2012-Rule-5.7

Synopsis

A class, struct, union or enum declaration that clashes with a previous declaration.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) A tag name shall be a unique identifier

Coding standards

MISRA C:2012 Rule-5.7

(Required) A tag name shall be a unique identifier

Code examples

The following code example fails the check and will give a warning:

```
void f1()
{
  class TYPE {};
}

void f2()
{
  float TYPE; // non-compliant
}
```

The following code example passes the check and will not give a warning about this issue:

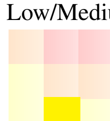
```
enum ENS {ONE, TWO };

void f1()
{
  class TYPE {};
}

void f4()
{
  union GRRR {
    int i;
    float f;
  };
}
```

MISRAC2012-Rule-5.8

Synopsis
External identifier names should be unique

Enabled by default
Yes

Severity/Certainty
Low/Medium



Full description
(Required) Identifiers that define objects or functions with external linkage shall be unique

Coding standards
MISRA C:2012 Rule-5.8

> (Required) Identifiers that define objects or functions with external linkage shall be unique

Code examples
The following code example fails the check and will give a warning:

```
/* file1.c */
#include <stdint.h>
void foo ( void ) /* "foo" has external linkage */
{
  int16_t index; /* "index" has no linkage */
}
```

The following code example passes the check and will not give a warning about this issue:

```
/* file1.c */
#include <stdint.h>
int32_t count; /* "count" has external linkage */
void foo ( void ) /* "foo" has external linkage */
{
  int16_t index; /* "index" has no linkage */
}
```

MISRAC2012-Rule-6.1

Synopsis

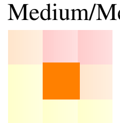
Bitfields with plain int type

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Bit-fields shall only be declared with an appropriate type

Coding standards

MISRA C:2012 Rule-6.1

(Required) Bit-fields shall only be declared with an appropriate type

Code examples

The following code example fails the check and will give a warning:

```
struct bad {
   int x:3;
};
enum digs { ONE, TWO, THREE, FOUR };

struct bad {
   digs d:3;
};
```

The following code example passes the check and will not give a warning about this issue:

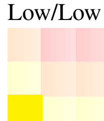
```
struct good {
  signed int x:3;
};
struct good {
  unsigned int x:3;
};
```

MISRAC2012-Rule-6.2

Synopsis
Signed single-bit fields (excluding anonymous fields)

Enabled by default
Yes

Severity/Certainty
Low/Low



Full description
(Required) Single-bit named bit fields shall not be of a signed type

Coding standards
MISRA C:2012 Rule-6.2

(Required) Single-bit named bit fields shall not be of a signed type

Code examples
The following code example fails the check and will give a warning:

```
struct S
{
  signed int a : 1; // Non-compliant
};
```

The following code example passes the check and will not give a warning about this issue:

```
struct S
{
  signed int b : 2;
  signed int   : 0;
  signed int   : 1;
  signed int   : 2;
};
```

MISRAC2012-Rule-7.1

Synopsis
Uses of octal integer constants

Enabled by default
Yes

Severity/Certainty
Low/Medium

Full description
(Required) Octal constants shall not be used

Coding standards
MISRA C:2012 Rule-7.1

(Required) Octal constants shall not be used

Code examples
The following code example fails the check and will give a warning:

```
void
func(void)
{
    int x = 077;
}
```

The following code example passes the check and will not give a warning about this issue:

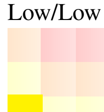
```
void
func(void)
{
    int x = 63;
}
```

MISRAC2012-Rule-7.2

Synopsis
A U suffix shall be applied to all constants of unsigned type.

Enabled by default
Yes

Severity/Certainty
Low/Low



Full description
(Required) A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type

Coding standards

MISRA C:2012 Rule-7.2

> (Required) A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  // 2147483648 -- does not fit in 31bits
  unsigned int x = 0x80000000;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  unsigned int x = 0x80000000u;
}
```

MISRAC2012-Rule-7.3

Synopsis

Lower case character 'l' should not be used as a suffix.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The lowercase character "l" shall not be used in a literal suffix

Coding standards

MISRA C:2012 Rule-7.3

> (Required) The lowercase character "l" shall not be used in a literal suffix

Code examples

The following code example fails the check and will give a warning:

```
#include "mc2_types.h"

void func()
{
  const int64_t b = 0l;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include "mc2_types.h"

void func()
{
    const int64_t a = 0L;
}
```

MISRAC2012-Rule-7.4_a

Synopsis

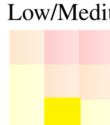
A string literal is assigned to a variable not declared as constant

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char"

Coding standards

MISRA C:2012 Rule-7.4

> (Required) A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char"

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  char *s = "Hello, World!";
}
```

The following code example passes the check and will not give a warning about this issue:

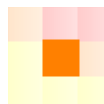
```
void example(void) {
  const char *s = "Hello, World!";
}
```

MISRAC2012-Rule-7.4_b

Synopsis
Part of string literal is modified via array subscript operator []

Enabled by default
Yes

Severity/Certainty
Medium/Medium



Full description
(Required) A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char"

Coding standards
MISRA C:2012 Rule-7.4

(Required) A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char"

Code examples
The following code example fails the check and will give a warning:

```
void example(void) {
  "012345"[0]++;
}
```

The following code example passes the check and will not give a warning about this issue:

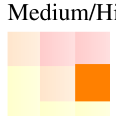
```
void example(void) {
  const char *c = "01234";
}
```

MISRAC2012-Rule-8.1

Synopsis
Whenever an object or function is declared or defined, its type shall be explicitly stated.

Enabled by default
Yes

Severity/Certainty
Medium/High



Full description
(Required) Types shall be explicitly specified

Coding standards
CERT DCL31-C

Declare identifiers before using them

MISRA C:2012 Rule-8.1

(Required) Types shall be explicitly specified

Code examples
The following code example fails the check and will give a warning:

```
void func(void)
{
    static y;
}
```

The following code example passes the check and will not give a warning about this issue:

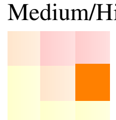
```
void func(void)
{
    int x;
}
```

MISRAC2012-Rule-8.2_a

Synopsis
Functions declared with an empty () parameter list that does not form a valid prototype

Enabled by default
Yes

Severity/Certainty
Medium/High



Full description
(Required) Function types shall be in prototype form with named parameters

Coding standards
CERT DCL20-C

Always specify void even if a function accepts no arguments

MISRA C:2012 Rule-8.2

(Required) Function types shall be in prototype form with named parameters

Code examples
The following code example fails the check and will give a warning:

```
void func();/* not a valid prototype in C */
void func2(void)
{
    func();
}
```

The following code example passes the check and will not give a warning about this issue:

```
void func(void);
void func2(void)
{
    func();
}
```

MISRAC2012-Rule-8.2_b

Synopsis Function prototypes must name all parameters

Enabled by default Yes

Severity/Certainty Low/High

Full description (Required) Function types shall be in prototype form with named parameters

Coding standards MISRA C:2012 Rule-8.2

(Required) Function types shall be in prototype form with named parameters

Code examples The following code example fails the check and will give a warning:

```
char *strchr(const char *, int c);

void func(void)
{
    strchr("hello, world!\n", '!');
}
```

The following code example passes the check and will not give a warning about this issue:

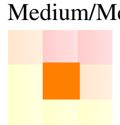
```
char *strchr(const char *s, int c);

void func(void)
{
    strchr("hello, world!\n", '!');
}
```

MISRAC2012-Rule-8.10

Synopsis
All inline functions should be declared as static

Enabled by default
Yes

Severity/Certainty
Medium/Medium



Full description
(Required) An inline function shall be declared with the static storage class

Coding standards
MISRA C:2012 Rule-8.10

(Required) An inline function shall be declared with the static storage class

Code examples
The following code example fails the check and will give a warning:

```
inline int example(int a) {
  return a + 1;
}
```

The following code example passes the check and will not give a warning about this issue:

```
inline static int example(int a) {
  return a + 1;
}
```

MISRAC2012-Rule-8.11

Synopsis
External arrays declared without size stated explicitly or defined implicitly by initialization.

Enabled by default
No

Severity/Certainty

Low/Medium

Full description

(Advisory) When an array with external linkage is declared, its size should be explicitly specified

Coding standards

MISRA C:2012 Rule-8.11

(Advisory) When an array with external linkage is declared, its size should be explicitly specified

Code examples

The following code example fails the check and will give a warning:

```
extern int a[];
```

The following code example passes the check and will not give a warning about this issue:

```
extern int a[10];
extern int b[] = { 0, 1, 2 };
```

MISRAC2012-Rule-8.14

Synopsis

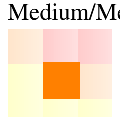
The use of the `restrict' type qualifier is forbidden for function parameters

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The restrict type qualifier shall not be used

Coding standards

MISRA C:2012 Rule-8.14

(Required) The restrict type qualifier shall not be used

Code examples

The following code example fails the check and will give a warning:

```
void example(void * restrict p, void * restrict q, int n) {
  printf("Bad function!\n");
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void * p, void * q, int n) {
  printf("Bad function!\n");
}
```

MISRAC2012-Rule-9.1_a

Synopsis

Possibly dereference of an uninitialized or NULL pointer.

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

(Mandatory) The value of an object with automatic storage duration shall not be read before it has been set This may result in memory corruption or a program crash. Pointer values should be initialized on all execution paths which result in a dereference, to avoid this.

Coding standards

CERT EXP33-C

Do not reference uninitialized memory

CWE 457

Use of Uninitialized Variable

CWE 824

Access of Uninitialized Pointer

MISRA C:2012 Rule-9.1

(Mandatory) The value of an object with automatic storage duration shall not be read before it has been set

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int *p;
  *p = 4;  //p is uninitialized
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int *p,a;
  p = &a;
  *p = 4;  //OK - p holds a valid address
}
```

MISRAC2012-Rule-9.1_b

Synopsis

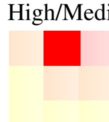
Checks reads from local buffers are preceded by writes.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

(Mandatory) The value of an object with automatic storage duration shall not be read before it has been set This is a semi-equivalent initialization check for arrays, which ensures that at least one element of the array has been written before any element is attempted to be read. A warning generally means that you have read an uninitialized value, and the program may behave erroneously or crash in some situations.

Coding standards

CERT EXP33-C

Do not reference uninitialized memory

CWE 457

Use of Uninitialized Variable

MISRA C:2012 Rule-9.1

(Mandatory) The value of an object with automatic storage duration shall not be read before it has been set

Code examples

The following code example fails the check and will give a warning:

```
void example() {
  int x[20];
  x[0] = 1;
  int b = x[1]; /* bad read but check can't detect which elements
*/
}
/* won't work until signature of memcpy is known */
#include <string.h>
void example() {
  int a[20];
  int b[20];
  memcpy(a,b,20);
}

/* read thru alias */
void example() {
  int x[20];
  int *a = x;
  int b = a[1]; /* read x thru alias a, but x not init */
}
void example() {
  int a[20];
  int b = a[1];
}
void example() {
  int x[20];
  *x = 1;
  int b = x[1]; /* bad read but check can't detect which elements
*/
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example() {
  int x[20];
  int *p = x;
  x[0]=1;
  int k = *p; /* read thru alias */
}
void example() {
  int x[20];
  int *p = x;
  p[0]=1; /* write thru alias */
  int k = *x;
}
struct X { int e; };
void example() {
  struct X x[20];
  x->e = 1;
  { struct X b = x[0]; } /* x[0] has been initialized via x->e,
but Goanna currently doesn't have pointer alias analysis on
individual array elements */
}
void example() {
  int x[20];
  *(x+0) = 1;
  int b = x[1]; /* bad read but check can't detect which elements
*/
}
extern void f(int*);
void example() {
  int a[20];
  f(a);
  int b = a[1];
}
void example() {
  int a[20] =
{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
  int b = a[1];
}
void example() {
  int x[20];
  *x = 1;
  int b = x[1]; /* bad read but check can't detect which elements
*/
}
/* write thru alias */
void example() {
  int x[20];
  int *a = x;
```

```
  f(a); /* assumed init of x thru alias a */
  int b = x[1];
}
void example() {
  int x[20];
  x[0] = 1;
  int b = x[1]; /* bad read but check can't detect which elements
*/
}
```

MISRAC2012-Rule-9.1_c

Synopsis

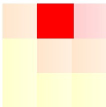
In all executions, a struct has one or more fields read before they are initialized.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

(Mandatory) The value of an object with automatic storage duration shall not be read before it has been set Using uninitialized values could lead to unexpected results or unpredictable program behavior, particularly in the case of pointer fields.

Coding standards

CERT EXP33-C

Do not reference uninitialized memory

CWE 457

Use of Uninitialized Variable

MISRA C:2012 Rule-9.1

(Mandatory) The value of an object with automatic storage duration shall not be read before it has been set

Code examples

The following code example fails the check and will give a warning:

```
struct st {
  int x;
  int y;
};

void example(void) {
  int a;
  struct st str;
  a = str.x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
struct st {
  int x;
  int y;
};

void example(int i) {
  int a;
  struct st str;
  str.x = i;
  a = str.x;
}
```

MISRAC2012-Rule-9.1_d

Synopsis

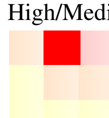
A field of a local struct is read before it is initialized.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

(Mandatory) The value of an object with automatic storage duration shall not be read before it has been set

Coding standards

CERT EXP33-C

Do not reference uninitialized memory

CWE 457

Use of Uninitialized Variable

MISRA C:2012 Rule-9.1

(Mandatory) The value of an object with automatic storage duration shall not be read before it has been set

Code examples

The following code example fails the check and will give a warning:

```
struct st {
  int x;
  int y;
};

void example(void) {
  int a;
  struct st str;
  a = str.x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
struct st {
  int x;
  int y;
};

void example(void) {
  int a;
  struct st str;
  str.x = 0;
  a = str.x;
}
```

MISRAC2012-Rule-9.1_e

Synopsis

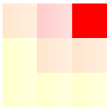
In all executions, a variable is read before it is assigned a value.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

(Mandatory) The value of an object with automatic storage duration shall not be read before it has been set value. Different paths may result in reading a variable at different program points. Whichever path is executed, uninitialized data is read, and behavior may consequently be unpredictable.

Coding standards

CERT EXP33-C

Do not reference uninitialized memory

CWE 457

Use of Uninitialized Variable

MISRA C:2012 Rule-9.1

(Mandatory) The value of an object with automatic storage duration shall not be read before it has been set

Code examples

The following code example fails the check and will give a warning:

```
int main(void) {
  int x;

  x++;  //x is uninitialized

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int main(void) {
  int x = 0;

  x++;

  return 0;
}
```

MISRAC2012-Rule-9.1_f

Synopsis

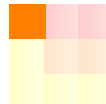
In some execution, a variable is read before it is assigned a value.

Enabled by default

Yes

Severity/Certainty

High/Low



Full description

(Mandatory) The value of an object with automatic storage duration shall not be read before it has been set There may be some execution paths where the variable is assigned a value before it is read. In such cases behavior may be unpredictable.

Coding standards

CWE 457

Use of Uninitialized Variable

MISRA C:2012 Rule-9.1

(Mandatory) The value of an object with automatic storage duration shall not be read before it has been set

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

int main(void) {
  int x, y;

  if (rand()) {
    x = 0;
  }

  y = x;  //x may not be initialized

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

int main(void) {
  int x;

  if (rand()) {
    x = 0;
  }

  /* x never read */

  return 0;
}
```

MISRAC2012-Rule-9.3

Synopsis

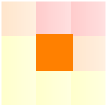
Arrays shall not be partially initialized

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Arrays shall not be partially initialized

Coding standards

MISRA C:2012 Rule-9.3

(Required) Arrays shall not be partially initialized

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int y[3][4] = { { 1, 2, 3 }, { 4, 5, 6 } };
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int y[3][2] = { { 1, 2 }, { 3, 4 }, { 5, 6 } };
}
```

MISRAC2012-Rule-9.5_a

Synopsis

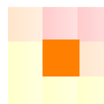
Arrays initialized with designated initializers must have a fixed length

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly

Coding standards

MISRA C:2012 Rule-9.5

(Required) Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int a1[] = { [0] = 1 };
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int a1[10] = { [0] = 1 };
}
```

MISRAC2012-Rule-9.5_b

Synopsis

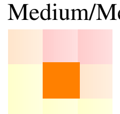
Flexible array members cannot be initalized with a designated initalizer

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly

Coding standards

MISRA C:2012 Rule-9.5

(Required) Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly

Code examples

The following code example fails the check and will give a warning:

```
struct A {
        int x;
        int y [];
};
struct A a1 = {1,{[1]=2}};

void example (void) {

}
```

The following code example passes the check and will not give a warning about this issue:

```
struct A {
        int x;
        int y [2];
};
struct A a1 = {1,{[1]=2}};

void example (void) {

}
```

MISRAC2012-Rule-10.1_R2

Synopsis

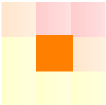
An expression of essentially Boolean type should always be used where an operand is interpreted as a Boolean value

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Operands shall not be of an inappropriate essential type

Coding standards

MISRA C:2012 Rule-10.1

(Required) Operands shall not be of an inappropriate essential type

Code examples

The following code example fails the check and will give a warning:

```
void func(int * ptr)
{
  if (!ptr) {}
}
void func()
{
  if (!0) {}
}
void example(void) {
   int x = 0;
   int y = 1;
   int a = x || y << 2;
}
void example(void) {
   int x = 0;
   int y = 1;
   int a = 5;
   (a + (x || y)) ? example() : example();
}
void example(void) {
   int x = 5;
   int y = 11;
   if (x || y) {
   }
}
void example(void) {

   int d, c, b, a;

   d = ( c & a ) && b;

}
```

The following code example passes the check and will not give a warning about this issue:

```
bool test()
{
  return true;
}

void example(void) {
  if(test()) {}
}
typedef charboolean_t;/* Compliant: Boolean-by-enforcement */

void example(void)
{
    boolean_t d;
    boolean_t c = 1;
    boolean_t b = 0;
    boolean_t a = 1;

    d = ( c && a ) && b;

}
void func(bool * ptr)
{
  if (*ptr) {}
}
typedef intboolean_t;

void example(void) {
    boolean_t x = 0;
    boolean_t y = 1;
    boolean_t a = 0;
    if (a && (x || y)) {
    }
}
void example(void) {
   int x = 0;
   int y = 1;
   int a = x == y;
}
#include <stdbool.h>

void example(void) {
    bool x = false;
    bool y = true;
    if (x || y) {
    }
}
typedef charboolean_t;
```

```
void example(void) {
    boolean_t x = 0;
    boolean_t y = 1;
    boolean_t a = x || y;
    a ? example() : example();
}
```

MISRAC2012-Rule-10.1_R3

Synopsis

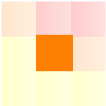
An operand of essentially Boolean type should noe be used where an operand is interpreted as a numeric value

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Operands shall not be of an inappropriate essential type

Coding standards

MISRA C:2012 Rule-10.1

(Required) Operands shall not be of an inappropriate essential type

Code examples

The following code example fails the check and will give a warning:

```
void func(bool b)
{
  bool x;
  bool y;
  y = x % b;
}
void example(void) {
   int x = 0;
   int y = 1;
   int a = 5;
   (a + (x || y)) ? example() : example();
}
void example(void) {}
void example(void) {
   int x = 0;
   int y = 1;
   int a = (x == y) << 2;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int
isgood(int ch)
{
    return (ch & 0x80) == 0;
}

int example(int r, int f1, int f2)
{
  if (r && f1 == f2)
    return 1;
  else
    return 0;
}
bool test()
{
  return true;
}

void example(void) {
  if(test()) {}
}
typedef charboolean_t;/* Compliant: Boolean-by-enforcement */

void example(void)
{
    boolean_t d;
    boolean_t c = 1;
    boolean_t b = 0;
    boolean_t a = 1;

    d = ( c && a ) && b;

}
class foo {
  int val;
public:
  bool operator==(const foo &rhs) const { return val == rhs.val;
}
};

int example(bool r, const foo &f1, const foo &f2)
{
  if (r && f1 == f2)
    return 1;
  else
    return 0;
}
```

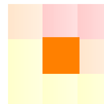
```
void func(bool * ptr)
{
  if (*ptr) {}
}
void func()
{
  bool x;
  bool y;
  y = x && y;
}
typedef intboolean_t;

void example(void) {
    boolean_t x = 0;
    boolean_t y = 1;
    boolean_t a = 0;
    if (a && (x || y)) {
    }
}
void example(void) {
   int x = 0;
   int y = 1;
   int a = x == y;
}
#include <stdbool.h>

void example(void) {
    bool x = false;
    bool y = true;
    if (x || y) {
    }
}
typedef charboolean_t;
void example(void) {
    boolean_t x = 0;
    boolean_t y = 1;
    boolean_t a = x || y;
    a ? example() : example();
}
```

MISRAC2012-Rule-10.1_R4

Synopsis	An operand of essentially character type should not be used where an operand is interpreted as a numeric value
Enabled by default	Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Operands shall not be of an inappropriate essential type

Coding standards

MISRA C:2012 Rule-10.1

(Required) Operands shall not be of an inappropriate essential type

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  char a = 'a';
  a << 1;
}
void example(void) {
  char a = 'a';
  char b = 'b';
  a & b;
}
void example(void) {
  char a = 'a';
  char b = 'b';
  char c;
  c = a * b;
}
void example(void) {
  int a[10];
  char b;
  a[b]++;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  char a = 'a';
  char b = 'b';
  char c;
  c = a + b;
}
```

MISRAC2012-Rule-10.1_R5

Synopsis

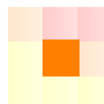
An operand of essentially enum type should not be used in an arithmetic operation because an enum object uses an implementation-defined integer type.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Operands shall not be of an inappropriate essential type An operation involving an enum object may therefore yield a result with an unexpected type. Note that an enumeration constant from an anonymous enum has seentially signed type

Coding standards

MISRA C:2012 Rule-10.1

(Required) Operands shall not be of an inappropriate essential type

Code examples

The following code example fails the check and will give a warning:

```
enum ens { ONE, TWO, THREE };

void func(ens b)
{
  ens x;
  bool y;
  y = x | b;
}
void example(void) {}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {}
enum ens { ONE, TWO, THREE };

void func(ens b)
{
  ens y;
  y = b;
}
```

MISRAC2012-Rule-10.1_R6

Synopsis

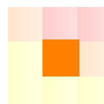
Shift and bitwise opearation should only be performed on operands of essentially unsigned type.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Operands shall not be of an inappropriate essential type The numeric value resulting from their use on essentially signed types is implemntation defined

Coding standards

MISRA C:2012 Rule-10.1

(Required) Operands shall not be of an inappropriate essential type

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int x = -(1U);

  x ^ 1;
  x & 0x7F;
  ((unsigned int)x) & 0x7F;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int x = -1;
  ((unsigned int)x) ^ 1U;
  2U ^ 1U;
  ((unsigned int)x) & 0x7FU;
  ((unsigned int)x) & 0x7FU;
}
```

MISRAC2012-Rule-10.1_R7

Synopsis

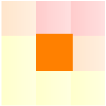
The right hand operand of a shift operator should be of essentially unsigned type to ensure that undefined behaviour does not result from a negative shift.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Operands shall not be of an inappropriate essential type

Coding standards

MISRA C:2012 Rule-10.1

(Required) Operands shall not be of an inappropriate essential type

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int a;
  unsigned int b;
  b << a;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  unsigned int a;
  unsigned int b;
  b << a;
}
```

MISRAC2012-Rule-10.1_R8

Synopsis

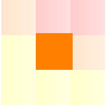
An operand of essentially unsigned typed should not be used as the operand to the unary minus operator, as the signedness of the result is determined by the implementation size of int

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Operands shall not be of an inappropriate essential type

Coding standards
MISRA C:2012 Rule-10.1

(Required) Operands shall not be of an inappropriate essential type

Code examples
The following code example fails the check and will give a warning:

```
void example(void) {
  unsigned int max = -1U;
  // use max = ~0U;
}
```

The following code example passes the check and will not give a warning about this issue:

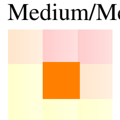
```
void example(void) {
  int neg_one = -1;
}
```

MISRAC2012-Rule-10.2

Synopsis
Expressions of essentially character type shall not be used inapproriately in addition and subtraction operations

Enabled by default
Yes

Severity/Certainty
Medium/Medium



Full description
(Required) Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations

Coding standards
MISRA C:2012 Rule-10.2

(Required) Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations

Code examples
The following code example fails the check and will give a warning:

```
typedef enum test {
  one,
  two,
  three
} myEnum;

void example(void) {
  char a = 'a' - two;
}
void example(void) {
  int a = 5;
  char c = (a == 10) + '0';
}
void example(void) {
  char a = 10 - 'a';
}
void example(void) {
  char a = 'a' - (10 == 5);
}
void example(void) {
  double a = 1.00f;
  char c = 'a' - a;
}
void example(void) {
  char a = '9';
  char c = a + '0';
}
typedef enum test {
  one,
  two,
  three
} myEnum;

void example(void) {
  myEnum a = one;
  char c = a + '0';
}
void example(void) {
  double a = 1.00f;
  char c = a + '0';
}
enum {
  one,
  two,
  three
} myEnum;
```

```
#define four 4

void example(void) {
  char c = one + '0';
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  unsigned int a = 9;
  char dig = a + '0';
}
void example(void) {
  int a = 9;
  char dig = a + '0';
}
void example(void) {
  int a = 9;
  char b = 'a' - a;
}
#include <stdint.h>

void example (void) {
  uint8_t a = 5;
  '0' + a;
}
void example(void) {
  unsigned int a = 9;
  char b = 'a' - a;
}
void example(void) {
  char a = '9';
  char b = 'a' - a;
}
#include <stdint.h>

void example (void) {
  int8_t a = 5;
  a + '0';
}
```

MISRAC2012-Rule-10.3

Synopsis

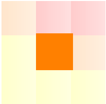
The value of an expression shall not be assigned to an object with a narrower essential type or a different essential type category

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category

Coding standards

MISRA C:2012 Rule-10.3

> (Required) The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  char a = 'a';
  unsigned int b = 10;
  b = a;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  unsigned int a = 10;
  unsigned int b = 5;
  b = a;
}
```

MISRAC2012-Rule-10.4

Synopsis

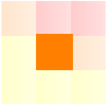
Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category

Coding standards

MISRA C:2012 Rule-10.4

> (Required) Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  unsigned int a = 5;
  float f = 0.001f;
  a + f;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int a = 10;
  int b = 10;
  a + b;
}
```

MISRAC2012-Rule-10.6

Synopsis

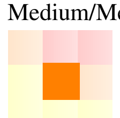
The value of a composite expression shall not be assigned to an object with wider essential type

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The value of a composite expression shall not be assigned to an object with wider essential type

Coding standards

MISRA C:2012 Rule-10.6

> (Required) The value of a composite expression shall not be assigned to an object with wider essential type

Code examples

The following code example fails the check and will give a warning:

```
#include <stdint.h>

void example(void) {
  uint16_t a = 5;
  uint16_t b = 10;
  uint32_t c;
  c = a + b;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdint.h>

void example(void) {
  uint16_t a;
  uint16_t b;
  b = a + a;
}
```

MISRAC2012-Rule-10.7

Synopsis

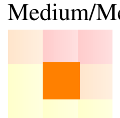
If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) 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

Coding standards

MISRA C:2012 Rule-10.7

(Required) 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

Code examples

The following code example fails the check and will give a warning:

```
/* MISRA C 2012 Rule 10.7 Example */

#include "mc2_types.h"
#include "mc2_1000.h"

extern uint32_t   u32a;
extern uint16_t   u16b;

void example(void) {
     u32a * ( u16a + u16b );    /* Implicit conversion of ( u16a +
u16b )      */
}
```

The following code example passes the check and will not give a warning about this issue:

```
/* MISRA C 2012 Rule 10.7 Example */

#include "mc2_types.h"
#include "mc2_1000.h"

extern uint32_t   u32a;
extern uint16_t   u16b;

void example(void) {
      u32a * u16a + u16b;                   /* No composite
conversion       */
}
```

MISRAC2012-Rule-10.8

Synopsis

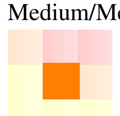
The value of a composite expression shall not be cast to a different essential type category or a wider essential type

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The value of a composite expression shall not be cast to a different essential type category or a wider essential type

Coding standards

MISRA C:2012 Rule-10.8

(Required) The value of a composite expression shall not be cast to a different essential type category or a wider essential type

Code examples

The following code example fails the check and will give a warning:

```
#if __FLOAT_SIZE__ == __DOUBLE_SIZE__
  #error "IGNORE_TEST: double and float have same size"
#endif

void example(void) {
  float array[10];

  // arithmetic makes it a complex expression
  double x = (double)(array[5] + 3.0f);
}
void example(void) {
  int array[10];
  // complex expression cannot change sign
  unsigned int x = (unsigned int)(array[5] + 5);
}
void example(void) {
  int s16a = 3;
  int s16b = 3;

  // arithmetic makes it a complex expression
  long long x = (long long)(s16a + s16b);
}
void example(void) {
  int array[10];
  // complex expression cannot change type
  float x = (float)(array[5] + 5);
}
```

The following code example passes the check and will not give a warning about this issue:

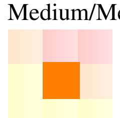
```
void example(void) {
  int array[10];
  // non-complex expression can change type
  float x = (float)(array[5]);
}
void example(void) {
  int array[10];

  // A non complex expression is considered safe
  long x = (long)(array[5]);
}
void example(void) {
  int array[10];

  // non-complex expressions can change sign
  unsigned int x = (unsigned int)(array[5]);
}
void example(void) {
  float array[10];

  // A non complex expression is considered safe
  double x = (double)(array[5]);
}
```

MISRAC2012-Rule-11.1

Synopsis
Conversion shall not be performed between a pointer to a function and any other type

Enabled by default
Yes

Severity/Certainty
Medium/Medium



Full description
(Required) Conversions shall not be performed between a pointer to a function and any other type

Coding standards
MISRA C:2012 Rule-11.1

(Required) Conversions shall not be performed between a pointer to a function and any other type

Code examples
The following code example fails the check and will give a warning:

```
#include <stdlib.h>

void example(void) {

  int (*fptr)(int,int);

  (int*)fptr;

}
```

The following code example passes the check and will not give a warning about this issue:

```
/* MISRA C 2012 Rule 11.1 Example */

#include "mc2_types.h"

typedef void ( *fp16 ) ( int16_t n );

typedef fp16 ( *pfp16 ) ( void );

void example(void) {
    pfp16 pfp1;

    ( void ) ( *pfp1 ( ) );   /* Compliant - exception 2 - cast
function
                           * pointer into void
*/
}
```

MISRAC2012-Rule-11.3

Synopsis

A pointer to object type is cast to a pointer to different object type

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) A cast shall not be performed between a pointer to object type and a pointer to a different object type Conversions of this type may be invalid if the new pointer type required a stricter alignment.

Coding standards

MISRA C:2012 Rule-11.3

> (Required) A cast shall not be performed between a pointer to object type and a pointer to a different object type

Code examples

The following code example fails the check and will give a warning:

```
typedef unsigned int uint32_t;
typedef unsigned char uint8_t;

void example(void) {
  uint8_t * p1;
  uint32_t * p2;
  p2 = (uint32_t *)p1;
}
```

The following code example passes the check and will not give a warning about this issue:

```
typedef unsigned int uint32_t;
typedef unsigned char uint8_t;

void example(void) {
  uint8_t * p1;
  uint8_t * p2;
  p2 = (uint8_t *)p1;
}
```

MISRAC2012-Rule-11.4

Synopsis

A cast should not be performed between a pointer type and an integral type.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

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

Coding standards

MISRA C:2012 Rule-11.4

> (Advisory) A conversion should not be performed between a pointer to object and an integer type

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {

   int *p;
   int x;

   x = (int)p;

}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {

   int *p;
   int *x;

   x = p;

}
```

MISRAC2012-Rule-11.7

Synopsis

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

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

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

Coding standards

MISRA C:2012 Rule-11.7

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

Code examples

The following code example fails the check and will give a warning:

```
/* MISRA C 2012 Rule 11.7 Example */

#include "mc2_types.h"

int16_t  *p;
float32_t f;

void example(void) {
    f = ( float32_t ) p;   /* Non-compliant */
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include "mc2_types.h"
#include "mc2_1000.h"

void example(void) {
  int16_t *p;
  int32_t f;

  f = ( int32_t ) p;
}
```

MISRAC2012-Rule-11.8

Synopsis

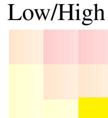
Casts that remove any const or volatile qualification.

Enabled by default

Yes

Severity/Certainty

Low/High



Full description

(Required) A cast shall not remove any const or volatile qualification from the type pointed to by a pointer This violates the principle of type qualification. This check does not look for changes to the qualification of the pointer during the cast.

Coding standards

MISRA C:2012 Rule-11.8

(Required) A cast shall not remove any const or volatile qualification from the type pointed to by a pointer

Code examples

The following code example fails the check and will give a warning:

```
typedef unsigned short uint16_t;

void example(void) {

  uint16_t x;
  const uint16_t *    pci;      /* pointer to const int */
  uint16_t *          pi;       /* pointer to int */

  pi = (uint16_t *)pci; // not compliant

}
```

The following code example passes the check and will not give a warning about this issue:

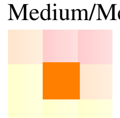
```
typedef unsigned short uint16_t;

void example(void) {

  uint16_t x;
  uint16_t * const    cpi = &x; /* const pointer to int */
  uint16_t *          pi;       /* pointer to int */

  pi = cpi; // compliant - no cast required

}
```

MISRAC2012-Rule-11.9

Synopsis
An integer constant is used where the NULL macro should be

Enabled by default
Yes

Severity/Certainty
Medium/Medium



Full description
(Required) The macro NULL shall be the only permitted form of integer null pointer constant

Coding standards
MISRA C:2012 Rule-11.9

> (Required) The macro NULL shall be the only permitted form of integer null pointer constant

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

void example(void) {
  char *a = malloc(sizeof(char) * 10);
  if (a != 0) {
    *a = 5;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

void example(void) {
  int *a = malloc(sizeof(int) * 10);
  if (a != NULL) {
    *a = 5;
  }
}
```

MISRAC2012-Rule-12.1

Synopsis

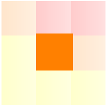
Add parentheses to avoid implicit operator precedence.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

(Advisory) The precedence of operators within expressions should be made explicit

Coding standards

MISRA C:2012 Rule-12.1

(Advisory) The precedence of operators within expressions should be made explicit

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
    int i;
    int j;
    int k;
    int result;

    result = i + j * k;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
    int i;
    int j;
    int k;
    int result;

    result = i + (j - k);
}
```

MISRAC2012-Rule-12.2

Synopsis

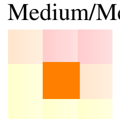
Out of range shifts

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) 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 In this case, the right-hand operand may be negative, or too large. This check is for all platforms. The behavior in this situation is undefined; the code may work as intended, or data could become erroneous.

Coding standards

CERT INT34-C

Do not shift a negative number of bits or more bits than exist in the operand

CWE 682

Incorrect Calculation

MISRA C:2012 Rule-12.2

> (Required) 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

Code examples

The following code example fails the check and will give a warning:

```
unsigned int foo(unsigned long long x, unsigned int y)
{
  int shift = 65; // too big
  return 3ULL << shift;
}
unsigned int foo(unsigned int x, unsigned int y)
{
  int shift = 33; // too big
  return 3U << shift;
}
```

The following code example passes the check and will not give a warning about this issue:

```
unsigned int foo(unsigned int x)
{
  int y = 1;  // OK - this is within the correct range
  return x << y;
}
unsigned int foo(unsigned long long x)
{
  int y = 63; // ok
  return x << y;
}
```

MISRAC2012-Rule-12.3

Synopsis

Uses of the comma operator

Enabled by default

No

Severity/Certainty

Low/High

Full description

(Advisory) The comma operator should not be used

Coding standards MISRA C:2012 Rule-12.3

(Advisory) The comma operator should not be used

Code examples The following code example fails the check and will give a warning:

```
#include <string.h>

void reverse(char *string) {
  int i, j;
  j = strlen(string);
  for (i = 0; i < j; i++, j--) {
    char temp = string[i];
    string[i] = string[j];
    string[j] = temp;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <string.h>

void reverse(char *string) {
  int i;
  int length = strlen(string);
  int half_length = length / 2;
  for (i = 0; i < half_length; i++) {
    int opposite = length - i;
    char temp = string[i];
    string[i] = string[opposite];
    string[opposite] = temp;
  }
}
```

MISRAC2012-Rule-12.4

Synopsis Evaluation of constant expressions should not lead to unsigned integer wrap-around

Enabled by default No

Severity/Certainty Low/Medium

Full description | (Advisory) Evaluation of constant expressions should not lead to unsigned integer wrap-around

Coding standards | MISRA C:2012 Rule-12.4

(Advisory) Evaluation of constant expressions should not lead to unsigned integer wrap-around

Code examples | The following code example fails the check and will give a warning:

```
void example(void) {
   (0xFFFFFFFF + 1u);
}
```

The following code example passes the check and will not give a warning about this issue:

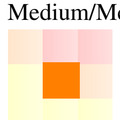
```
void example(void) {
   0x7FFFFFFF + 0;
}
```

MISRAC2012-Rule-13.1

Synopsis | The initalisation list of an array should not contain side effects

Enabled by default | Yes

Severity/Certainty | Medium/Medium



Full description | (Required) Initializer lists shall not contain persistent side effects

Coding standards | MISRA C:2012 Rule-13.1

(Required) Initializer lists shall not contain persistent side effects

Code examples | The following code example fails the check and will give a warning:

```
volatile int v1;

extern void p ( int a[2] );

int x = 10;

void example(void) {
  int a[2] = { v1, 0 };

  p( (int[2]) { x++, x-- });

}
```

The following code example passes the check and will not give a warning about this issue:

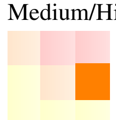
```
void example(void) {
  int a[2] = { 1, 2 };
}
```

MISRAC2012-Rule-13.2_a

Synopsis
Expressions which depend on order of evaluation

Enabled by default
Yes

Severity/Certainty
Medium/High



Full description
(Required) The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders expression with an unspecified evaluation order, between two consecutive sequence points. ANSI C does not specify an evaluation order for different parts of an expression. For this reason different compilers are free to perform their own optimizations regarding the evaluation order. Projects containing statements that violate this check are not readily ported between architectures or compilers, and their ports may prove difficult to debug. Only four operators have a guaranteed order of evaluation: logical AND (`a && b`) evaluates the left operand, then the right operand only if the left is found to be true; logical OR (`a || b`) evaluates the left operand, then the right operand only if the left is found to be false; a ternary conditional (`a ? b : c`) evaluates the first operand, then either the second or the third, depending on whether the first is found to be true or false; and a comma (`a , b`) evaluates its left operand before its right.

Coding standards

CERT EXP10-C

Do not depend on the order of evaluation of subexpressions or the order in which side effects take place

CERT EXP30-C

Do not depend on order of evaluation between sequence points

CWE 696

Incorrect Behavior Order

MISRA C:2012 Rule-13.2

(Required) The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders

Code examples

The following code example fails the check and will give a warning:

```
int main(void) {
  int i = 0;

  i = i * i++;  //unspecified order of operations

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int main(void) {
  int i = 0;
  int x = i;

  i++;
  x = x * i;  //OK - statement is broken up

  return 0;
}
```

MISRAC2012-Rule-13.2_b

Synopsis

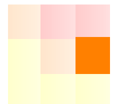
There shall be no more than one read access with volatile-qualified type within one sequence point

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

(Required) The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders

Coding standards

CERT EXP10-C

> Do not depend on the order of evaluation of subexpressions or the order in which side effects take place

CERT EXP30-C

> Do not depend on order of evaluation between sequence points

CWE 696

> Incorrect Behavior Order

MISRA C:2012 Rule-13.2

> (Required) The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders

Code examples

The following code example fails the check and will give a warning:

```
#include "mc2_types.h"
#include "mc2_header.h"

void example(void) {
  uint16_t x;
  volatile uint16_t v;
  x = v + v;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int main(void) {
  int i = 0;
  int x = i;

  i++;
  x = x * i;  //OK - statement is broken up

  return 0;
}
```

MISRAC2012-Rule-13.2_c

Synopsis

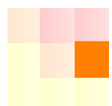
There shall be no more than one modification access with volatile-qualified type within one sequence point

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

(Required) The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders

Coding standards

CERT EXP10-C

Do not depend on the order of evaluation of subexpressions or the order in which side effects take place

CERT EXP30-C

Do not depend on order of evaluation between sequence points

CWE 696

Incorrect Behavior Order

MISRA C:2012 Rule-13.2

(Required) The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders

Code examples

The following code example fails the check and will give a warning:

```
#include "mc2_types.h"
#include "mc2_header.h"

void example(void) {
  uint16_t x;
  volatile uint16_t v, w;
  v = w = x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdbool.h>
void InitializeArray(int *);
const int *example(void)
{
  static volatile bool s_initialized = false;
  static int s_array[256];

  if (!s_initialized)
  {
    InitializeArray(s_array);
    s_initialized = true;
  }
  return s_array;
}
```

MISRAC2012-Rule-13.3

Synopsis

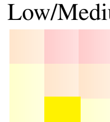
Uses of increment (++) and decrement (--) operators mixed with other operators in an expression.

Enabled by default

No

Severity/Certainty

Low/Medium



Full description

(Advisory) 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

Coding standards

MISRA C:2012 Rule-13.3

> (Advisory) 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

Code examples

The following code example fails the check and will give a warning:

```
void example(char *src, char *dst) {
  while ((*src++ = *dst++));
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(char *src, char *dst) {
   while (*src) {
      *dst = *src;
      src++;
      dst++;
   }
}
```

MISRAC2012-Rule-13.4_a

Synopsis

An assignment may be mistakenly used as the condition for an `if`, `for`, `while` or `do` statement.

Enabled by default

No

Severity/Certainty

Low/High

Full description

(Advisory) The result of an assignment operator should not be used This bug will likely result in incorrect program flow, and possibly an infinte loop.

Coding standards

CERT EXP18-C

Do not perform assignments in selection statements

CERT EXP19-CPP

Do not perform assignments in conditional expressions

CWE 481

Assigning instead of Comparing

MISRA C:2012 Rule-13.4

(Advisory) The result of an assignment operator should not be used

Code examples

The following code example fails the check and will give a warning:

```
int example(void) {
  int x = 2;
  if (x = 3)
    return 1;
  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(void) {
  int x = 2;
  if (x == 3)
    return 1;
  return 0;
}
```

MISRAC2012-Rule-13.4_b

Synopsis

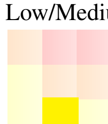
Assignment in a sub-expression.

Enabled by default

No

Severity/Certainty

Low/Medium



Full description

(Advisory) The result of an assignment operator should not be used

Coding standards

MISRA C:2012 Rule-13.4

(Advisory) The result of an assignment operator should not be used

Code examples

The following code example fails the check and will give a warning:

```
void func()
{
  int x;
  int y;
  int z;
  x = y = z;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void func()
{
  int x = 2;
  int y;
  int z;
  x = y;
  x == y;
}
```

MISRAC2012-Rule-13.5

Synopsis

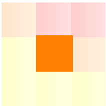
Right hand operands of && or || that contain side effects

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

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

Coding standards

CWE 768

Incorrect Short Circuit Evaluation

MISRA C:2012 Rule-13.5

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

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int i;
  int size = rand() && i++;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int i;
  int size = rand() && i;
}
```

MISRAC2012-Rule-13.6

Synopsis

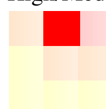
The operand of the sizeof operator shall not contain any expression which has potential side effects

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

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

Coding standards

CERT EXP06-C

Operands to the sizeof operator should not contain side effects

CERT EXP06-CPP

Operands to the sizeof operator should not contain side effects

MISRA C:2012 Rule-13.6

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

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int i;
  int size = sizeof(i++);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int i;
  int size = sizeof(i);
  i++;
}
```

MISRAC2012-Rule-14.1_a

Synopsis
Floating-point values in the controlling expression of a for statement.

Enabled by default
Yes

Severity/Certainty
Low/Medium

Full description
(Required) A loop counter shall not have essentially floating type

Coding standards
MISRA C:2012 Rule-14.1

(Required) A loop counter shall not have essentially floating type

Code examples
The following code example fails the check and will give a warning:

```
void example(int input, float f) {
  int i;
  for (i = 0; i < input && f < 0.1f; ++i) {
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(int input, float f) {
  int i;
  int f_condition = f < 0.1f;
  for (i = 0; i < input && f_condition; ++i) {
    f_condition = f < 0.1f;
  }
}
```

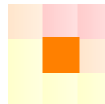
MISRAC2012-Rule-14.1_b

Synopsis
An essentially float variable, used in the loop condition, is modified in the loop body

Enabled by default
Yes

Severity/Certainty

Medium/Medium



Full description

(Required) A loop counter shall not have essentially floating type

Coding standards

MISRA C:2012 Rule-14.1

(Required) A loop counter shall not have essentially floating type

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int a = 10;
  float f = 0.001f;

  while (f < 1.00f) {
    f = f + (float) a;
    a++;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int a = 10;
  float f = 0.001f;

  while (a < 30) {
    f = f + (float) a;
    a++;
  }
}
```

MISRAC2012-Rule-14.2

Synopsis

A `for` loop counter variable is modified in the body of the loop.

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

(Required) A for loop shall be well-formed statement) should not be assigned to in the body of the for loop. While it's legal to modify the loop counter within the body of a `for` loop (in place of a `while` loop), the conventional use of a `for` loop is to iterate over a predetermined range, incrementing the loop counter once per iteration. Modification of the loop counter within the `for` loop body is probably accidental, and could result in erroneous behavior or an infinite loop.

Coding standards

MISRA C:2012 Rule-14.2

(Required) A for loop shall be well-formed

Code examples

The following code example fails the check and will give a warning:

```
int main(void) {
  int i;

  /* i is incremented inside the loop body */
  for (i = 0; i < 10; i++) {
    i = i + 1;
  }

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

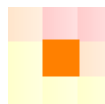
```
int main(void) {
  int i;
  int x = 0;

  for (i = 0; i < 10; i++) {
    x = i + 1;
  }

  return 0;
}
```

MISRAC2012-Rule-14.3_a

Synopsis The condition in if, for, while, do-while and ternary operator will always be met.

Enabled by default Yes

Severity/Certainty Medium/Medium



Full description (Required) Controlling expressions shall not be invariant

Coding standards CERT EXP17-C

Do not perform bitwise operations in conditional expressions

MISRA C:2012 Rule-14.3

(Required) Controlling expressions shall not be invariant

Code examples The following code example fails the check and will give a warning:

```
void example(void) {

    int x = 5;

    for (x = 0; x < 6 && 1; x--) {
    }
}
```

The following code example passes the check and will not give a warning about this issue:

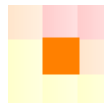
```
void example(void) {

    int x = 5;

    for (x = 0; x < 6 && 1; x++) {
    }
}
```

MISRAC2012-Rule-14.3_b

Synopsis The condition in if, for, while, do-while and ternary operator will never be met.

Enabled by default
Yes

Severity/Certainty
Medium/Medium



Full description
(Required) Controlling expressions shall not be invariant

Coding standards
CERT EXP17-C

Do not perform bitwise operations in conditional expressions

MISRA C:2012 Rule-14.3

(Required) Controlling expressions shall not be invariant

Code examples
The following code example fails the check and will give a warning:

```
void example(void) {

    int x = 5;

    for (x = 0; x < 6 && x >= 1; x++) {
    }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {

    int x = 5;

    for (x = 0; x < 6 && x >= 0; x++) {
    }
}
```

MISRAC2012-Rule-14.4_a

Synopsis
Non-boolean termination conditions in do ... while statements.

Enabled by default
Yes

Severity/Certainty

Low/Medium

Full description

(Required) The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type

Coding standards

MISRA C:2012 Rule-14.4

> (Required) The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type

Code examples

The following code example fails the check and will give a warning:

```
typedefintint32_t;
int32_t func();

void example(void)
{
  do {
  } while (func());
}
```

The following code example passes the check and will not give a warning about this issue:

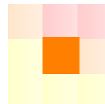
```
#include <stddef.h>

int * fn()
{
  int * ptr;
  return ptr;
}

int fn2()
{
  return 5;
}

bool fn3()
{
  return true;
}

void example(void)
{
  while (int *ptr = fn() )  // Compliant by exception
  {}

  do
  {
    int *ptr = fn();
    if ( NULL == ptr )
    {
      break;
    }
  }
  while (true); // Compliant

  while (int len = fn2() )  // Compliant by exception
  {}

  if (int *p = fn()) {}   // Compliant by exception
  if (int len = fn2() ) {} // Complioant by exception
  if (bool flag = fn3()) {} // Compliant
}
```

MISRAC2012-Rule-14.4_b

Synopsis	Non-boolean termination conditions in for loops.
Enabled by default	Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type

Coding standards

MISRA C:2012 Rule-14.4

> (Required) The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type

Code examples

The following code example fails the check and will give a warning:

```
void example(void)
{
  for (int x = 10;x;--x) {}
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stddef.h>

int * fn()
{
  int * ptr;
  return ptr;
}

int fn2()
{
  return 5;
}

bool fn3()
{
  return true;
}

void example(void)
{
  for (fn(); fn3(); fn2())  // Compliant
  {}

  for (fn(); true; fn()) // Compliant
  {
    int *ptr = fn();
    if ( NULL == ptr )
    {
      break;
    }
  }

  for (int len = fn2(); len < 10; len++)  // Compliant
    ;
}
```

MISRAC2012-Rule-14.4_c

Synopsis	Non-boolean conditions in if statements.
Enabled by default	Yes

Severity/Certainty

Low/Medium

Full description

(Required) The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type

Coding standards

MISRA C:2012 Rule-14.4

> (Required) The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type

Code examples

The following code example fails the check and will give a warning:

```
void example(void)
{
  int u8;
  if (u8) {}
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stddef.h>

int * fn()
{
  int * ptr;
  return ptr;
}

int fn2()
{
  return 5;
}

bool fn3()
{
  return true;
}

void example(void)
{
  while (int *ptr = fn() )  // Compliant by exception
  {}

  do
  {
    int *ptr = fn();
    if ( NULL == ptr )
    {
      break;
    }
  }
  while (true); // Compliant

  while (int len = fn2() )  // Compliant by exception
  {}

  if (int *p = fn()) {}   // Compliant by exception
  if (int len = fn2() ) {} // Complioant by exception
  if (bool flag = fn3()) {} // Compliant
}
```

MISRAC2012-Rule-14.4_d

Synopsis	Non-boolean termination conditions in while statements.
Enabled by default	Yes

Severity/Certainty

Low/Medium

Full description

(Required) The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type

Coding standards

MISRA C:2012 Rule-14.4

> (Required) The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type

Code examples

The following code example fails the check and will give a warning:

```
void example(void)
{
  int u8;
  while (u8) {}
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stddef.h>

int * fn()
{
  int * ptr;
  return ptr;
}

int fn2()
{
  return 5;
}

bool fn3()
{
  return true;
}

void example(void)
{
  while (int *ptr = fn() )  // Compliant by exception
  {}

  do
  {
    int *ptr = fn();
    if ( NULL == ptr )
    {
      break;
    }
  }
  while (true); // Compliant

  while (int len = fn2() )  // Compliant by exception
  {}

  if (int *p = fn()) {}   // Compliant by exception
  if (int len = fn2() ) {} // Complioant by exception
  if (bool flag = fn3()) {} // Compliant
}
```

MISRAC2012-Rule-15.1

Synopsis	Uses of goto.
Enabled by default	No

Severity/Certainty

Low/Medium

Full description

(Advisory) The goto statement should not be used

Coding standards

MISRA C:2012 Rule-15.1

(Advisory) The goto statement should not be used

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {

  goto testin;

testin:
  printf("Reached by goto");

}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {

  printf ("Not reached by goto");

}
```

MISRAC2012-Rule-15.2

Synopsis

Goto declared after target label.

Enabled by default

Yes

Severity/Certainty

Low/Low

Full description

(Required) The goto statement shall jump to a label declared later in the same function

Coding standards MISRA C:2012 Rule-15.2

(Required) The goto statement shall jump to a label declared later in the same function

Code examples The following code example fails the check and will give a warning:

```
void f1 ( )
{
  int j = 0;
  for ( j = 0; j < 10 ; ++j )
  {
L1: // Non-compliant
    j;
  }
  goto L1;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void f1 ( )
{
  int j = 0;
  goto L1;
  for ( j = 0; j < 10 ; ++j )
  {
    j;
  }
L1:
  return;
}
```

MISRAC2012-Rule-15.3

Synopsis The target of the goto is a nested code block.

Enabled by default Yes

Severity/Certainty Low/Low

Full description

(Required) Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement

Coding standards

MISRA C:2012 Rule-15.3

> (Required) Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement

Code examples

The following code example fails the check and will give a warning:

```
void f1 ( )
{
  int j = 0;
  goto L1;
  for (;;)
  {
L1: // Non-compliant
    j;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void f2()
{
  for(;;)
  {
    for(;;)
    {
      goto L1;
    }
  }
L1:
  return;
}
```

MISRAC2012-Rule-15.4

Synopsis

There should be no more than one break or goto statement used to terminate any iteration statement

Enabled by default

No

Severity/Certainty	Low/Medium
Full description	(Advisory) There should be no more than one break or goto statement used to terminate any iteration statement
Coding standards	MISRA C:2012 Rule-15.4 (Advisory) There should be no more than one break or goto statement used to terminate any iteration statement
Code examples	The following code example fails the check and will give a warning:

```
void func()
{
  int x = 1;
  for ( int i = 0; i < 10; i++ )
  {
    if ( x )
    {
      break;
    }
    else if ( i )
    {
      break;  // Non-compliant - second jump from loop
    }
    else
    {
      // Code
    }
  }
}
int fn(void);

void example(void) {
   int i = fn();
   int j;
   int counter = 0;
   switch (i) {
      case 1:
        break;
      case 2:
      case 3:
        counter++;
        if (i==3) {
              break;
        }
        counter++;
        break;
      case 4:
        for (j = 0; j < 10; j++) {
              if (j == i) {
                     break;
              }
              if (j == counter) {
                     break;
              }
        }
        counter--;
        break;
```

```
    default:
      break;
  }
}
int fn(int i);

void example(void) {
  int counter = 0;
  int i = 0;
  for (i = 0; i < 100; i++) {
    switch (i % 9) {
      case 8:
          counter++;
          break;
      default:
          break;
    }
    if (fn(i)) {
      break;
    }
    if (fn(i)) {
      break;
    }
  }
}

int test1(int);
int test2(int);

void example(void)
{
  int i = 0;
  for (i = 0; i < 10; i++) {
    if (test1(i)) {
      break;
    } else if (test2(i)) {
      break;
    }
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void)
{
  int i = 0;
  for (i = 0; i < 10 && i != 9; i++) {
    if (i == 9) {
      break;
    }
  }
}
void func()
{
  int x = 1;
  for ( int i = 0; i < 10; i++ )
  {
    if ( x )
    {
      break;
    }
    else if ( i )
    {
      while ( true )
      {
        if ( x )
        {
          break;
        }
        do
        {
          break;
        }
        while(true);
      }
    }
    else
    {
    }
  }
}
int fn(void);

void example(void) {
  int i = fn();
  int j;
  int counter = 0;
  switch (i) {
    case 1:
      break;
```

```
        case 2:
        case 3:
          counter++;
          if (i==3) {
              break;
          }
          counter++;
          break;
        case 4:
          for (j = 0; j < 10; j++) {
              if (j == i) {
                    break;
              }
          }
          counter--;
          break;
        default:
          break;
      }
    }
    int fn(int i);

    void example(void) {
      int counter = 0;
      int i = 0;
      int stop = 0;
      for (i = 0; i < 100 && !stop; i++) {
        switch (i % 9) {
          case 8:
              counter++;
              break;
          default:
              break;
        }
        stop = fn(i);
      }
    }
```

MISRAC2012-Rule-15.5

Synopsis	A function shall have a single point of exit at the end of the function.
Enabled by default	No

Severity/Certainty

Low/Medium

Full description

(Advisory) A function should have a single point of exit at the end function. This is required by IEC 61508, under good programming style.

Coding standards

MISRA C:2012 Rule-15.5

(Advisory) A function should have a single point of exit at the end

Code examples

The following code example fails the check and will give a warning:

```
extern int errno;

void example(void) {
  if (errno) {
    return;
  }
  return;
}
```

The following code example passes the check and will not give a warning about this issue:

```
extern int errno;

void example(void) {
  if (errno) {
    goto end;
  }
end:
  {
    return;
  }
}
```

MISRAC2012-Rule-15.6_a

Synopsis

Missing braces in do ... while statements

Enabled by default

Yes

Severity/Certainty

Low/Low

Full description

(Required) The body of an iteration-statement or a selection-statement shall be acompound-statement

Coding standards

CERT EXP19-C

Use braces for the body of an if, for, or while statement

CWE 483

Incorrect Block Delimitation

MISRA C:2012 Rule-15.6

(Required) The body of an iteration-statement or a selection-statement shall be acompound-statement

Code examples

The following code example fails the check and will give a warning:

```
int example(void) {
  do
    return 0;
  while (1);
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(void) {
  do {
    return 0;
  } while (1);
}
```

MISRAC2012-Rule-15.6_b

Synopsis

Missing braces in for statements

Enabled by default

Yes

Severity/Certainty

Low/Low

Full description

(Required) The body of an iteration-statement or a selection-statement shall be acompound-statement

Coding standards

CERT EXP19-C

Use braces for the body of an if, for, or while statement

CWE 483

Incorrect Block Delimitation

MISRA C:2012 Rule-15.6

(Required) The body of an iteration-statement or a selection-statement shall be acompound-statement

Code examples

The following code example fails the check and will give a warning:

```
int example(void) {
  for (;;)
    return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(void) {
  for (;;){
    return 0;
  }
}
```

MISRAC2012-Rule-15.6_c

Synopsis

Missing braces in if, else, and else if statements

Enabled by default

Yes

Severity/Certainty

Low/Low

Full description

(Required) The body of an iteration-statement or a selection-statement shall be acompound-statement

Coding standards

CERT EXP19-C

Use braces for the body of an if, for, or while statement

CWE 483

Incorrect Block Delimitation

MISRA C:2012 Rule-15.6

(Required) The body of an iteration-statement or a selection-statement shall be acompound-statement

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  if (random());
  if (random());
  else;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  if (random()) {
  }
  if (random()) {
  } else {
  }
  if (random()) {
  } else if (random()) {
  }
}
```

MISRAC2012-Rule-15.6_d

Synopsis

Missing braces in switch statements

Enabled by default

Yes

Severity/Certainty

Low/Low

Full description

(Required) The body of an iteration-statement or a selection-statement shall be acompound-statement

Coding standards

CERT EXP19-C

Use braces for the body of an if, for, or while statement

CWE 483

Incorrect Block Delimitation

MISRA C:2012 Rule-15.6

(Required) The body of an iteration-statement or a selection-statement shall be acompound-statement

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  while(1);
  for(;;);
  do ;
  while (0);
  switch(0);
}
```

The following code example passes the check and will not give a warning about this issue:

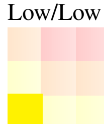
```
void example(void) {
  while(1) {
  }
  for(;;) {
  }
  do {
  } while (0);
  switch(0) {
  }
}
```

MISRAC2012-Rule-15.6_e

Synopsis: Missing braces in while statements

Enabled by default: Yes

Severity/Certainty: Low/Low



Full description: (Required) The body of an iteration-statement or a selection-statement shall be acompound-statement

Coding standards:

CERT EXP19-C

Use braces for the body of an if, for, or while statement

CWE 483

Incorrect Block Delimitation

MISRA C:2012 Rule-15.6

(Required) The body of an iteration-statement or a selection-statement shall be acompound-statement

Code examples:

The following code example fails the check and will give a warning:

```
int example(void) {
  while (1)
    return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(void) {
  while (1){
    return 0;
  }
}
```

MISRAC2012-Rule-15.7

Synopsis: If ... else if constructs that are not terminated with an else clause.

Enabled by default Yes

Severity/Certainty Low/High

Full description (Required) All if ... else if constructs shall be terminated with an else statement

Coding standards MISRA C:2012 Rule-15.7

(Required) All if ... else if constructs shall be terminated with an else statement

Code examples The following code example fails the check and will give a warning:

```
void example(void) {
  if (!rand()) {
    printf("The first random number is 0");
  } else if (!rand()) {
    printf("The second random number is 0");
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  if (!rand()) {
    printf("The first random number is 0");
  } else if (!rand()) {
    printf("The second random number is 0");
  } else {
    printf("Neither random number was 0");
  }
}
```

MISRAC2012-Rule-16.1

Synopsis Switch statements that do not conform to the MISRA C switch syntax.

Enabled by default Yes

Severity/Certainty

Low/High

Full description

(Required) All switch statements shall be well-formed switch-statement : switch '(' expression ')' '{' case-label-clause-list default-label-clause? '}' case-label-clause-list: case-label case-clause? case-label-clause-list case-label case-clause? case-label: case constant-expression ':' case-clause: statement-list? break ';' '{' declaration-list? statement-list? break ';' '}' default-label-clause : default-label default-clause default-label: default ':' default-clause: case-clause

Coding standards

MISRA C:2012 Rule-16.1

(Required) All switch statements shall be well-formed

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  switch(expr()) {
    // at least one case label
    case 1:
      // statement list
      stmt();
      stmt();
      // WARNING: missing break at end of statement list
    default:
      break; // statement list ends in a break
  }

  switch(expr()) {
    // WARNING: missing at least one case label
    default:
      break; // statement list ends in a break
  }

  switch(expr()) {
    // at least one case label
    case 1:
      // statement list
      stmt();
      stmt();
      break; // statement list ends in a break
    case 0:
      stmt();
      // WARNING: declaration list without block
      int decl = 0;
      int x;
      // statement list
      stmt();
      stmt();
      break; // statement list ends in a break
    default:
      break; // statement list ends in a break
  }

  switch(expr()) {
    // at least one case label
    case 1: {
      // statement list
      stmt();
      // WARNING: Additional block inside of the case clause
block
      {
      stmt();
```

```
      }
      break;
    }
    default:
      break; // statement list ends in a break
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  switch(expr()) {
    // at least one case label
    case 1:
      // statement list (no declarations)
      stmt();
      stmt();
      break; // statement list ends in a break
    case 0: {
      // one level of block is allowed
      // declaration list
      int decl = 0;
      // statement list
      stmt();
      stmt();
      break; // statement list ends in a break
    }
    case 2: // empty cases are allowed
    default:
      break; // statement list ends in a break
  }
}
```

MISRAC2012-Rule-16.2

Synopsis	Switch labels in nested blocks.
Enabled by default	Yes
Severity/Certainty	Low/Medium

Full description

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

Coding standards

MISRA C:2012 Rule-16.2

> (Required) A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {

  switch(rand()) {
    {case 1:}
    case 2:
    case 3:
    default:
  }

}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {

  switch(rand()) {
    case 1:
    case 2:
    case 3:
    default:
  }

}
```

MISRAC2012-Rule-16.3

Synopsis

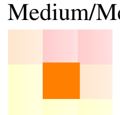
Non-empty switch cases not terminated by break

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) An unconditional break statement shall terminate every switch-clause

Coding standards

CERT MSC17-C

Finish every set of statements associated with a case label with a break statement

CWE 484

Omitted Break Statement in Switch

MISRA C:2012 Rule-16.3

(Required) An unconditional break statement shall terminate every switch-clause

Code examples

The following code example fails the check and will give a warning:

```
void example(int input) {

  while (rand()) {
    switch(input) {
      case 0:
        if (rand()) {
          break;
        }
      default:
        break;
    }
  }
}
void example(int input) {

  switch(input) {
    case 0:
      if (rand()) {
        break;
      }
    default:
      break;
  }

}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(int input) {

  switch(input) {
    case 0:
      if (rand()) {
        break;
      }
      break;
    default:
      break;
  }

}
void example(int input) {

  switch(input) {
    case 0:
      if (rand()) {
        break;
      } else {
        break;
      }
      // All paths above contain a break, therefore we do not
warn
    default:
      break;
  }

}
```

MISRAC2012-Rule-16.4

Synopsis	Switch statements with no default clause.
Enabled by default	Yes
Severity/Certainty	Low/Medium 
Full description	(Required) Every switch statement shall have a default label
Coding standards	CWE 478

Missing Default Case in Switch Statement

MISRA C:2012 Rule-16.4

(Required) Every switch statement shall have a default label

Code examples

The following code example fails the check and will give a warning:

```
int example(int x) {
  switch(x){
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(int x) {
  switch(x){
    case 3:
      return 0;
      break;
    case 5:
      return 1;
      break;
    default:
      return 2;
      break;
  }
}
```

MISRAC2012-Rule-16.5

Synopsis

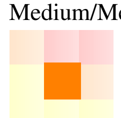
A switch's default label should be either the first or last label of the switch

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) A default label shall appear as either the first or the last switch label of a switch statement

Coding standards

MISRA C:2012 Rule-16.5

(Required) A default label shall appear as either the first or the last switch label of a switch statement

Code examples

The following code example fails the check and will give a warning:

```
void test(int a) {
  switch (a) {
    case 1:
      a = 1;
      break;
    default:
      a = 10;
      break;
    case 2:
      a = 2;
      break;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void test(int a) {
  switch (a) {
    case 1:
      a = 1;
      break;
    case 2:
      a = 2;
      break;
    default:
      a = 10;
      break;
  }
}
```

MISRAC2012-Rule-16.6

Synopsis

Switch statements with no cases.

Enabled by default

Yes

Severity/Certainty Low/Medium

Full description (Required) Every switch statement shall have at least two switch-clauses

Coding standards MISRA C:2012 Rule-16.6

(Required) Every switch statement shall have at least two switch-clauses

Code examples The following code example fails the check and will give a warning:

```
int example(int x) {
  switch(x){
    default:
      return 2;
      break;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(int x) {
  switch(x){
    case 3:
      return 0;
      break;
    case 5:
      return 1;
      break;
    default:
      return 2;
      break;
  }
}
```

MISRAC2012-Rule-16.7

Synopsis A switch expression shall not represent a value that is effectively boolean.

Enabled by default Yes

Severity/Certainty

Low/Medium

Full description

(Required) A switch-expression shall not have essentially Boolean type

Coding standards

MISRA C:2012 Rule-16.7

(Required) A switch-expression shall not have essentially Boolean type

Code examples

The following code example fails the check and will give a warning:

```
void example(int x) {
  switch(x == 0) {
    case 0:
    case 1:
    default:
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(int x) {
  switch(x) {
    case 1:
    case 0:
    default:
  }
}
```

MISRAC2012-Rule-17.1

Synopsis

The use of the stdarg header is not permitted

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The features of <stdarg.h> shall not be used

Coding standards MISRA C:2012 Rule-17.1

(Required) The features of <stdarg.h> shall not be used

Code examples The following code example fails the check and will give a warning:

```
#include <stdlib.h>
#include <stdarg.h>

void example(int a, ...) {
  va_list vl;
  va_list v2;
  int val;
  va_start(vl, a);
  va_copy(vl, v2);
  val=va_arg(vl, int);
  va_end(vl);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

int example(void) {
  return EXIT_SUCCESS;
}
```

MISRAC2012-Rule-17.2_a

Synopsis Functions that call themselves directly.

Enabled by default Yes

Severity/Certainty Low/Medium

Full description (Required) Functions shall not call themselves, either directly or indirectly

Coding standards MISRA C:2012 Rule-17.2

(Required) Functions shall not call themselves, either directly or indirectly

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  example();
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC2012-Rule-17.2_b

Synopsis

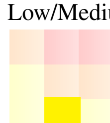
Functions that call themselves indirectly.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) Functions shall not call themselves, either directly or indirectly

Coding standards

MISRA C:2012 Rule-17.2

(Required) Functions shall not call themselves, either directly or indirectly

Code examples

The following code example fails the check and will give a warning:

```
void example(void);
void callee(void) {
    example();
}
void example(void) {
    callee();
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void);
void callee(void) {
    // example();
}
void example(void) {
    callee();
}
```

MISRAC2012-Rule-17.3

Synopsis

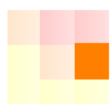
Functions used without prototyping

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

(Mandatory) A function shall not be declared implicitly

Coding standards

CERT DCL31-C

Declare identifiers before using them

MISRA C:2012 Rule-17.3

(Mandatory) A function shall not be declared implicitly

Code examples

The following code example fails the check and will give a warning:

```
void func2(void)
{
    func();
}
```

The following code example passes the check and will not give a warning about this issue:

```
void func(void);
void func2(void)
{
    func();
}
```

MISRAC2012-Rule-17.4

Synopsis

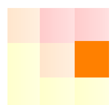
For some execution, no return statement is executed in a function with a non-void return type

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

(Mandatory) All exit paths from a function with non-void return type shall have an explicit return statement with an expression Checks whether all execution paths in non-void functions contain a return statement before they exit. If a non-void function has no return statement, it will return an undefined value. This will not pose a problem if the function is used as a void function, however, if the function return value is used it will cause unpredictable behavior.   Note: This is a weaker check than the one performed by gcc. Its check allows more aggressive coding without violating the rule. However, a rule violation in gcc means there is no path leading to a return statement. non-`void` return type.

Coding standards

CERT MSC37-C

Ensure that control never reaches the end of a non-void function

MISRA C:2012 Rule-17.4

(Mandatory) All exit paths from a function with non-void return type shall have an explicit return statement with an expression

Code examples

The following code example fails the check and will give a warning:

```
#include <stdio.h>

int example(void) {
  int x;

  scanf("%d",&x);

  if (x > 10) {
    return 10;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdio.h>

int example(void) {
  int x;

  scanf("%d",&x);

  if (x > 10) {
    return 10;
  }

  return 0;
}
```

MISRAC2012-Rule-17.6

Synopsis

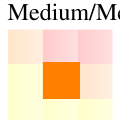
Array parameters shall not have the static keyword between the []

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Mandatory) The declaration of an array parameter shall not contain the static keyword between the [ ]

Coding standards

MISRA C:2012 Rule-17.6

> (Mandatory) The declaration of an array parameter shall not contain the static keyword between the [ ]

Code examples

The following code example fails the check and will give a warning:

```
void example(int a[static 20]) {
  for (int i = 0; i < 10; i++) {
    a[i] = i;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(int a[20]) {
  for (int i = 0; i < 10; i++) {
    a[i] = i;
  }
}
```

MISRAC2012-Rule-17.7

Synopsis

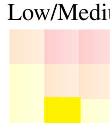
Unused function return values (excluding overloaded operators)

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) The value returned by a function having non-void return type shall be used return value of a function shall always be used. Overloaded operators are excluded, as they should behave in the same way as built-in operators. The return value of a function may be discarded by use of a (void) cast.

Coding standards

CWE 252

Unchecked Return Value

MISRA C:2012 Rule-17.7

(Required) The value returned by a function having non-void return type shall be used

Code examples

The following code example fails the check and will give a warning:

```
int func ( int para1 )
{
   return para1;
}

void discarded ( int para2 )
{
  func(para2);          // value discarded - Non-compliant
}
```

The following code example passes the check and will not give a warning about this issue:

```
int func ( int para1 )
{
   return para1;
}

int not_discarded ( int para2 )
{
  if (func(para2) > 5){
    return 1;
  }
  return 0;
}
```

MISRAC2012-Rule-18.1_a

Synopsis

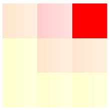
Array access is out of bounds.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

(Required) A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand This is likely to corrupt data and/or crash the program, and may result in security vulnerabilities.

Coding standards

CERT ARR33-C

Guarantee that copies are made into storage of sufficient size

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

CWE 121

Stack-based Buffer Overflow

CWE 124

Buffer Underwrite ('Buffer Underflow')

CWE 126

Buffer Over-read

CWE 127

Buffer Under-read

CWE 129

Improper Validation of Array Index

MISRA C:2012 Rule-18.1

(Required) A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand

Code examples

The following code example fails the check and will give a warning:

```
/* Goanna correctly detects that the array access,
   a[x - 10] is always within bounds, because 'x'
   is always in the range 10 <= x < 20, but a[x]
   is not. */

int ex(int x, int y)
{
  int a[10];

  if((x >= 0) && (x < 20)) {
    if(x < 10) {
      y = a[x];
    } else {
      y = a[x - 10];
      y = a[x];
    }
  }

  return y;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int main(void)
{
  int a[4];

  a[3] = 0;

  return 0;
}
```

MISRAC2012-Rule-18.1_b

Synopsis

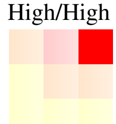
Array access may be out of bounds, depending on which path is executed.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

(Required) A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand This may corrupt data and/or crash the program, and may also result in security vulnerabilities.

Coding standards

CERT ARR33-C

Guarantee that copies are made into storage of sufficient size

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

CWE 121

Stack-based Buffer Overflow

CWE 124

Buffer Underwrite ('Buffer Underflow')

CWE 126

Buffer Over-read

CWE 127

Buffer Under-read

CWE 129

Improper Validation of Array Index

MISRA C:2012 Rule-18.1

(Required) A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand

Code examples

The following code example fails the check and will give a warning:

```
int cond;

int main(void)
{
  int a[7];
  int x;

  if (cond)
    x = 3;
  else
    x = 20;

  a[x] = 0;  //x may be set to 20 in line 11
             //but a only has an interval of [0,6]
  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int cond;

int main(void)
{
  int a[25];
  int x;

  if (cond)
    x = 3;
  else
    x = 20;

  a[x] = 0;  //here, both possible values of
             //x are in the interval [0,24]
  return 0;
}
```

MISRAC2012-Rule-18.1_c

Synopsis

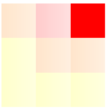
A pointer to an array is used outside the array bounds

Enabled by default

Yes

Severity/Certainty

High/High



Full description

(Required) A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand

Coding standards

CERT ARR33-C

Guarantee that copies are made into storage of sufficient size

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

CWE 121

Stack-based Buffer Overflow

CWE 122

Heap-based Buffer Overflow

CWE 124

Buffer Underwrite ('Buffer Underflow')

CWE 126

Buffer Over-read

CWE 127

Buffer Under-read

CWE 129

Improper Validation of Array Index

MISRA C:2012 Rule-18.1

(Required) A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int arr[10];
  int *p = arr;
  p[10];
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int arr[10];
  int *p = arr;
  p[9];
}
```

MISRAC2012-Rule-18.1_d

Synopsis

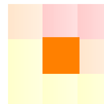
A pointer to an array is potentially used outside the array bounds

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand

Coding standards

CERT ARR33-C

Guarantee that copies are made into storage of sufficient size

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

CWE 121

Stack-based Buffer Overflow

CWE 122

Heap-based Buffer Overflow

CWE 124

Buffer Underwrite ('Buffer Underflow')

CWE 126

Buffer Over-read

CWE 127

Buffer Under-read

CWE 129

Improper Validation of Array Index

MISRA C:2012 Rule-18.1

(Required) A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand

Code examples

The following code example fails the check and will give a warning:

```
void example(int b) {
  int arr[10];
  int *p = arr;
  int x = (b<10 ? 8 : 11);
  p[x];
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(int b) {
  int arr[12];
  int *p = arr;
  int x = (b<10 ? 8 : 11);
  p[x];
}
```

MISRAC2012-Rule-18.5

Synopsis

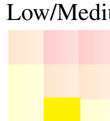
The declaration of objects should contain no more than two levels of pointer indirection.

Enabled by default

No

Severity/Certainty

Low/Medium



Full description

(Advisory) Declarations should contain no more than two levels of pointer nesting

Coding standards

MISRA C:2012 Rule-18.5

(Advisory) Declarations should contain no more than two levels of pointer nesting

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
    int ***p;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
    int **p;
}
```

MISRAC2012-Rule-18.6_a

Synopsis

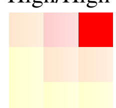
May return address on the stack.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

(Required) 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 Depending on the circumstances, this code and subsequent memory accesses could appear to work, but the operations are illegal and a program crash, or memory corruption, is very likely. Returning a copy of the object, using a global variable, or dynamically allocating memory, are possible alternatives.

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 562

Return of Stack Variable Address

MISRA C:2012 Rule-18.6

(Required) 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

Code examples

The following code example fails the check and will give a warning:

```
int *f() {
  int x;
  return &x;  //x is a local variable
}
int *example(void) {
  int a[20];
  return a;  //a is a local array
}
```

The following code example passes the check and will not give a warning about this issue:

```
int* example(void) {
  int *p,i;
  p = (int *)malloc(sizeof(int));
  return p;  //OK - p is dynamically allocated

}
```

MISRAC2012-Rule-18.6_b

Synopsis

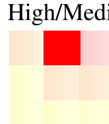
Store a stack address in a global pointer.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

(Required) 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 This is particularly dangerous because the program may appear to work normally, while it is in fact accessing illegal memory. Other results of this are a program crash, or the data changing unpredictably.

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 466

Return of Pointer Value Outside of Expected Range

MISRA C:2012 Rule-18.6

(Required) 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

Code examples

The following code example fails the check and will give a warning:

```
int *px;
void example() {
  int i = 0;
  px = &i; // assigning the address of stack
           // variable a to the global px
}
```

The following code example passes the check and will not give a warning about this issue:

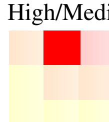
```
void example(int *pz) {
  int x; int *px = &x;
  int *py = px; /* local variable */
  pz = px; /* parameter */
}
```

MISRAC2012-Rule-18.6_c

Synopsis
Store a stack address in the field of a global struct.

Enabled by default
Yes

Severity/Certainty
High/Medium



Full description
(Required) 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 This is particularly dangerous because the program may appear to work normally, while it is in fact accessing illegal memory. Other results of this are a program crash, or the data changing unpredictably.

Coding standards
CERT DCL30-C

Declare objects with appropriate storage durations

CWE 466

Return of Pointer Value Outside of Expected Range

MISRA C:2012 Rule-18.6

(Required) 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

Code examples

The following code example fails the check and will give a warning:

```
struct S{
  int *px;
} s;

void example() {
  int i = 0;
  s.px = &i; //storing local address in global struct
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

struct S{
  int *px;
} s;

void example() {
  int i = 0;
  s.px = &i; //OK - the field is written to later
  s.px = NULL;
}
```

MISRAC2012-Rule-18.6_d

Synopsis

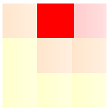
Store stack address outside function via parameter.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

(Required) 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 This is particularly dangerous because the program may appear to work normally, while it is in fact accessing illegal memory. Other results of this are a program crash, or the data changing unpredictably. Known false positives: this test checks for any expression refering to the store located by the parameter and so the assignment 'local[*parameter] = & local;' will invoke a warning.

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 466

Return of Pointer Value Outside of Expected Range

MISRA C:2012 Rule-18.6

(Required) 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

Code examples

The following code example fails the check and will give a warning:

```
void example(int **ppx) {
  int x;
  ppx[0] = &x;  //local address
}
```

The following code example passes the check and will not give a warning about this issue:

```
static int y = 0;
void example3(int **ppx){
  *ppx = &y;  //OK - static address
}
```

MISRAC2012-Rule-18.7

Synopsis

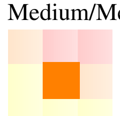
Flexible array members shall not be declared

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Flexible array members shall not be declared

Coding standards

MISRA C:2012 Rule-18.7

(Required) Flexible array members shall not be declared

Code examples

The following code example fails the check and will give a warning:

```
struct example {
  int size;
  int data[];
} example;

void function(void) {
  struct example *e;
}
```

The following code example passes the check and will not give a warning about this issue:

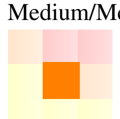
```
struct example {
  int size;
  int data[5];
} example;

void function(void) {
  struct example *e;
}
```

MISRAC2012-Rule-18.8

Synopsis
Arrays shall not be declared with a variable length

Enabled by default
Yes

Severity/Certainty
Medium/Medium



Full description
(Required) Variable-length array types shall not be used

Coding standards
MISRA C:2012 Rule-18.8

(Required) Variable-length array types shall not be used

Code examples
The following code example fails the check and will give a warning:

```
void example(int a) {
  int arr[a];
}
```

The following code example passes the check and will not give a warning about this issue:

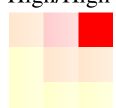
```
void example(int a) {
  int arr[10];
}
```

MISRAC2012-Rule-19.1

Synopsis Assignments from one field of a union to another.

Enabled by default Yes

Severity/Certainty High/High



Full description (Mandatory) An object shall not be assigned or copied to an overlapping object

Coding standards MISRA C:2012 Rule-19.1

(Mandatory) An object shall not be assigned or copied to an overlapping object

Code examples The following code example fails the check and will give a warning:

```
union cheat {
    char c[5];
    int i;
};

void example(union cheat *u)
{
    u->i = u->c[2];
}
union {
    char c[5];
    int i;
} u;

void example(void)
{
    u.i = u.c[2];
}
void example(void)
{
  union
  {
    char c[5];
    int i;
  } u;
  u.i = u.c[2];
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void)
{
  union
  {
    char c[5];
    int i;
  } u;
  int x;
  x = (int)u.c[2];
  u.i = x;
}
void example(void)
{
  struct
  {
    char c[5];
    int i;
  } u;
  u.i = u.c[2];
}
union cheat {
  char c[5];
  int i;
};

union cheat u;

void example(void)
{
  int x;
  x = (int)u.c[2];
  u.i = x;
}
```

MISRAC2012-Rule-19.2

Synopsis	All unions
Enabled by default	No
Severity/Certainty	Low/Medium

Full description

(Advisory) The union keyword should not be used

Coding standards

MISRA C:2012 Rule-19.2

(Advisory) The union keyword should not be used

Code examples

The following code example fails the check and will give a warning:

```
union cheat {
  int   i;
  float f;
};

int example(float f) {
  union cheat u;
  u.f = f;
  return u.i;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(int x) {
  return x;
}
```

MISRAC2012-Rule-20.2

Synopsis

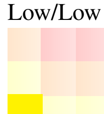
Illegal characters in header file names

Enabled by default

Yes

Severity/Certainty

Low/Low



Full description

(Required) The ',' or characters and the /* or // character sequences shall not occur in a header file name ', \, /*, or // characters are used between the " delimiters in a header name preprocessing token.

Coding standards

MISRA C:2012 Rule-20.2

(Required) The ',' or \ characters and the /* or // character sequences shall not occur in a header file name

Code examples

The following code example fails the check and will give a warning:

```
#include "fi'le.h"/* Non-compliant */
void example(void) {}
```

The following code example passes the check and will not give a warning about this issue:

```
#include "header.h"
void example(void) {}
```

MISRAC2012-Rule-20.4_c89

Synopsis

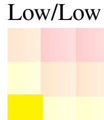
A macro shall not be defined with the same name as a keyword.

Enabled by default

Yes

Severity/Certainty

Low/Low



Full description

(Required) A macro shall not be defined with the same name as a keyword

Coding standards

MISRA C:2012 Rule-20.4

(Required) A macro shall not be defined with the same name as a keyword

Code examples

The following code example fails the check and will give a warning:

```
#define int some_other_type
```

The following code example passes the check and will not give a warning about this issue:

```
#define unless( E ) if ( ! ( E ) ) /* Compliant */
```

MISRAC2012-Rule-20.4_c99

Synopsis

A macro shall not be defined with the same name as a keyword.

Enabled by default

Yes

Severity/Certainty

Low/Low

Full description

(Required) A macro shall not be defined with the same name as a keyword

Coding standards

MISRA C:2012 Rule-20.4

(Required) A macro shall not be defined with the same name as a keyword

Code examples

The following code example fails the check and will give a warning:

```
/* The following example is compliant in C90, but not C99,
because inline is not a keyword in C90. */

/* Remove inline if compiling for C90 */
#define inline
```

The following code example passes the check and will not give a warning about this issue:

```
#define unless( E ) if ( ! ( E ) ) /* Compliant */
```

MISRAC2012-Rule-20.5

Synopsis

All #undef's

Enabled by default

No

Severity/Certainty

Low/Low

Full description

(Advisory) #undef should not be used or meaning of a macro when it is used in the code.

Coding standards

MISRA C:2012 Rule-20.5

(Advisory) #undef should not be used

Code examples

The following code example fails the check and will give a warning:

```
#defineSYM
#undef SYM
void example(void) {}
```

The following code example passes the check and will not give a warning about this issue:

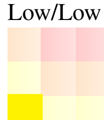
```
void example(void) {}
```

MISRAC2012-Rule-20.10

Synopsis — # or ## operator used in a macro definition

Enabled by default — No

Severity/Certainty — Low/Low



Full description — (Advisory) The # and ## preprocessor operators should not be used

Coding standards — MISRA C:2012 Rule-20.10

(Advisory) The # and ## preprocessor operators should not be used

Code examples — The following code example fails the check and will give a warning:

```
#defineA(X,Y)X##Y/* Non-compliant */

#define A(Y)#Y/* Non-compliant */
```

The following code example passes the check and will not give a warning about this issue:

```
#define A(x)(x)/* Compliant */
```

MISRAC2012-Rule-21.1

Synopsis — #define or #undef of a reserved identifier in the standard library

Enabled by default — Yes

Severity/Certainty

Low/Low

Full description

(Required) #define and #undef shall not be used on a reserved identifier or reserved macro name practice to #define a macro name that is a C/C++ reserved identifier, or C/C++ keyword or the name of any macro, object or function in the standard library. For example, there are some specific reserved words and function names that are known to give rise to undefined behaviour if they are redefined or undefined, including defined, __LINE__, __FILE__, __DATE__, __TIME__, __STDC__, errno and assert.

Coding standards

MISRA C:2012 Rule-21.1

(Required) #define and #undef shall not be used on a reserved identifier or reserved macro name

Code examples

The following code example fails the check and will give a warning:

```
#define__TIME__11111111/* Non-compliant */
```

The following code example passes the check and will not give a warning about this issue:

```
#define A(x)(x)/* Compliant */
```

MISRAC2012-Rule-21.2

Synopsis

A library function is being overridden.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) A reserved identifier or macro name shall not be declared

Coding standards

MISRA C:2012 Rule-21.2

(Required) A reserved identifier or macro name shall not be declared

Code examples

The following code example fails the check and will give a warning:

```
extern "C" void strcpy(void);
void strcpy(void) {}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {}
```

MISRAC2012-Rule-21.3

Synopsis

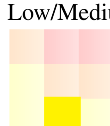
All uses of malloc, calloc, realloc, and free

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) The memory allocation and deallocation functions of <stdlib.h> shall not be used

Coding standards

MISRA C:2012 Rule-21.3

> (Required) The memory allocation and deallocation functions of <stdlib.h> shall not be used

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

void *example(void) {
  return malloc(100);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC2012-Rule-21.4

Synopsis

All uses of <setjmp.h>

Enabled by default Yes

Severity/Certainty Low/Medium

Full description (Required) The standard header file <setjmp.h> shall not be used

Coding standards CERT ERR34-CPP

Do not use longjmp

MISRA C:2012 Rule-21.4

(Required) The standard header file <setjmp.h> shall not be used

Code examples The following code example fails the check and will give a warning:

```
#include <setjmp.h>

jmp_buf ex;

void example(void) {
  setjmp(ex);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC2012-Rule-21.5

Synopsis All uses of <signal.h>

Enabled by default Yes

Severity/Certainty Low/Medium

Full description (Required) The standard header file <signal.h> shall not be used

Coding standards

MISRA C:2012 Rule-21.5

(Required) The standard header file <signal.h> shall not be used

Code examples

The following code example fails the check and will give a warning:

```
#include <signal.h>
#include <stddef.h>

void example(void) {
  signal(SIGFPE, NULL);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC2012-Rule-21.6

Synopsis

All uses of <stdio.h>

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The Standard Library input/output functions shall not be used

Coding standards

MISRA C:2012 Rule-21.6

(Required) The Standard Library input/output functions shall not be used

Code examples

The following code example fails the check and will give a warning:

```
#include <stdio.h>

void example(void) {
  printf("Hello, world!\n");
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC2012-Rule-21.7

Synopsis: All uses of atof, atoi, atol and atoll

Enabled by default: Yes

Severity/Certainty: Low/Medium

Full description: (Required) The atof, atoi, atol and atoll functions of <stdlib.h> shall not be used

Coding standards: CERT INT06-C

Use strtol() or a related function to convert a string token to an integer

MISRA C:2012 Rule-21.7

(Required) The atof, atoi, atol and atoll functions of <stdlib.h> shall not be used

Code examples: The following code example fails the check and will give a warning:

```
#include <stdlib.h>

int example(char buf[]) {
  return atoi(buf);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC2012-Rule-21.8

Synopsis: All uses of abort, exit, getenv, and system

Enabled by default: Yes

Severity/Certainty

Low/Medium

Full description

(Required) The library functions abort, exit, getenv and system of <stdlib.h> shall not be used

Coding standards

MISRA C:2012 Rule-21.8

(Required) The library functions abort, exit, getenv and system of <stdlib.h> shall not be used

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

void example(void) {
  abort();
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC2012-Rule-21.9

Synopsis

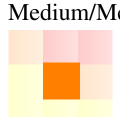
(Required) The library functions bsearch and qsort of <stdlib.h> shall not be used.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The library functions bsearch and qsort of <stdlib.h> shall not be used

Coding standards

MISRA C:2012 Rule-21.9

(Required) The library functions bsearch and qsort of <stdlib.h> shall not be used

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

int values[] = { 40, 10, 100, 90, 20, 25 };

int compare (const void * a, const void * b)
{
  return ( *(int*)a - *(int*)b );
}

int main ()
{
  qsort (values, 6, sizeof(int), compare);
  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

int values[] = { 40, 10, 100, 90, 20, 25 };

int compare (const void * a, const void * b)
{
  return ( *(int*)a - *(int*)b );
}

int main ()
{
  return 0;
}
```

MISRAC2012-Rule-21.10

Synopsis

All uses of <time.h> functions: asctime, clock, ctime, difftime, gmtime, localtime, mktime, strftime, and time

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The Standard Library time and date functions shall not be used

Coding standards

MISRA C:2012 Rule-21.10

(Required) The Standard Library time and date functions shall not be used

Code examples

The following code example fails the check and will give a warning:

```
#include <stddef.h>
#include <time.h>

time_t example(void) {
  return time(NULL);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC2012-Rule-21.11

Synopsis

The use of the tgmath header is not permitted

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The standard header file <tgmath.h> shall not be used

Coding standards

MISRA C:2012 Rule-21.11

(Required) The standard header file <tgmath.h> shall not be used

Code examples

The following code example fails the check and will give a warning:

```
#include <tgmath.h>

float f1, f2;

void example(void) {
  f1 = sqrt(f2);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <math.h>

float f1, f2;

void example(void) {
  f1 = sqrt(f2);
}
```

MISRAC2012-Rule-22.1_a

Synopsis

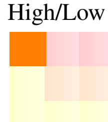
A memory leak due to improper deallocation.

Enabled by default

Yes

Severity/Certainty

High/Low



Full description

(Required) All resources obtained dynamically by means of Standard Library functions shall be explicitly released There must be no possible execution path during which the value is not freed, returned, or passed into another function as an argument, before it is lost. This is a memory leak.

Coding standards

CERT MEM31-C

Free dynamically allocated memory exactly once

CWE 401

Improper Release of Memory Before Removing Last Reference ('Memory Leak')

CWE 772

Missing Release of Resource after Effective Lifetime

MISRA C:2012 Rule-22.1

(Required) All resources obtained dynamically by means of Standard Library functions shall be explicitly released

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

extern int rand();

void example(void) {
  int *ptr = malloc(sizeof(int));

  if (rand()){

    //losing reference to memory allocated
    //from the first malloc
    ptr = malloc(sizeof(int));
  }

  free(ptr);
}
#include <stdlib.h>

int main(void) {
    int *ptr = (int*)malloc(sizeof (int));
    if (rand() < 5) {
        free(ptr); // Not free() on all paths.
    }
    return 0;
}
#include <stdlib.h>

int main(void) {
  int *ptr = (int *)malloc(sizeof(int));

  ptr = NULL; //losing reference to the allocated memory

  free(ptr);

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

int main(void) {
    int *ptr = (int*)malloc(sizeof(int));
    if (rand() < 5) {
        free(ptr);
    } else {
        free(ptr);
    }
    return 0;
}
#include <stdlib.h>

extern int rand();

void example(void) {
  int *ptr = malloc(sizeof(int));
  free(ptr);
}
```

MISRAC2012-Rule-22.1_b

Synopsis

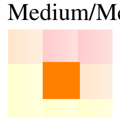
All file pointers obtained dynamically by means of Standard Library functions shall be explicitly released

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) All resources obtained dynamically by means of Standard Library functions shall be explicitly released the resources. Releasing file pointers as soon as possible reduces the possibility that exhaustion will occur.

Coding standards

CWE 404

Improper Resource Shutdown or Release

MISRA C:2012 Rule-22.1

(Required) All resources obtained dynamically by means of Standard Library functions shall be explicitly released

Code examples

The following code example fails the check and will give a warning:

```
#include <stdio.h>

void example(void) {
  FILE *fp = fopen("test.txt", "c");
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdio.h>

void example(void) {
  FILE *fp = fopen("test.txt", "c");
  fclose(fp);
}
#include <stdio.h>

void iCloseFilePointers(FILE *fp) {
  fclose(fp);
}

void example(void) {
  FILE *fp = fopen("text.txt", "w");
  iCloseFilePointers(fp);
}
```

MISRAC2012-Rule-22.2_a

Synopsis

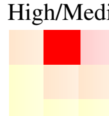
Freeing a memory location more than once.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

(Mandatory) A block of memory shall only be freed if it was allocated by means of a Standard Library function

Coding standards

CERT MEM31-C

Free dynamically allocated memory exactly once

CWE 415

Double Free

MISRA C:2012 Rule-22.2

(Mandatory) A block of memory shall only be freed if it was allocated by means of a Standard Library function

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>
void f(int *p) {
  free(p);
  if(p) free(p);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

void example(void)
{
  int *p=malloc(4);
  free(p);
}
```

MISRAC2012-Rule-22.2_b

Synopsis

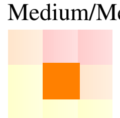
Freeing a memory location more than once on some paths but not others.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Mandatory) A block of memory shall only be freed if it was allocated by means of a Standard Library function

Coding standards

CERT MEM31-C

Free dynamically allocated memory exactly once

CWE 415

Double Free

MISRA C:2012 Rule-22.2

(Mandatory) A block of memory shall only be freed if it was allocated by means of a Standard Library function

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>
void example(void) {
    int *ptr = (int*)malloc(sizeof(int));
    free(ptr);
    if(rand() % 2 == 0)
    {
      free(ptr);
    }
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>
void example(void) {
    int *ptr = (int*)malloc(sizeof(int));
    if(rand() % 2 == 0)
    {
      free(ptr);
    }
    else
    {
      free(ptr);
    }
}
```

MISRAC2012-Rule-22.2_c

Synopsis

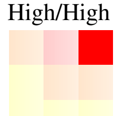
A stack address is possibly freed.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

(Mandatory) A block of memory shall only be freed if it was allocated by means of a Standard Library function Additionally, erroneously using `free()` on stack memory may corrupt `stdlib`'s memory bookkeeping, affecting heap memory.

Coding standards

CERT MEM34-C

Only free memory allocated dynamically

CWE 590

Free of Memory not on the Heap

MISRA C:2012 Rule-22.2

(Mandatory) A block of memory shall only be freed if it was allocated by means of a Standard Library function

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>
void example(void){
  int x=0;
  free(&x);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int *p;
  p = (int *)malloc(sizeof( int));
  free(p);
}
```

MISRAC2012-Rule-22.4

Synopsis

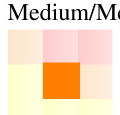
A file opened as read-only is written to

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description (Mandatory) There shall be no attempt to write to a stream which has been opened as read-only

Coding standards MISRA C:2012 Rule-22.4

> (Mandatory) There shall be no attempt to write to a stream which has been opened as read-only

Code examples The following code example fails the check and will give a warning:

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

void example(void) {
  FILE *f1;
  f1 = fopen("test-file.txt", "r");
  fprintf(f1, "Hello, World!");
  fclose(f1);
}
```

The following code example passes the check and will not give a warning about this issue:

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

void example(void) {
  FILE *f1;
  f1 = fopen("test-file.txt", "r+");
  fprintf(f1, "Hello, World!");
  fclose(f1);
}
```

MISRAC2012-Rule-22.5_a

Synopsis A pointer to a FILE object shall not be dereferenced

Enabled by default Yes

Severity/Certainty Low/Medium

Full description (Mandatory) A pointer to a FILE object shall not be dereferenced

Coding standards

MISRA C:2012 Rule-22.5

(Mandatory) A pointer to a FILE object shall not be dereferenced

Code examples

The following code example fails the check and will give a warning:

```
#include <stdio.h>

void example(void) {
   FILE *pf1;
   FILE f3;

   f3 = *pf1;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdio.h>

void example(void) {
  FILE *f1;
  FILE *f2;

  f1 = f2;
}
```

MISRAC2012-Rule-22.5_b

Synopsis

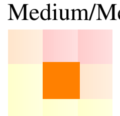
A file pointer is implicitly derefrenced by a library function

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Mandatory) A pointer to a FILE object shall not be dereferenced

Coding standards

MISRA C:2012 Rule-22.5

(Mandatory) A pointer to a FILE object shall not be dereferenced

Code examples

The following code example fails the check and will give a warning:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void example(void) {
  FILE *ptr1 = fopen("hello", "r");
  int *a;
  memcpy(ptr1, a, 10);
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void example(void) {
  FILE *ptr1;
  int *a;
  memcpy(a, a, 0);
}
```

MISRAC2012-Rule-22.6

Synopsis

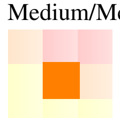
A file pointer is used after it has been closed.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Mandatory) The value of a pointer to a FILE shall not be used after the associated stream has been closed

Coding standards

MISRA C:2012 Rule-22.6

(Mandatory) The value of a pointer to a FILE shall not be used after the associated stream has been closed

Code examples

The following code example fails the check and will give a warning:

```
#include <stdio.h>

void example(void) {
  FILE *f1;
  f1 = fopen("test_file", "w");
  fclose(f1);
  fprintf(f1, "Hello, World!\n");
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdio.h>

void example(void) {
  FILE *f1;
  f1 = fopen("test_file", "w");
  fprintf(f1, "Hello, World!\n");
  fclose(f1);
}
```

MISRAC++2008-0-1-1

Synopsis

In all executions, a part of the program is not executed.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) A project shall not contain unreachable code. Though not necessarily a problem, dead code can indicate programmer confusion about the program's branching structure.

Coding standards

CERT MSC07-C

Detect and remove dead code

CWE 561

Dead Code

MISRA C++ 2008 0-1-1

(Required) A project shall not contain unreachable code.

Code examples

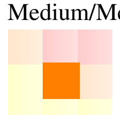
The following code example fails the check and will give a warning:

```
#include <stdio.h>

int f(int mode) {
    switch (mode) {
        case 0:
            return 1;
            printf("Hello!"); // This line cannot execute.
        default:
            return -1;
    }
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdio.h>

int f(int mode) {
    switch (mode) {
        case 0:
            printf("Hello!"); // This line can execute.
            return 1;
        default:
            return -1;
    }
}
```

MISRAC++2008-0-1-2_a

Synopsis	The condition in if, for, while, do-while and ternary operator will always be met.
Enabled by default	Yes
Severity/Certainty	Medium/Medium 
Full description	(Required) A project shall not contain infeasible paths.
Coding standards	CERT EXP17-C

Do not perform bitwise operations in conditional expressions

MISRA C++ 2008 0-1-2

(Required) A project shall not contain infeasible paths.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {

    int x = 5;

    for (x = 0; x < 6 && 1; x--) {
    }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {

    int x = 5;

    for (x = 0; x < 6 && 1; x++) {
    }
}
```

MISRAC++2008-0-1-2_b

Synopsis

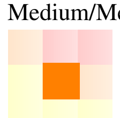
The condition in if, for, while, do-while and ternary operator will never be met.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) A project shall not contain infeasible paths.

Coding standards

CERT EXP17-C

Do not perform bitwise operations in conditional expressions

MISRA C++ 2008 0-1-2

(Required) A project shall not contain infeasible paths.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {

    int x = 5;

    for (x = 0; x < 6 && x >= 1; x++) {
    }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {

    int x = 5;

    for (x = 0; x < 6 && x >= 0; x++) {
    }
}
```

MISRAC++2008-0-1-2_c

Synopsis

A case statement within a switch statement is unreachable.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) A project shall not contain infeasible paths. the switch's expression cannot have the value of the case's label. This often occurs because of literal values having been assigned to the switch condition. An unreachable case is not inherently dangerous, but may represent a misunderstanding of program behavior on the programmer's part.

Coding standards

CERT MSC07-C

Detect and remove dead code

MISRA C++ 2008 0-1-2

(Required) A project shall not contain infeasible paths.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int x = 42;

  switch(2 * x) {
  case 42 :  //unreachable case, as x is 84
    ;
  default :
    ;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int x = 42;

  switch(2 * x) {
  case 84 :
    ;
  default :
    ;
  }
}
```

MISRAC++2008-0-1-3

Synopsis

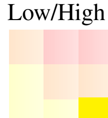
A variable is neither read nor written for any execution.

Enabled by default

Yes

Severity/Certainty

Low/High



Full description

(Required) A project shall not contain unused variables. includes initialization, and reading includes passing the variable as a parameter in a function call. This is not inherently dangerous, but may indicate an oversight or lack of understanding of the program behavior.

Coding standards

CERT MSC13-C

Detect and remove unused values

CWE 563

Unused Variable

MISRA C++ 2008 0-1-3

(Required) A project shall not contain unused variables.

Code examples

The following code example fails the check and will give a warning:

```
int example(void) {
  int x;  //this value is not used

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(void) {
  int x = 0;  //OK - x is returned

  return x;
}
```

MISRAC++2008-0-1-4

Synopsis

A variable is assigned a value that is never used.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) A project shall not contain non-volatile POD variables having only one use. destroys that value before it is used. This check does not detect situations where the value is simply lost when the function ends. This is not inherently dangerous, but may indicate an oversight or lack of understanding of the program behavior.

Coding standards

MISRA C++ 2008 0-1-4

(Required) A project shall not contain non-volatile POD variables having only one use.

Code examples

The following code example fails the check and will give a warning:

```
int example(void) {
  int x;

  x = 20;

  x = 3;
  return 0;
}
#include <stdlib.h>

void ex(void) {
  int *p = 0;
  int *q = 0;
  p = (int *)malloc(sizeof(int));
  q = (int *)malloc(sizeof(int));
  p = q;  //p is not used after this assignment
  return;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

int *ex(void) {
  int *p;
  p = (int *)malloc(sizeof(int));
  return p; //the value is returned
}
int example(void) {
  int x;

  x = 20;

  return x;
}
```

MISRAC++2008-0-1-6

Synopsis

A variable is assigned a value that is never used.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) A project shall not contain instances of non-volatile variables being given values that are never subsequently used. destroys that value before it is used. This check does not detect situations where the value is simply lost when the function ends. This is not inherently dangerous, but may indicate an oversight or lack of understanding of the program behavior.

Coding standards

MISRA C++ 2008 0-1-6

> (Required) A project shall not contain instances of non-volatile variables being given values that are never subsequently used.

Code examples

The following code example fails the check and will give a warning:

```
int example(void) {
  int x;

  x = 20;

  x = 3;
  return 0;
}
#include <stdlib.h>

void ex(void) {
  int *p = 0;
  int *q = 0;
  p = (int *)malloc(sizeof(int));
  q = (int *)malloc(sizeof(int));
  p = q;  //p is not used after this assignment
  return;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

int *ex(void) {
  int *p;
  p = (int *)malloc(sizeof(int));
  return p; //the value is returned
}
int example(void) {
  int x;

  x = 20;

  return x;
}
```

MISRAC++2008-0-1-7

Synopsis

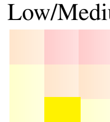
Unused function return values (excluding overloaded operators)

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) The value returned by a function having a non-void return type that is not an overloaded operator shall always be used. return value of a function shall always be used. Overloaded operators are excluded, as they should behave in the same way as built-in operators. The return value of a function may be discarded by use of a (void) cast.

Coding standards

CWE 252

Unchecked Return Value

MISRA C++ 2008 0-1-7

(Required) The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.

Code examples

The following code example fails the check and will give a warning:

```
int func ( int para1 )
{
   return para1;
}

void discarded ( int para2 )
{
  func(para2);          // value discarded - Non-compliant
}
```

The following code example passes the check and will not give a warning about this issue:

```
int func ( int para1 )
{
   return para1;
}

int not_discarded ( int para2 )
{
  if (func(para2) > 5){
    return 1;
  }
  return 0;
}
```

MISRAC++2008-0-1-8

Synopsis

A function with no return type and no side effects effectively does nothing.

Enabled by default

No

Severity/Certainty

Low/Low

Full description

(Required) All functions with void return type shall have external side effect(s).

Coding standards

MISRA C++ 2008 0-1-8

(Required) All functions with void return type shall have external side effect(s).

Code examples

The following code example fails the check and will give a warning:

```
void pointless (int i, char c)
{
  int local;
  local = 0;
  local = i;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void func(int i)
{
  int p;
  p = i;
  int *ptr;
  ptr = &i;
  i = p;
  i++;
}
```

MISRAC++2008-0-1-9

Synopsis In all executions, a part of the program is not executed.

Enabled by default Yes

Severity/Certainty Low/Medium

Full description (Required) There shall be no dead code. Though not necessarily a problem, dead code can indicate programmer confusion about the program's branching structure.

Coding standards CERT MSC07-C

Detect and remove dead code

CWE 561

Dead Code

MISRA C++ 2008 0-1-9

(Required) There shall be no dead code.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdio.h>

int f(int mode) {
    switch (mode) {
        case 0:
            return 1;
            printf("Hello!"); // This line cannot execute.
        default:
            return -1;
    }
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdio.h>

int f(int mode) {
    switch (mode) {
        case 0:
            printf("Hello!"); // This line can execute.
            return 1;
        default:
            return -1;
    }
}
```

MISRAC++2008-0-1-11

Synopsis

A function parameter is declared but not used.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) There shall be no unused parameters (named or unnamed) in nonvirtual functions. For example, the function may need to observe some calling protocol, or in C++ it may be a virtual function which doesn't need as much information from its arguments as related classes' equivalent functions do. Often, though, the warning indicates a genuine error.

Coding standards

CWE 563

Unused Variable

MISRA C++ 2008 0-1-11

(Required) There shall be no unused parameters (named or unnamed) in nonvirtual functions.

Code examples

The following code example fails the check and will give a warning:

```
int example(int x) {
  /* `x' is not used */
  return 20;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(int x) {
  return x + 20;
}
```

MISRAC++2008-0-2-1

Synopsis

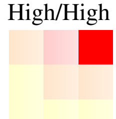
Assignments from one field of a union to another.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

(Required) An object shall not be assigned to an overlapping object.

Coding standards

MISRA C++ 2008 0-2-1

(Required) An object shall not be assigned to an overlapping object.

Code examples

The following code example fails the check and will give a warning:

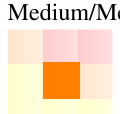
```
union cheat {
    char c[5];
    int i;
};

void example(union cheat *u)
{
    u->i = u->c[2];
}
union {
    char c[5];
    int i;
} u;

void example(void)
{
    u.i = u.c[2];
}
void example(void)
{
  union
  {
    char c[5];
    int i;
  } u;
  u.i = u.c[2];
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void)
{
  union
  {
    char c[5];
    int i;
  } u;
  int x;
  x = (int)u.c[2];
  u.i = x;
}
void example(void)
{
  struct
  {
    char c[5];
    int i;
  } u;
  u.i = u.c[2];
}
union cheat {
  char c[5];
  int i;
};

union cheat u;

void example(void)
{
  int x;
  x = (int)u.c[2];
  u.i = x;
}
```

MISRAC++2008-0-3-2

Synopsis	The return value for a library function that may return an error value is not used.
Enabled by default	Yes
Severity/Certainty	Medium/Medium



Full description

(Required) If a function generates error information, then that error information shall be tested.

Coding standards

CWE 252

Unchecked Return Value

CWE 394

Unexpected Status Code or Return Value

MISRA C++ 2008 0-3-2

(Required) If a function generates error information, then that error information shall be tested.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

void example(void) {
  malloc(sizeof(int));  // This function could fail,
                        // and the return value is
                        // not checked
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

void example(void) {
  int *x = (int *)malloc(sizeof(int));  // OK - return value
                                        // is stored
}
```

MISRAC++2008-2-3-1

Synopsis

Uses of trigraphs (in string literals only)

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description
: (Required) Trigraphs shall not be used.

Coding standards
: MISRA C++ 2008 2-3-1

  (Required) Trigraphs shall not be used.

Code examples
: The following code example fails the check and will give a warning:

```
void func()
{
  char * str = "abc??!def";
}
```

The following code example passes the check and will not give a warning about this issue:

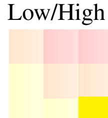
```
void func()
{
  char * str = "abc??def";
}
```

MISRAC++2008-2-7-1

Synopsis
: Appearances of /* inside comments

Enabled by default
: Yes

Severity/Certainty
: Low/High



Full description
: (Required) The character sequence /* shall not be used within a C-style comment. Consider: /* A comment, end comment marker accidentally omitted <<New Page>> initialise(X); /* this comment is not compliant */ In this case, X will not be initialised because the code is hidden in a comment.

Coding standards
: MISRA C++ 2008 2-7-1

  (Required) The character sequence /* shall not be used within a C-style comment.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  /* This comment starts here
  /* Nested comment starts here
  */
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  /* This comment starts here */
  /* Nested comment starts here
  */
}
```

MISRAC++2008-2-7-2

Synopsis

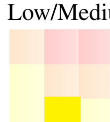
To allow comments to contain pseudo-code or code samples, only comments that end in ';', '{', or '}' characters are considered to be commented-out code.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) Sections of code shall not be "commented out" using C-style comments. Code sections in comments are identified where the comment ends in ';', '{', or '}' characters.

Coding standards

MISRA C++ 2008 2-7-2

> (Required) Sections of code shall not be "commented out" using C-style comments.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  /*
  int i;
  */
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
#if 0
  int i;
#endif
}
```

MISRAC++2008-2-7-3

Synopsis

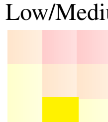
To allow comments to contain pseudo-code or code samples, only comments that end in ';', '{', or '}' characters are considered to be commented-out code.

Enabled by default

No

Severity/Certainty

Low/Medium



Full description

(Advisory) Sections of code should not be "commented out" using C++ comments. Code sections in comments are identified where the comment ends in ';', '{', or '}' characters.

Coding standards

MISRA C++ 2008 2-7-3

> (Advisory) Sections of code should not be "commented out" using C++ comments.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  //int i;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
#if 0
  int i;
#endif
}
```

MISRAC++2008-2-10-2_a

Synopsis

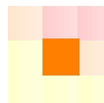
The definition of a local variable hides a global definition.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope. This may be intentional, but a different name should be used in case a reference to the global variable is attempted, and the local value changed or returned accidentally.

Coding standards

CERT DCL01-C

> Do not reuse variable names in subscopes

CERT DCL01-CPP

> Do not reuse variable names in subscopes

MISRA C++ 2008 2-10-2

> (Required) Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.

Code examples

The following code example fails the check and will give a warning:

```
int x;


int foo (int y ){
  int x=0;
  x++;
  return x+y;

}
```

The following code example passes the check and will not give a warning about this issue:

```
int x;

int foo (int y ){

  x++;
  return x+y;

}
```

MISRAC++2008-2-10-2_b

Synopsis

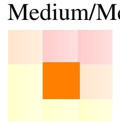
The definition of a local variable hides a previous local definition.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope. This may be intentional, but a different name should be used in case a reference to the outer variable is attempted, and the inner value changed or returned accidentally.

Coding standards

CERT DCL01-C

Do not reuse variable names in subscopes

CERT DCL01-CPP

Do not reuse variable names in subscopes

MISRA C++ 2008 2-10-2

(Required) Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.

Code examples

The following code example fails the check and will give a warning:

```
int foo(int x ){

  for (int y= 0; y < 10 ; y++){

    for (int y = 0; y < 100; y ++){
      return x+y;
    }
  }
  return x;
}


int foo2(int x){
  int y = 10;

  for (int y= 0; y < 10 ; y++)
    x++;
    return x;
}

int foo3(int x){

  int y = 10;
  {
    int y = 100;
    return x + y;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

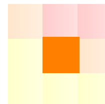
```
int foo(int x){

  for (int y=0; y < 10; y++)
    x++;
  for (int y=0; y < 10; y++)
    x++;
  return x;
}
```

MISRAC++2008-2-10-2_c

Synopsis	A variable declaration hides a parameter of the function
Enabled by default	Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope. This may be intentional, but a different name should be used in case a reference to the argument is attempted, and the inner value changed or returned accidentally.

Coding standards

CERT DCL01-C

Do not reuse variable names in subscopes

CERT DCL01-CPP

Do not reuse variable names in subscopes

MISRA C++ 2008 2-10-2

(Required) Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.

Code examples

The following code example fails the check and will give a warning:

```
int foo(int x){


  for (int x = 0; x < 100; x++);

  return x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int foo(int x){
  int y;

  return x;
}
```

MISRAC++2008-2-10-2_d (C++ only)

Synopsis

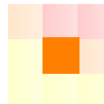
The definition of a local variable hides a member of the class.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope. This may be intentional, but a different name should be used in case a reference to the class member is attempted, and the local value changed or returned accidentally.

Coding standards

CERT DCL01-C

Do not reuse variable names in subscopes

CERT DCL01-CPP

Do not reuse variable names in subscopes

MISRA C++ 2008 2-10-2

(Required) Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.

Code examples

The following code example fails the check and will give a warning:

```
class A {
  int x;

public:

  void foo(int y){

    for(int x = 0; x < 10 ; x++){
      y++;
    }

  }

  void foo2(int y){
    int x = 0;
    x+=y;
    return;

  }

  void foo3(int y){

    {
      int x = 0;
      x+=y;
      return;
    }
  }


};
```

The following code example passes the check and will not give a warning about this issue:

```
class A {
  int x;

};


class B{
  int y;
void foo();
};


void B::foo() {
  int x;
}
```

MISRAC++2008-2-10-3

Synopsis
Typedef with this name already declared.

Enabled by default
Yes

Severity/Certainty
Low/Medium

Full description
(Required) A typedef name (including qualification, if any) shall be a unique identifier.

Coding standards
MISRA C++ 2008 2-10-3

> (Required) A typedef name (including qualification, if any) shall be a unique identifier.

Code examples
The following code example fails the check and will give a warning:

```
typedef int WIDTH;
//dummy comment
void f1()
{
  WIDTH w1;
}

void f2()
{
  typedef float WIDTH;
  WIDTH w2;
  WIDTH w3;
}
```

The following code example passes the check and will not give a warning about this issue:

```
namespace NS1
{
  typedef int WIDTH;
}
// f2.cc
namespace NS2
{
  typedef float WIDTH; // Compliant - NS2::WIDTH is not the same
as NS1::WIDTH
}
NS1::WIDTH w1;
NS2::WIDTH w2;
```

MISRAC++2008-2-10-4

Synopsis: A class, struct, union or enum declaration that clashes with a previous declaration.

Enabled by default: Yes

Severity/Certainty: Low/Medium

Full description: (Required) A class, union or enum name (including qualification, if any) shall be a unique identifier.

Coding standards

MISRA C++ 2008 2-10-4

(Required) A class, union or enum name (including qualification, if any) shall be a unique identifier.

Code examples

The following code example fails the check and will give a warning:

```
void f1()
{
  class TYPE {};
}

void f2()
{
  float TYPE; // non-compliant
}
```

The following code example passes the check and will not give a warning about this issue:

```
enum ENS {ONE, TWO };

void f1()
{
  class TYPE {};
}

void f4()
{
  union GRRR {
    int i;
    float f;
  };
}
```

MISRAC++2008-2-10-5

Synopsis

A identifier is used that can clash with another static identifier.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

(Advisory) The identifier name of a non-member object or function with static storage duration should not be reused.

Coding standards

MISRA C++ 2008 2-10-5

(Advisory) The identifier name of a non-member object or function with static storage duration should not be reused.

Code examples

The following code example fails the check and will give a warning:

```
namespace NS1
{
  static int global = 0;
}

namespace NS2
{
  void fn()
  {
    int global; // Non-compliant
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
namespace NS1
{
  int global = 0;
}

namespace NS2
{
  void f1()
  {
    int global; // Non-compliant
  }
}

void f2()
{
  static int global;
}
```

MISRAC++2008-2-13-2

Synopsis

Uses of octal integer constants

Enabled by default Yes

Severity/Certainty Low/Medium

Full description (Required) Octal constants (other than zero) and octal escape sequences (other than 0) shall not be used.

Coding standards MISRA C++ 2008 2-13-2

(Required) Octal constants (other than zero) and octal escape sequences (other than 0) shall not be used.

Code examples The following code example fails the check and will give a warning:

```
void
func(void)
{
    int x = 077;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void
func(void)
{
    int x = 63;
}
```

MISRAC++2008-2-13-3

Synopsis A U suffix shall be applied to all constants of unsigned type.

Enabled by default Yes

Severity/Certainty Low/Low

Full description

(Required) A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.

Coding standards

MISRA C++ 2008 2-13-3

(Required) A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  // 2147483648 -- does not fit in 31bits
  unsigned int x = 0x80000000;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  unsigned int x = 0x80000000u;
}
```

MISRAC++2008-2-13-4_a

Synopsis

Lower case suffixes on floating constants

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Literal suffixes shall be upper case.

Coding standards

MISRA C++ 2008 2-13-4

(Required) Literal suffixes shall be upper case.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdint.h>

void func()
{
  float       l = 2.4l;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdint.h>

void func()
{
  uint32_t    a = 0U;
  int64_t     c = 0L;
  uint64_t    e = 0UL;
  uint32_t    g = 0x12bU;
  float       i = 1.2F;
  float       k = 1.2L;
}
```

MISRAC++2008-2-13-4_b

Synopsis
Lower case suffixes on integer constants

Enabled by default
Yes

Severity/Certainty
Low/Medium

Full description
(Required) Literal suffixes shall be upper case.

Coding standards
CERT DCL16-C

Use 'L', not 'l', to indicate a long value

CERT DCL16-CPP

Use 'L', not 'l', to indicate a long value

MISRA C++ 2008 2-13-4

(Required) Literal suffixes shall be upper case.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdint.h>

void func()
{
  uint32_t    b = 0u;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdint.h>

void func()
{
  uint32_t    a = 0U;
  int64_t     c = 0L;
  uint64_t    e = 0UL;
  uint32_t    g = 0x12bU;
  float       i = 1.2F;
  float       k = 1.2L;
}
```

MISRAC++2008-3-1-1

Synopsis

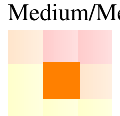
Non-inline functions defined in header files

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) It shall be possible to include any header file in multiple translation units without violating the One Definition Rule. Header files should not be used to define functions. This makes it clear that only C source files contain executable code. A header file is defined to be any file that is included in a translation unit via the #include directive.

Coding standards

MISRA C++ 2008 3-1-1

(Required) It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.

Code examples

The following code example fails the check and will give a warning:

```
#include "definition.h"
/* Contents of definition.h:

void definition(void) {
}

*/

void example(void) {
  definition();
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include "declaration.h"
/* Contents of declaration.h:

void definition(void);

*/

void example(void) {
  definition();
}
```

MISRAC++2008-3-1-3

Synopsis

External arrays declared without size stated explicitly or defined implicitly by initialization.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.

Coding standards

MISRA C++ 2008 3-1-3

(Required) When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.

Code examples

The following code example fails the check and will give a warning:

```
extern int a[];
```

The following code example passes the check and will not give a warning about this issue:

```
extern int a[10];
extern int b[] = { 0, 1, 2 };
```

MISRAC++2008-3-9-2

Synopsis

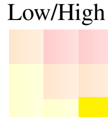
Uses of basic types char, int, short, long, double, and float without typedef

Enabled by default

No

Severity/Certainty

Low/High



Full description

(Advisory) typedefs that indicate size and signedness should be used in place of the basic numerical types. Best practice is to use typedefs for portability.

Coding standards

MISRA C++ 2008 3-9-2

(Advisory) typedefs that indicate size and signedness should be used in place of the basic numerical types.

Code examples

The following code example fails the check and will give a warning:

```
typedef signed charSCHAR;
typedef intINT;
typedef floatFLOAT;

INT func(FLOAT f, INT *pi)
{
    INT x;
    INT (*fp)(const char *);
}
```

The following code example passes the check and will not give a warning about this issue:

```
typedef signed charSCHAR;
typedef intINT;
typedef floatFLOAT;

INT func(FLOAT f, INT *pi)
{
    INT x;
    INT (*fp)(const SCHAR *);
}
```

MISRAC++2008-3-9-3

Synopsis

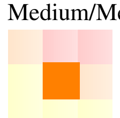
An expression provides access to the bit-representation of a floating point variable.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The underlying bit representations of floating-point values shall not be used.

Coding standards

MISRA C++ 2008 3-9-3

(Required) The underlying bit representations of floating-point values shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
void example(float f) {
  int * x = (int *)&f;
  int i = *x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(float f) {
  int i = (int)f;
}
```

MISRAC++2008-4-5-1

Synopsis

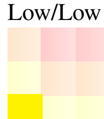
Uses of arithmetic operators on boolean operands.

Enabled by default

Yes

Severity/Certainty

Low/Low



Full description

(Required) Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator.

Coding standards

MISRA C++ 2008 4-5-1

> (Required) Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator.

Code examples

The following code example fails the check and will give a warning:

```
void func(bool b)
{
  bool x;
  bool y;
  y = x % b;
}
void example(void) {
   int x = 0;
   int y = 1;
   int a = 5;
   (a + (x || y)) ? example() : example();
}
void example(void) {
   int x = 0;
   int y = 1;
   int a = (x == y) << 2;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int
isgood(int ch)
{
    return (ch & 0x80) == 0;
}

int example(int r, int f1, int f2)
{
  if (r && f1 == f2)
    return 1;
  else
    return 0;
}
bool test()
{
  return true;
}

void example(void) {
  if(test()) {}
}
typedef charboolean_t;/* Compliant: Boolean-by-enforcement */

void example(void)
{
    boolean_t d;
    boolean_t c = 1;
    boolean_t b = 0;
    boolean_t a = 1;

    d = ( c && a ) && b;

}
class foo {
  int val;
public:
  bool operator==(const foo &rhs) const { return val == rhs.val;
}
};

int example(bool r, const foo &f1, const foo &f2)
{
  if (r && f1 == f2)
    return 1;
  else
    return 0;
}
```

```
void func(bool * ptr)
{
  if (*ptr) {}
}
void func()
{
  bool x;
  bool y;
  y = x && y;
}
typedef intboolean_t;

void example(void) {
    boolean_t x = 0;
    boolean_t y = 1;
    boolean_t a = 0;
    if (a && (x || y)) {
    }
}
void example(void) {
   int x = 0;
   int y = 1;
   int a = x == y;
}
#include <stdbool.h>

void example(void) {
    bool x = false;
    bool y = true;
    if (x || y) {
    }
}
typedef charboolean_t;
void example(void) {
    boolean_t x = 0;
    boolean_t y = 1;
    boolean_t a = x || y;
    a ? example() : example();
}
```

MISRAC++2008-4-5-2

Synopsis	Use of unsafe operators on variable of enumeration type.
Enabled by default	Yes

Severity/Certainty

Medium/Low

Full description

(Required) Expressions with type enum shall not be used as operands to builtin operators other than the subscript operator [ ], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=. ==, !=, &, [], or =. Other operators are unlikely to be meaningful (or intended).

Coding standards

MISRA C++ 2008 4-5-2

> (Required) Expressions with type enum shall not be used as operands to builtin operators other than the subscript operator [ ], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.

Code examples

The following code example fails the check and will give a warning:

```
enum ens { ONE, TWO, THREE };

void func(ens b)
{
  ens x;
  bool y;
  y = x | b;
}
```

The following code example passes the check and will not give a warning about this issue:

```
enum ens { ONE, TWO, THREE };

void func(ens b)
{
  ens y;
  y = b;
}
```

MISRAC++2008-4-5-3

Synopsis

Arithmetic on objects of type plain char, without an explicit signed or unsigned qualifier

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

(Required) Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator. such types explicitly as "signed char" or "unsigned char", to avoid unportable behavior.

Coding standards

CERT INT07-C

Use only explicitly signed or unsigned char type for numeric values

MISRA C++ 2008 4-5-3

(Required) Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator.

Code examples

The following code example fails the check and will give a warning:

```
typedefsigned charINT8;
typedefunsigned charUINT8;

UINT8
toascii(INT8 c)
{
    return (UINT8)c & 0x7f;
}

int func(int x)
{
    char sc = 4;
    char *scp = &sc;
    UINT8 (*fp)(INT8 c) = &toascii;

    x = x + sc;
    x *= *scp;
    return (*fp)(x);
}
```

The following code example passes the check and will not give a warning about this issue:

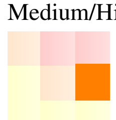
```
typedefsigned charINT8;
typedefunsigned charUINT8;

UINT8
toascii(INT8 c)
{
    return (UINT8)c & 0x7f;
}

int func(int x)
{
    signed char sc = 4;
    signed char *scp = &sc;
    UINT8 (*fp)(INT8 c) = &toascii;

    x = x + sc;
    x *= *scp;
    return (*fp)(x);
}
```

MISRAC++2008-5-0-1_a

Synopsis
Expressions which depend on order of evaluation

Enabled by default
Yes

Severity/Certainty
Medium/High



Full description
(Required) The value of an expression shall be the same under any order of evaluation that the standard permits. expression with an unspecified evaluation order, between two consecutive sequence points. ANSI C does not specify an evaluation order for different parts of an expression. For this reason different compilers are free to perform their own optimizations regarding the evaluation order. Projects containing statements that violate this check are not readily ported between architectures or compilers, and their ports may prove difficult to debug. Only four operators have a guaranteed order of evaluation: logical AND (`a && b`) evaluates the left operand, then the right operand only if the left is found to be true; logical OR (`a || b`) evaluates the left operand, then the right operand only if the left is found to be false; a ternary conditional (`a ? b : c`) evaluates the first operand, then either the second or the third, depending on whether the first is found to be true or false; and a comma (`a , b`) evaluates its left operand before its right.

Coding standards

CERT EXP10-C

Do not depend on the order of evaluation of subexpressions or the order in which side effects take place

CERT EXP30-C

Do not depend on order of evaluation between sequence points

CWE 696

Incorrect Behavior Order

MISRA C++ 2008 5-0-1

(Required) The value of an expression shall be the same under any order of evaluation that the standard permits.

Code examples

The following code example fails the check and will give a warning:

```
int main(void) {
  int i = 0;

  i = i * i++;  //unspecified order of operations

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int main(void) {
  int i = 0;
  int x = i;

  i++;
  x = x * i;  //OK - statement is broken up

  return 0;
}
```

MISRAC++2008-5-0-1_b

Synopsis

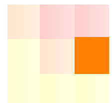
There shall be no more than one read access with volatile-qualified type within one sequence point

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

(Required) The value of an expression shall be the same under any order of evaluation that the standard permits.

Coding standards

CERT EXP10-C

Do not depend on the order of evaluation of subexpressions or the order in which side effects take place

CERT EXP30-C

Do not depend on order of evaluation between sequence points

CWE 696

Incorrect Behavior Order

MISRA C++ 2008 5-0-1

(Required) The value of an expression shall be the same under any order of evaluation that the standard permits.

Code examples

The following code example fails the check and will give a warning:

```
#include "mc2_types.h"
#include "mc2_header.h"

void example(void) {
  uint16_t x;
  volatile uint16_t v;
  x = v + v;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int main(void) {
  int i = 0;
  int x = i;

  i++;
  x = x * i;  //OK - statement is broken up

  return 0;
}
```

MISRAC++2008-5-0-1_c

Synopsis

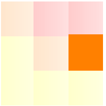
There shall be no more than one modification access with volatile-qualified type within one sequence point

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

(Required) The value of an expression shall be the same under any order of evaluation that the standard permits.

Coding standards

CERT EXP10-C

Do not depend on the order of evaluation of subexpressions or the order in which side effects take place

CERT EXP30-C

Do not depend on order of evaluation between sequence points

CWE 696

Incorrect Behavior Order

MISRA C++ 2008 5-0-1

(Required) The value of an expression shall be the same under any order of evaluation that the standard permits.

Code examples

The following code example fails the check and will give a warning:

```
#include "mc2_types.h"
#include "mc2_header.h"

void example(void) {
  uint16_t x;
  volatile uint16_t v, w;
  v = w = x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdbool.h>
void InitializeArray(int *);
const int *example(void)
{
  static volatile bool s_initialized = false;
  static int s_array[256];

  if (!s_initialized)
  {
    InitializeArray(s_array);
    s_initialized = true;
  }
  return s_array;
}
```

MISRAC++2008-5-0-2

Synopsis

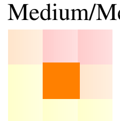
Add parentheses to avoid implicit operator precedence.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

(Advisory) Limited dependence should be placed on C++ operator precedence rules in expressions.

Coding standards

MISRA C++ 2008 5-0-2

> (Advisory) Limited dependence should be placed on C++ operator precedence rules in expressions.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
    int i;
    int j;
    int k;
    int result;

    result = i + j * k;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
    int i;
    int j;
    int k;
    int result;

    result = i + (j - k);
}
```

MISRAC++2008-5-0-3

Synopsis

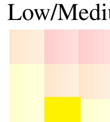
A cvalue expression shall not be implicitly converted to a different underlying type.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) A cvalue expression shall not be implicitly converted to a different underlying type.

Coding standards

MISRA C++ 2008 5-0-3

(Required) A cvalue expression shall not be implicitly converted to a different underlying type.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdint.h>
void f ( )
{
  int32_t s32;
  int8_t s8;
  s32 = s8 + s8; // Example 1 - Non-compliant
  // The addition operation is performed with an underlying type
of int8_t and the result
  // is converted to an underlying type of int32_t.
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdint.h>
void f ( )
{
  int32_t s32;
  int8_t s8;
  s32 = static_cast < int32_t > ( s8 ) + s8; // Example 2 -
Compliant
  // the addition is performed with an underlying type of int32_t
and therefore
  // no underlying type conversion is required.
}
#include <stdint.h>
void f ( )
{
  int32_t s32;
  int8_t s8;
  s32 = s32 + s8; // Example 3 - Compliant
  // the addition is performed with an underlying type of int32_t
and therefore
  // no underlying type conversion is required.
}
```

MISRAC++2008-5-0-4

Synopsis

An implicit integral conversion shall not change the signedness of the underlying type.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) An implicit integral conversion shall not change the signedness of the underlying type.

Coding standards

MISRA C++ 2008 5-0-4

(Required) An implicit integral conversion shall not change the signedness of the underlying type.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdint.h>
void f()
{
  int8_t s8;
  uint8_t u8;
  u8 = s8 + u8; // Non-compliant
}
```

```
#include <stdint.h>
void f()
{
  int8_t s8;
  uint8_t u8;
  s8 = u8; // Non-compliant
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdint.h>
void f()
{
  int8_t s8;
  uint8_t u8;
  u8 = static_cast< uint8_t > ( s8 ) + u8; // Compliant
}
```

MISRAC++2008-5-0-5

Synopsis There shall be no implicit floating-integral conversions.

Enabled by default Yes

Severity/Certainty Low/Medium

Full description (Required) There shall be no implicit floating-integral conversions.

Coding standards MISRA C++ 2008 5-0-5

(Required) There shall be no implicit floating-integral conversions.

Code examples

The following code example fails the check and will give a warning:

```
#include "mc2_types.h"
void f()
{
  float32_t f32;
  int32_t s32;
  f32 = s32; // Non-compliant
}

#include "mc2_types.h"
void f()
{
  float32_t f32;
  int32_t s32;
  s32 = f32; // Non-compliant
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include "mc2_types.h"
void f()
{
  float32_t f32;
  int32_t s32;
  f32 = static_cast< float32_t > ( s32 ); // Compliant
}
```

MISRAC++2008-5-0-6

Synopsis

An implicit integral or floating-point conversion shall not reduce the size of the underlying type.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) An implicit integral or floating-point conversion shall not reduce the size of the underlying type.

Coding standards

MISRA C++ 2008 5-0-6

> (Required) An implicit integral or floating-point conversion shall not reduce the size of the underlying type.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdint.h>
void f ( )
{
  int32_t s32;
  int16_t s16;
  s16 = s32; // Non-compliant
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdint.h>
void f ( )
{
  int32_t s32;
  int16_t s16;
  s16 = static_cast< int16_t > ( s32 ); // Compliant
}
```

MISRAC++2008-5-0-7

Synopsis

There shall be no explicit floating-integral conversions of a cvalue expression.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) There shall be no explicit floating-integral conversions of a cvalue expression.

Coding standards

MISRA C++ 2008 5-0-7

> (Required) There shall be no explicit floating-integral conversions of a cvalue expression.

Code examples

The following code example fails the check and will give a warning:

```
#include "mc2_types.h"
// Integral to Float
void f1 ( )
{
  int16_t s16a;
  int16_t s16b;
  float32_t f32a;
  // The following performs integer division
  f32a = static_cast< float32_t > ( s16a / s16b ); //
Non-compliant
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include "mc2_types.h"
// Integral to Float
void f1 ( )
{
  int16_t s16a;
  int16_t s16b;
  int16_t s16c;
  float32_t f32a;
  // The following also performs integer division
  s16c = s16a / s16b;
  f32a = static_cast< float32_t > ( s16c ); // Compliant
}

#include "mc2_types.h"
// Integral to Float
void f1 ( )
{
  int16_t s16a;
  int16_t s16b;
  float32_t f32a;
  // The following performs floating-point division
  f32a = static_cast< float32_t > ( s16a ) / s16b; // Compliant
}
```

MISRAC++2008-5-0-8

Synopsis

An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.

Enabled by default Yes

Severity/Certainty Low/Medium

Full description (Required) An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.

Coding standards MISRA C++ 2008 5-0-8

> (Required) An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.

Code examples The following code example fails the check and will give a warning:

```
#include <stdint.h>
void f ( )
{
  int16_t s16;
  int32_t s32;
  s32 = static_cast< int32_t > ( s16 + s16 ); // Non-compliant
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdint.h>
void f ( )
{
  int16_t s16;
  int32_t s32;
  s32 = static_cast< int32_t > ( s16 ) + s16 ; // Compliant
}
```

MISRAC++2008-5-0-9

Synopsis An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.

Enabled by default Yes

Severity/Certainty

Low/Medium

Full description

(Required) An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.

Coding standards

MISRA C++ 2008 5-0-9

> (Required) An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdint.h>
void f ( )
{
  int8_t s8;
  uint8_t u8;
  s8 = static_cast< int8_t >( u8 + u8 ); // Non-compliant
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdint.h>
void f ( )
{
  int8_t s8;
  uint8_t u8;
  s8 = static_cast< int8_t >( u8 )
     + static_cast< int8_t >( u8 ); // Compliant
}
```

MISRAC++2008-5-0-10

Synopsis

Bitwise operation on unsigned char or unsigned short, that are not immediately cast to this type to ensure consistent truncation

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.

Coding standards

MISRA C++ 2008 5-0-10

> (Required) If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.

Code examples

The following code example fails the check and will give a warning:

```
typedef unsigned char uint8_t;
typedef unsigned short uint16_t;

void example(void) {
  uint8_t port = 0x5aU;
  uint8_t result_8;
  uint16_t result_16;
  uint8_t mode;

  result_16 = ((port << 4) & mode) >> 6;
}
typedef unsigned char uint8_t;
typedef unsigned short uint16_t;

void example(void) {
  uint8_t port = 0x5aU;
  uint8_t result_8;
  uint16_t result_16;
  uint16_t mode;

  result_8 = (~port) >> 4;
}
```

The following code example passes the check and will not give a warning about this issue:

```
typedef unsigned char uint8_t;
typedef unsigned short uint16_t;

void example(void) {
  uint8_t port = 0x5aU;
  uint8_t result_8;
  uint16_t result_16;
  uint16_t mode;

  result_8 = ( static_cast< uint8_t > (~port) ) >> 4; //
Compliant
  result_16 = ( static_cast < uint16_t > ( static_cast< uint16_t
> ( port ) << 4 ) & mode ) >> 6; // Compliant
}
typedef unsigned char uint8_t;
typedef unsigned short uint16_t;

void example(void) {
  uint8_t port = 0x5aU;
  uint8_t result_8;
  uint16_t result_16;
  uint16_t mode;

  uint16_t port_16 = static_cast< uint16_t > ( port );
  uint16_t port_shifted = static_cast< uint16_t > ( port_16 << 4
);
  result_16 = ( port_shifted & mode ) >> 6; // Compliant
}
```

MISRAC++2008-5-0-13_a

Synopsis: Non-boolean termination conditions in do ... while statements.

Enabled by default: Yes

Severity/Certainty: Low/Medium

Full description: (Required) The condition of an if-statement and the condition of an iteration-statement shall have type bool.

Coding standards: MISRA C++ 2008 5-0-13

(Required) The condition of an if-statement and the condition of an iteration-statement shall have type bool.

Code examples

The following code example fails the check and will give a warning:

```
typedefintint32_t;
int32_t func();

void example(void)
{
  do {
  } while (func());
}
```

The following code example passes the check and will not give a warning about this issue:

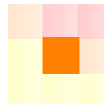
```
#include <stddef.h>

int * fn()
{
  int * ptr;
  return ptr;
}

int fn2()
{
  return 5;
}

bool fn3()
{
  return true;
}

void example(void)
{
  while (int *ptr = fn() )  // Compliant by exception
  {}

  do
  {
    int *ptr = fn();
    if ( NULL == ptr )
    {
      break;
    }
  }
  while (true); // Compliant

  while (int len = fn2() )  // Compliant by exception
  {}

  if (int *p = fn()) {}   // Compliant by exception
  if (int len = fn2() ) {} // Complioant by exception
  if (bool flag = fn3()) {} // Compliant
}
```

MISRAC++2008-5-0-13_b

Synopsis	Non-boolean termination conditions in for loops.
Enabled by default	Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The condition of an if-statement and the condition of an iteration-statement shall have type bool.

Coding standards

MISRA C++ 2008 5-0-13

> (Required) The condition of an if-statement and the condition of an iteration-statement shall have type bool.

Code examples

The following code example fails the check and will give a warning:

```
void example(void)
{
  for (int x = 10;x;--x) {}
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stddef.h>

int * fn()
{
  int * ptr;
  return ptr;
}

int fn2()
{
  return 5;
}

bool fn3()
{
  return true;
}

void example(void)
{
  for (fn(); fn3(); fn2())  // Compliant
  {}

  for (fn(); true; fn()) // Compliant
  {
    int *ptr = fn();
    if ( NULL == ptr )
    {
      break;
    }
  }

  for (int len = fn2(); len < 10; len++)  // Compliant
    ;
}
```

MISRAC++2008-5-0-13_c

Synopsis	Non-boolean conditions in if statements.
Enabled by default	Yes

Severity/Certainty

Low/Medium

Full description

(Required) The condition of an if-statement and the condition of an iteration-statement shall have type bool.

Coding standards

MISRA C++ 2008 5-0-13

> (Required) The condition of an if-statement and the condition of an iteration-statement shall have type bool.

Code examples

The following code example fails the check and will give a warning:

```
void example(void)
{
  int u8;
  if (u8) {}
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stddef.h>

int * fn()
{
  int * ptr;
  return ptr;
}

int fn2()
{
  return 5;
}

bool fn3()
{
  return true;
}

void example(void)
{
  while (int *ptr = fn() )  // Compliant by exception
  {}

  do
  {
    int *ptr = fn();
    if ( NULL == ptr )
    {
      break;
    }
  }
  while (true); // Compliant

  while (int len = fn2() )  // Compliant by exception
  {}

  if (int *p = fn()) {}   // Compliant by exception
  if (int len = fn2() ) {} // Complioant by exception
  if (bool flag = fn3()) {} // Compliant
}
```

MISRAC++2008-5-0-13_d

Synopsis	Non-boolean termination conditions in while statements.
Enabled by default	Yes

Severity/Certainty

Low/Medium

Full description

(Required) The condition of an if-statement and the condition of an iteration-statement shall have type bool.

Coding standards

MISRA C++ 2008 5-0-13

> (Required) The condition of an if-statement and the condition of an iteration-statement shall have type bool.

Code examples

The following code example fails the check and will give a warning:

```
void example(void)
{
  int u8;
  while (u8) {}
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stddef.h>

int * fn()
{
  int * ptr;
  return ptr;
}

int fn2()
{
  return 5;
}

bool fn3()
{
  return true;
}

void example(void)
{
  while (int *ptr = fn() )  // Compliant by exception
  {}

  do
  {
    int *ptr = fn();
    if ( NULL == ptr )
    {
      break;
    }
  }
  while (true); // Compliant

  while (int len = fn2() )  // Compliant by exception
  {}

  if (int *p = fn()) {}   // Compliant by exception
  if (int len = fn2() ) {} // Complioant by exception
  if (bool flag = fn3()) {} // Compliant
}
```

MISRAC++2008-5-0-14

Synopsis	Non-boolean operands to the conditional (? :) operator
Enabled by default	Yes

Severity/Certainty

Low/Medium

Full description

(Required) The first operand of a conditional-operator shall have type bool.

Coding standards

MISRA C++ 2008 5-0-14

(Required) The first operand of a conditional-operator shall have type bool.

Code examples

The following code example fails the check and will give a warning:

```
void example(int x) {
  int z;
  z = x ? 1 : 2;  //x is an int, not a bool
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(int x) {
  int z;
  z = x + 0 > 3 ? 1 : 2;  //OK - the condition is a comparison
}
void example(bool b) {
  int x;
  x = b ? 1 : 2;  //OK - b is a bool
}
```

MISRAC++2008-5-0-15_a

Synopsis

Array indexing shall be the only allowed form of pointer arithmetic.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Array indexing shall be the only form of pointer arithmetic.

Coding standards

MISRA C++ 2008 5-0-15

(Required) Array indexing shall be the only form of pointer arithmetic.

Code examples

The following code example fails the check and will give a warning:

```
typedef int INT32;

void example(INT32 array[]) {
  INT32 *pointer = array;
  INT32 *end = array + 10;
  for (; pointer != end; pointer += 1) {
    *pointer = 0;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
typedef int INT32;

void example(INT32 array[]) {
  INT32 index = 0;
  INT32 end = 10;
  for (; index != end; index += 1) {
    array[index] = 0;
  }
}
```

MISRAC++2008-5-0-15_b

Synopsis

Array indexing shall only be applied to objects defined as an array type.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Array indexing shall be the only form of pointer arithmetic.

Coding standards

MISRA C++ 2008 5-0-15

(Required) Array indexing shall be the only form of pointer arithmetic.

Code examples

The following code example fails the check and will give a warning:

```
typedef unsigned charUINT8;
typedefunsigned intUINT;

void example(UINT8 *p, UINT size) {
  UINT i;
  for (i = 0; i < size; i++) {
    p[i] = 0;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
typedef unsigned charUINT8;
typedef unsigned intUINT;

void example(void) {
  UINT8 p[10];
  UINT  i;
  for (i = 0; i < 10; i++) {
    p[i] = 0;
  }
}
```

MISRAC++2008-5-0-16_a

Synopsis

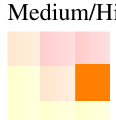
Pointer arithmetic applied to a pointer that references a stack address

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

(Required) A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.

Coding standards

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

MISRA C++ 2008 5-0-16

(Required) A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
    int i;
    int *p = &i;
    p++;
    *p = 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
    int i;
    int *p = &i;
    *p = 0;
}
```

MISRAC++2008-5-0-16_b

Synopsis

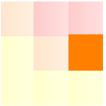
Invalid pointer arithmetic with an automatic variable that is neither an array nor a pointer.

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

(Required) A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array. This check warns when the address of an automatic variable is taken, and arithmetic is performed on it, as this behavior indicates that an invalid memory access attempt may occur. It handles local variables, parameters and globals, including structs.

Coding standards

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

MISRA C++ 2008 5-0-16

(Required) A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.

Code examples

The following code example fails the check and will give a warning:

```
void example(int x) {
  *(&x+10) = 5;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(int *x) {
  *(x+10) = 5;
}
```

MISRAC++2008-5-0-16_c

Synopsis

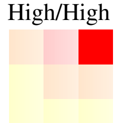
Array access is out of bounds.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

(Required) A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array. This is likely to corrupt data and/or crash the program, and may result in security vulnerabilities.

Coding standards

CERT ARR33-C

Guarantee that copies are made into storage of sufficient size

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

CWE 121

Stack-based Buffer Overflow

CWE 124

Buffer Underwrite ('Buffer Underflow')

CWE 126

Buffer Over-read

CWE 127

Buffer Under-read

CWE 129

Improper Validation of Array Index

MISRA C++ 2008 5-0-16

(Required) A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.

Code examples

The following code example fails the check and will give a warning:

```
/* Goanna correctly detects that the array access,
   a[x - 10] is always within bounds, because 'x'
   is always in the range 10 <= x < 20, but a[x]
   is not. */

int ex(int x, int y)
{
  int a[10];

  if((x >= 0) && (x < 20)) {
    if(x < 10) {
      y = a[x];
    } else {
      y = a[x - 10];
      y = a[x];
    }
  }

  return y;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int main(void)
{
  int a[4];

  a[3] = 0;

  return 0;
}
```

MISRAC++2008-5-0-16_d

Synopsis

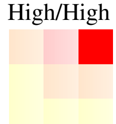
Array access may be out of bounds, depending on which path is executed.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

(Required) A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array. This may corrupt data and/or crash the program, and may also result in security vulnerabilities.

Coding standards

CERT ARR33-C

Guarantee that copies are made into storage of sufficient size

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

CWE 121

Stack-based Buffer Overflow

CWE 124

Buffer Underwrite ('Buffer Underflow')

CWE 126

Buffer Over-read

CWE 127

Buffer Under-read

CWE 129

Improper Validation of Array Index

MISRA C++ 2008 5-0-16

(Required) A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.

Code examples

The following code example fails the check and will give a warning:

```
int cond;

int main(void)
{
  int a[7];
  int x;

  if (cond)
    x = 3;
  else
    x = 20;

  a[x] = 0;  //x may be set to 20 in line 11
             //but a only has an interval of [0,6]
  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int cond;

int main(void)
{
  int a[25];
  int x;

  if (cond)
    x = 3;
  else
    x = 20;

  a[x] = 0;  //here, both possible values of
             //x are in the interval [0,24]
  return 0;
}
```

MISRAC++2008-5-0-16_e

Synopsis

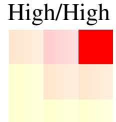
A pointer to an array is used outside the array bounds

Enabled by default

Yes

Severity/Certainty

High/High



Full description

(Required) A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.

Coding standards

CERT ARR33-C

Guarantee that copies are made into storage of sufficient size

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 120

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

CWE 121

Stack-based Buffer Overflow

CWE 122

Heap-based Buffer Overflow

CWE 124

Buffer Underwrite ('Buffer Underflow')

CWE 126

Buffer Over-read

CWE 127

Buffer Under-read

CWE 129

Improper Validation of Array Index

MISRA C++ 2008 5-0-16

(Required) A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int arr[10];
  int *p = arr;
  p[10];
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int arr[10];
  int *p = arr;
  p[9];
}
```

MISRAC++2008-5-0-16_f

Synopsis

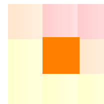
A pointer to an array is potentially used outside the array bounds

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.

Coding standards

CERT ARR33-C

> Guarantee that copies are made into storage of sufficient size

CWE 119

> Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 120

> Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

CWE 121

> Stack-based Buffer Overflow

CWE 122

> Heap-based Buffer Overflow

CWE 124

> Buffer Underwrite ('Buffer Underflow')

CWE 126

> Buffer Over-read

CWE 127

> Buffer Under-read

CWE 129

> Improper Validation of Array Index

MISRA C++ 2008 5-0-16

> (Required) A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.

Code examples

The following code example fails the check and will give a warning:

```
void example(int b) {
  int arr[10];
  int *p = arr;
  int x = (b<10 ? 8 : 11);
  p[x];
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(int b) {
  int arr[12];
  int *p = arr;
  int x = (b<10 ? 8 : 11);
  p[x];
}
```

MISRAC++2008-5-0-19

Synopsis

The declaration of objects should contain no more than two levels of pointer indirection.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The declaration of objects shall contain no more than two levels of pointer indirection.

Coding standards

MISRA C++ 2008 5-0-19

(Required) The declaration of objects shall contain no more than two levels of pointer indirection.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
    int ***p;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
    int **p;
}
```

MISRAC++2008-5-0-21

Synopsis

Applications of bitwise operators to signed operands

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Bitwise operators shall only be applied to operands of unsigned underlying type.

Coding standards

CERT INT13-C

Use bitwise operators only on unsigned operands

MISRA C++ 2008 5-0-21

(Required) Bitwise operators shall only be applied to operands of unsigned underlying type.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int x = -(1U);

  x ^ 1;
  x & 0x7F;
  ((unsigned int)x) & 0x7F;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int x = -1;
  ((unsigned int)x) ^ 1U;
  2U ^ 1U;
  ((unsigned int)x) & 0x7FU;
  ((unsigned int)x) & 0x7FU;
}
```

MISRAC++2008-5-2-4 (C++ only)

Synopsis

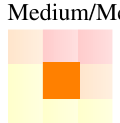
Uses of old style casts (other than void casts)

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used. cast. This may cause portability problems, e.g. a particular cast may not be valid on a system, but the compiler will perform the cast anyway. The new style casts static_cast, const_cast, and reinterpret_cast should be used instead because they make clear the intention of the cast. Also, the new style casts can easily be searched for in source code files, unlike old style casts.

Coding standards

CERT EXP05-CPP

Do not use C-style casts

MISRA C++ 2008 5-2-4

(Required) C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
int example(float b)
{
    return (int)b;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(float b)
{
    return static_cast<int>(b);
}
```

MISRAC++2008-5-2-5

Synopsis

Casts that remove any const or volatile qualification.

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

(Required) A cast shall not remove any const or volatile qualification from the type of a pointer or reference. This violates the principle of type qualification. This check does not look for changes to the qualification of the pointer during the cast.

Coding standards

MISRA C++ 2008 5-2-5

> (Required) A cast shall not remove any const or volatile qualification from the type of a pointer or reference.

Code examples

The following code example fails the check and will give a warning:

```
typedef unsigned short uint16_t;

void example(void) {

  uint16_t x;
  const uint16_t *    pci;      /* pointer to const int */
  uint16_t *          pi;       /* pointer to int */

  pi = (uint16_t *)pci; // not compliant

}
```

The following code example passes the check and will not give a warning about this issue:

```
typedef unsigned short uint16_t;

void example(void) {

  uint16_t x;
  uint16_t * const    cpi = &x; /* const pointer to int */
  uint16_t *          pi;       /* pointer to int */

  pi = cpi; // compliant - no cast required

}
```

MISRAC++2008-5-2-6

Synopsis

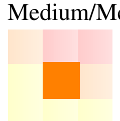
A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.

Coding standards

MISRA C++ 2008 5-2-6

> (Required) A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdint.h>
void f ( int32_t )
{
  reinterpret_cast< void * >( &f ); // Non-compliant
}

#include <stdint.h>
void f ( int32_t )
{
  reinterpret_cast< void (*)( ) >( &f ); // Non-compliant
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdint.h>
void f ( int32_t )
{
  void (*fp)(int32_t) = &f;
}
 void example(void) {
  (*((volatile unsigned long*) 0xE0028004UL)) = (1UL << 10UL);
}
```

MISRAC++2008-5-2-7

Synopsis	A pointer to object type is cast to a pointer to different object type
Enabled by default	Yes
Severity/Certainty	Low/Medium 
Full description	(Required) An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly. Conversions of this type may be invalid if the new pointer type required a stricter alignment.
Coding standards	MISRA C++ 2008 5-2-7

(Required) An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.

Code examples

The following code example fails the check and will give a warning:

```
typedef unsigned int uint32_t;
typedef unsigned char uint8_t;

void example(void) {
  uint8_t * p1;
  uint32_t * p2;
  p2 = (uint32_t *)p1;
}
```

The following code example passes the check and will not give a warning about this issue:

```
typedef unsigned int uint32_t;
typedef unsigned char uint8_t;

void example(void) {
  uint8_t * p1;
  uint8_t * p2;
  p2 = (uint8_t *)p1;
}
```

MISRAC++2008-5-2-9

Synopsis

A cast should not be performed between a pointer type and an integral type.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

(Advisory) A cast should not convert a pointer type to an integral type.

Coding standards

MISRA C++ 2008 5-2-9

(Advisory) A cast should not convert a pointer type to an integral type.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {

   int *p;
   int x;

   x = (int)p;

}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {

   int *p;
   int *x;

   x = p;

}
```

MISRAC++2008-5-2-10

Synopsis

Uses of increment (++) and decrement (--) operators mixed with other operators in an expression.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

(Advisory) The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.

Coding standards

MISRA C++ 2008 5-2-10

> (Advisory) The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.

Code examples

The following code example fails the check and will give a warning:

```
void example(char *src, char *dst) {
  while ((*src++ = *dst++));
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(char *src, char *dst) {
  while (*src) {
    *dst = *src;
    src++;
    dst++;
  }
}
```

MISRAC++2008-5-2-11_a (C++ only)

Synopsis

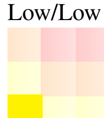
Overloaded && and || operators

Enabled by default

Yes

Severity/Certainty

Low/Low



Full description

(Required) The comma operator, && operator and the || operator shall not be overloaded. function calls whose sequence point and ordering semantics are different from those of the built- in versions. It may not be clear at the point of use that these operators are overloaded, and so developers may be unaware which semantics apply.

Coding standards

MISRA C++ 2008 5-2-11

(Required) The comma operator, && operator and the || operator shall not be overloaded.

Code examples

The following code example fails the check and will give a warning:

```
class C{
  bool x;
  bool operator||(bool other);
};

bool C::operator||(bool other){
  return x || other;
}
```

The following code example passes the check and will not give a warning about this issue:

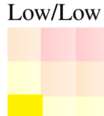
```
class C{
  int x;
  int operator+(int other);
};

int C::operator+(int other){
  return x + other;
}
```

MISRAC++2008-5-2-11_b (C++ only)

Synopsis Overloaded comma operator

Enabled by default Yes

Severity/Certainty Low/Low



Full description (Required) The comma operator, && operator and the || operator shall not be overloaded. function calls whose sequence point and ordering semantics are different from those of the built- in versions. It may not be clear at the point of use that these operators are overloaded, and so developers may be unaware which semantics apply.

Coding standards MISRA C++ 2008 5-2-11

> (Required) The comma operator, && operator and the || operator shall not be overloaded.

Code examples The following code example fails the check and will give a warning:

```
class C{
  bool x;
  bool operator,(bool other);
};

bool C::operator,(bool other){
  return x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
class C{
  int x;
  int operator+(int other);
};

int C::operator+(int other){
  return x + other;
}
```

MISRAC++2008-5-3-1

Synopsis

Operands of logical operators (&&, ||, and !) that are not of type bool.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Each operand of the ! operator, the logical && or the logical || operators shall have type bool.

Coding standards

MISRA C++ 2008 5-3-1

> (Required) Each operand of the ! operator, the logical && or the logical || operators shall have type bool.

Code examples

The following code example fails the check and will give a warning:

```
void func(int * ptr)
{
  if (!ptr) {}
}
void func()
{
  if (!0) {}
}
void example(void) {
   int x = 0;
   int y = 1;
   int a = x || y << 2;
}
void example(void) {
   int x = 0;
   int y = 1;
   int a = 5;
   (a + (x || y)) ? example() : example();
}
void example(void) {
   int x = 5;
   int y = 11;
   if (x || y) {
   }
}
void example(void) {

   int d, c, b, a;

   d = ( c & a ) && b;

}
```

The following code example passes the check and will not give a warning about this issue:

```
bool test()
{
  return true;
}

void example(void) {
  if(test()) {}
}
typedef charboolean_t;/* Compliant: Boolean-by-enforcement */

void example(void)
{
    boolean_t d;
    boolean_t c = 1;
    boolean_t b = 0;
    boolean_t a = 1;

    d = ( c && a ) && b;

}
void func(bool * ptr)
{
  if (*ptr) {}
}
typedef intboolean_t;

void example(void) {
    boolean_t x = 0;
    boolean_t y = 1;
    boolean_t a = 0;
    if (a && (x || y)) {
    }
}
void example(void) {
  int x = 0;
  int y = 1;
  int a = x == y;
}
#include <stdbool.h>

void example(void) {
    bool x = false;
    bool y = true;
    if (x || y) {
    }
}
typedef charboolean_t;
```

```
void example(void) {
    boolean_t x = 0;
    boolean_t y = 1;
    boolean_t a = x || y;
    a ? example() : example();
}
```

MISRAC++2008-5-3-2_a

Synopsis
Uses of unary - on unsigned expressions

Enabled by default
Yes

Severity/Certainty
Low/Medium

Full description
(Required) The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

Coding standards
MISRA C++ 2008 5-3-2

> (Required) The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

Code examples
The following code example fails the check and will give a warning:

```
void example(void) {
  unsigned int max = -1U;
  // use max = ~0U;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int neg_one = -1;
}
```

MISRAC++2008-5-3-2_b

Synopsis
Uses of unary - on unsigned expressions

Enabled by default Yes

Severity/Certainty Low/Medium

Full description (Required) The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

Coding standards MISRA C++ 2008 5-3-2

(Required) The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

Code examples The following code example fails the check and will give a warning:

```
void example(void) {
  unsigned int max = -1U;
  // use max = ~0U;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int neg_one = -1;
}
```

MISRAC++2008-5-3-3 (C++ only)

Synopsis The & operator shall not be overloaded.

Enabled by default Yes

Severity/Certainty Low/Low

Full description (Required) The unary & operator shall not be overloaded.

Coding standards MISRA C++ 2008 5-3-3

(Required) The unary & operator shall not be overloaded.

Code examples

The following code example fails the check and will give a warning:

```
class C{
  bool x;
  bool* operator&();
};

bool* C::operator&(){
  return &x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
class C{
  int x;
  int operator+(int other);
};

int C::operator+(int other){
  return x + other;
}
```

MISRAC++2008-5-3-4

Synopsis

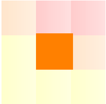
Sizeof expressions containing side effects

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Evaluation of the operand to the sizeof operator shall not contain side effects. side effects. The expectation of the programmer might be that the expression will be evaluated. However because sizeof only operates on the type of the expression, the expression itself is not evaluated.

Coding standards

CERT EXP06-C

Operands to the sizeof operator should not contain side effects

CERT EXP06-CPP

> Operands to the sizeof operator should not contain side effects

MISRA C++ 2008 5-3-4

> (Required) Evaluation of the operand to the sizeof operator shall not contain side effects.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int i;
  int size = sizeof(i++);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int i;
  int size = sizeof(i);
  i++;
}
```

MISRAC++2008-5-8-1

Synopsis

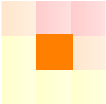
Out of range shifts

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand. In this case, the right-hand operand may be negative, or too large. This check is for all platforms. The behavior in this situation is undefined; the code may work as intended, or data could become erroneous.

Coding standards

CERT INT34-C

> Do not shift a negative number of bits or more bits than exist in the operand

CWE 682

Incorrect Calculation

MISRA C++ 2008 5-8-1

(Required) The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.

Code examples

The following code example fails the check and will give a warning:

```
unsigned int foo(unsigned long long x, unsigned int y)
{
  int shift = 65; // too big
  return 3ULL << shift;
}
unsigned int foo(unsigned int x, unsigned int y)
{
  int shift = 33; // too big
  return 3U << shift;
}
```

The following code example passes the check and will not give a warning about this issue:

```
unsigned int foo(unsigned int x)
{
  int y = 1;  // OK - this is within the correct range
  return x << y;
}
unsigned int foo(unsigned long long x)
{
  int y = 63; // ok
  return x << y;
}
```

MISRAC++2008-5-14-1

Synopsis

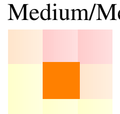
Right hand operands of && or || that contain side effects

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The right hand operand of a logical && or || operator shall not contain side effects.

Coding standards

CWE 768

Incorrect Short Circuit Evaluation

MISRA C++ 2008 5-14-1

(Required) The right hand operand of a logical && or || operator shall not contain side effects.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

void example(void) {
  int i;
  int size = rand() && i++;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

void example(void) {
  int i;
  int size = rand() && i;
}
```

MISRAC++2008-5-18-1

Synopsis

Uses of the comma operator

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

(Required) The comma operator shall not be used.

Coding standards

MISRA C++ 2008 5-18-1

(Required) The comma operator shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
#include <string.h>

void reverse(char *string) {
  int i, j;
  j = strlen(string);
  for (i = 0; i < j; i++, j--) {
    char temp = string[i];
    string[i] = string[j];
    string[j] = temp;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <string.h>

void reverse(char *string) {
  int i;
  int length = strlen(string);
  int half_length = length / 2;
  for (i = 0; i < half_length; i++) {
    int opposite = length - i;
    char temp = string[i];
    string[i] = string[opposite];
    string[opposite] = temp;
  }
}
```

MISRAC++2008-5-19-1

Synopsis

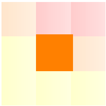
A constant unsigned integer expression overflows

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

(Advisory) Evaluation of constant unsigned integer expressions should not lead to wrap-around.

Coding standards

MISRA C++ 2008 5-19-1

(Advisory) Evaluation of constant unsigned integer expressions should not lead to wrap-around.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
   (0xFFFFFFFF + 1u);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
   0x7FFFFFFF + 0;
}
```

MISRAC++2008-6-2-1

Synopsis

Assignment in a sub-expression.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Assignment operators shall not be used in sub-expressions.

Coding standards

MISRA C++ 2008 6-2-1

(Required) Assignment operators shall not be used in sub-expressions.

Code examples

The following code example fails the check and will give a warning:

```
void func()
{
  int x;
  int y;
  int z;
  x = y = z;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void func()
{
  int x = 2;
  int y;
  int z;
  x = y;
  x == y;
}
```

MISRAC++2008-6-2-2

Synopsis

Floating point comparisons using == or !=

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

(Required) Floating-point expressions shall not be directly or indirectly tested for equality or inequality. The comparison will potentially be evaluated incorrectly, especially if either of the floats have been operated on arithmetically. In such a case, program logic will be compromised.

Coding standards

CERT FLP06-C

Understand that floating-point arithmetic in C is inexact

CERT FLP35-CPP

Take granularity into account when comparing floating point values

MISRA C++ 2008 6-2-2

(Required) Floating-point expressions shall not be directly or indirectly tested for equality or inequality.

Code examples

The following code example fails the check and will give a warning:

```
int main(void)
{
  float f = 3.0;
  int i = 3;

  if (f == i) //comparison of a float and an int
    ++i;

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int main(void)
{
  int i = 60;
  char c = 60;

  if (i == c)
    ++i;

  return 0;
}
```

MISRAC++2008-6-2-3

Synopsis

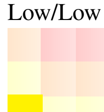
Stray semicolons on the same line as other code

Enabled by default

Yes

Severity/Certainty

Low/Low



Full description

(Required) Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character. by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character.

Coding standards	CERT EXP15-C Do not place a semicolon on the same line as an if, for, or while statement MISRA C++ 2008 6-2-3 (Required) Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int i;
  for (i=0; i!=10; ++i);  //Null statement as the
                          //body of this for loop
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int i;
  for (i=0; i!=10; ++i){  //An empty block is much
  }                       //more readable
}
```

MISRAC++2008-6-3-1_a

Synopsis

Missing braces in do ... while statements

Enabled by default

Yes

Severity/Certainty

Low/Low

Full description

(Required) The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.

Coding standards

CERT EXP19-C

Use braces for the body of an if, for, or while statement

CWE 483

Incorrect Block Delimitation

MISRA C++ 2008 6-3-1

(Required) The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.

Code examples

The following code example fails the check and will give a warning:

```
int example(void) {
  do
    return 0;
  while (1);
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(void) {
  do {
    return 0;
  } while (1);
}
```

MISRAC++2008-6-3-1_b

Synopsis

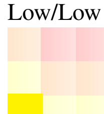
Missing braces in for statements

Enabled by default

Yes

Severity/Certainty

Low/Low



Full description

(Required) The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.

Coding standards

CERT EXP19-C

Use braces for the body of an if, for, or while statement

CWE 483

Incorrect Block Delimitation

MISRA C++ 2008 6-3-1

(Required) The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.

Code examples

The following code example fails the check and will give a warning:

```
int example(void) {
  for (;;)
    return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(void) {
  for (;;){
    return 0;
  }
}
```

MISRAC++2008-6-3-1_c

Synopsis

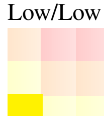
Missing braces in switch statements

Enabled by default

Yes

Severity/Certainty

Low/Low



Full description

(Required) The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.

Coding standards

CERT EXP19-C

Use braces for the body of an if, for, or while statement

CWE 483

Incorrect Block Delimitation

MISRA C++ 2008 6-3-1

(Required) The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  while(1);
  for(;;);
  do ;
  while (0);
  switch(0);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  while(1) {
  }
  for(;;) {
  }
  do {
  } while (0);
  switch(0) {
  }
}
```

MISRAC++2008-6-3-1_d

Synopsis

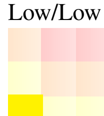
Missing braces in while statements

Enabled by default

Yes

Severity/Certainty

Low/Low



Full description

(Required) The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.

Coding standards

CERT EXP19-C

Use braces for the body of an if, for, or while statement

CWE 483

Incorrect Block Delimitation

MISRA C++ 2008 6-3-1

(Required) The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.

Code examples

The following code example fails the check and will give a warning:

```
int example(void) {
  while (1)
    return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(void) {
  while (1){
    return 0;
  }
}
```

MISRAC++2008-6-4-1

Synopsis

Missing braces in if, else, and else if statements

Enabled by default

Yes

Severity/Certainty

Low/Low

Full description

(Required) An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.

Coding standards

CERT EXP19-C

Use braces for the body of an if, for, or while statement

CWE 483

Incorrect Block Delimitation

MISRA C++ 2008 6-4-1

(Required) An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.

Code examples

The following code example fails the check and will give a warning:

```
#include "iar.h"

void example(void) {
  if (random());
  if (random());
  else;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include "iar.h"

void example(void) {
  if (random()) {
  }
  if (random()) {
  } else {
  }
  if (random()) {
  } else if (random()) {
  }
}
```

MISRAC++2008-6-4-2

Synopsis

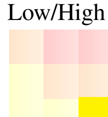
If ... else if constructs that are not terminated with an else clause.

Enabled by default

Yes

Severity/Certainty

Low/High



Full description

(Required) All if ... else if constructs shall be terminated with an else clause.

Coding standards

MISRA C++ 2008 6-4-2

(Required) All if ... else if constructs shall be terminated with an else clause.

Code examples

The following code example fails the check and will give a warning:

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

void example(void) {
  if (!rand()) {
    printf("The first random number is 0");
  } else if (!rand()) {
    printf("The second random number is 0");
  }
}
```

The following code example passes the check and will not give a warning about this issue:

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

void example(void) {
  if (!rand()) {
    printf("The first random number is 0");
  } else if (!rand()) {
    printf("The second random number is 0");
  } else {
    printf("Neither random number was 0");
  }
}
```

MISRAC++2008-6-4-3

Synopsis

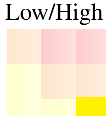
Switch statements that do not conform to the MISRA C switch syntax.

Enabled by default

Yes

Severity/Certainty

Low/High



Full description

(Required) A switch statement shall be a well-formed switch statement. switch-statement : switch '(' expression ')' '{' case-label-clause-list default-label-clause? '}' case-label-clause-list: case-label case-clause? case-label-clause-list case-label case-clause? case-label: case constant-expression ':' case-clause: statement-list? break ';' '{' declaration-list? statement-list? break ';' '}' default-label-clause : default-label default-clause default-label: default ':' default-clause: case-clause

Coding standards MISRA C++ 2008 6-4-3

(Required) A switch statement shall be a well-formed switch statement.

Code examples The following code example fails the check and will give a warning:

```
int expr();
void stmt();
void example(void) {
  switch(expr()) {
    // at least one case label
    case 1:
      // statement list
      stmt();
      stmt();
      // WARNING: missing break at end of statement list
    default:
      break; // statement list ends in a break
  }

  switch(expr()) {
    // WARNING: missing at least one case label
    default:
      break; // statement list ends in a break
  }

  switch(expr()) {
    // at least one case label
    case 1:
      // statement list
      stmt();
      stmt();
      break; // statement list ends in a break
    case 0:
      stmt();
      // WARNING: declaration list without block
      int decl = 0;
      int x;
      // statement list
      stmt();
      stmt();
      break; // statement list ends in a break
    default:
      break; // statement list ends in a break
  }

  switch(expr()) {
    // at least one case label
    case 1: {
      // statement list
      stmt();
      // WARNING: Additional block inside of the case clause
block
```

```
        {
        stmt();
        }
        break;
      }
      default:
        break; // statement list ends in a break
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
int expr();
void stmt();
void example(void) {
  switch(expr()) {
    // at least one case label
    case 1:
      // statement list (no declarations)
      stmt();
      stmt();
      break; // statement list ends in a break
    case 0: {
      // one level of block is allowed
      // declaration list
      int decl = 0;
      // statement list
      stmt();
      stmt();
      break; // statement list ends in a break
    }
    case 2: // empty cases are allowed
    default:
      break; // statement list ends in a break
  }
}
```

MISRAC++2008-6-4-4

Synopsis	Switch labels in nested blocks.
Enabled by default	Yes

Severity/Certainty

Low/Medium

Full description

(Required) A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.

Coding standards

MISRA C++ 2008 6-4-4

> (Required) A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

void example(void) {

   switch(rand()) {
      {case 1:}
      case 2:
      case 3:
      default:
   }

}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

void example(void) {

   switch(rand()) {
      case 1:
      case 2:
      case 3:
      default:
   }

}
```

MISRAC++2008-6-4-5

Synopsis

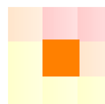
Non-empty switch cases not terminated by break

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) An unconditional throw or break statement shall terminate every non-empty switch-clause.

Coding standards

CERT MSC17-C

> Finish every set of statements associated with a case label with a break statement

CWE 484

> Omitted Break Statement in Switch

MISRA C++ 2008 6-4-5

> (Required) An unconditional throw or break statement shall terminate every non-empty switch-clause.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

void example(int input) {

  while (rand()) {
    switch(input) {
      case 0:
        if (rand()) {
          break;
        }
      default:
        break;
    }
  }
}
#include <stdlib.h>

void example(int input) {

  switch(input) {
    case 0:
      if (rand()) {
        break;
      }
    default:
      break;
  }

}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

void example(int input) {

  switch(input) {
    case 0:
      if (rand()) {
        break;
      }
      break;
    default:
      break;
  }

}
#include <stdlib.h>

void example(int input) {

  switch(input) {
    case 0:
      if (rand()) {
        break;
      } else {
        break;
      }
      // All paths above contain a break, therefore we do not
warn
    default:
      break;
  }

}
```

MISRAC++2008-6-4-6

Synopsis

Switch statements with no default clause, or a default clause that is not the final clause.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description (Required) The final clause of a switch statement shall be the default-clause.

Coding standards CWE 478

Missing Default Case in Switch Statement

MISRA C++ 2008 6-4-6

(Required) The final clause of a switch statement shall be the default-clause.

Code examples The following code example fails the check and will give a warning:

```
int example(int x) {
  switch(x){
    default:
      return 2;
      break;
    case 0:
      return 0;
      break;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(int x) {
  switch(x){
    case 3:
      return 0;
      break;
    case 5:
      return 1;
      break;
    default:
      return 2;
      break;
  }
}
```

MISRAC++2008-6-4-7

Synopsis A switch expression shall not represent a value that is effectively boolean.

Enabled by default Yes

Severity/Certainty

Low/Medium

Full description

(Required) The condition of a switch statement shall not have bool type.

Coding standards

MISRA C++ 2008 6-4-7

(Required) The condition of a switch statement shall not have bool type.

Code examples

The following code example fails the check and will give a warning:

```
void example(int x) {
  switch(x == 0) {
    case 0:
    case 1:
    default:
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(int x) {
  switch(x) {
    case 1:
    case 0:
    default:
  }
}
```

MISRAC++2008-6-4-8

Synopsis

Switch statements with no cases.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Every switch statement shall have at least one case-clause.

Coding standards

MISRA C++ 2008 6-4-8

(Required) Every switch statement shall have at least one case-clause.

Code examples

The following code example fails the check and will give a warning:

```
int example(int x) {
  switch(x){
    default:
      return 2;
      break;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(int x) {
  switch(x){
    case 3:
      return 0;
      break;
    case 5:
      return 1;
      break;
    default:
      return 2;
      break;
  }
}
```

MISRAC++2008-6-5-1_a

Synopsis

Floating-point values in the controlling expression of a for statement.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) A for loop shall contain a single loop-counter which shall not have floating type.

Coding standards

MISRA C++ 2008 6-5-1

> (Required) A for loop shall contain a single loop-counter which shall not have floating type.

Code examples

The following code example fails the check and will give a warning:

```
void example(int input, float f) {
  int i;
  for (i = 0; i < input && f < 0.1f; ++i) {
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(int input, float f) {
  int i;
  int f_condition = f < 0.1f;
  for (i = 0; i < input && f_condition; ++i) {
    f_condition = f < 0.1f;
  }
}
```

MISRAC++2008-6-5-2

Synopsis

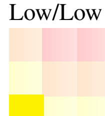
Loop counter may not match loop condition test.

Enabled by default

Yes

Severity/Certainty

Low/Low



Full description

(Required) If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.

Coding standards

CERT MSC21-C

> Use robust loop termination conditions

CERT MSC21-CPP

> Use inequality to terminate a loop whose counter changes by more than one

MISRA C++ 2008 6-5-2

(Required) If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.

Code examples

The following code example fails the check and will give a warning:

```
void example(void)
{
  for(int i = 0; i != 10; i += 2) {}
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void)
{
  for(int i = 0; i != 10; i++) {}
}
void example(void)
{
  for(int i = 0; i <= 10; i+= 2) {}
}
```

MISRAC++2008-6-5-3

Synopsis

A `for` loop counter variable is modified in the body of the loop.

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

(Required) The loop-counter shall not be modified within condition or statement. statement) should not be assigned to in the body of the for loop. While it's legal to modify the loop counter within the body of a `for` loop (in place of a `while` loop), the conventional use of a `for` loop is to iterate over a predetermined range, incrementing the loop counter once per iteration. Modification of the loop counter within the `for` loop body is probably accidental, and could result in erroneous behavior or an infinite loop.

Coding standards

MISRA C++ 2008 6-5-3

(Required) The loop-counter shall not be modified within condition or statement.

Code examples

The following code example fails the check and will give a warning:

```
int main(void) {
  int i;

  /* i is incremented inside the loop body */
  for (i = 0; i < 10; i++) {
    i = i + 1;
  }

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int main(void) {
  int i;
  int x = 0;

  for (i = 0; i < 10; i++) {
    x = i + 1;
  }

  return 0;
}
```

MISRAC++2008-6-5-4

Synopsis

Potential inconsistent loop counter modification.

Enabled by default

Yes

Severity/Certainty

Low/Low

Full description

(Required) The loop-counter shall be modified by one of: --, ++, -=n, or +=n; where n remains constant for the duration of the loop.

Coding standards

MISRA C++ 2008 6-5-4

> (Required) The loop-counter shall be modified by one of: --, ++, -=n, or +=n; where n remains constant for the duration of the loop.

Code examples

The following code example fails the check and will give a warning:

```
void example(void)
{
  int i;
  for(i = 0; i != 10; i= i * i) {}
}
int func(int x)
{
  return x + 1;
}

void example(void)
{
  for(int i = 0; i != 10; i+= func(i)) {}
}
```

The following code example passes the check and will not give a warning about this issue:

```
int func()
{
  return 1;
}

void example(void)
{
  for(int i = 0; i != 10; i+= func()) {}
}
void example(void)
{
  bool b;
  for(int i = 0; i != 10 || b; i-=2) {}
}
```

MISRAC++2008-6-5-6

Synopsis

A non-boolean variable is modified in the loop and used as loop condition.

Enabled by default

Yes

Severity/Certainty

Low/Low

Full description

(Required) A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.

Coding standards

MISRA C++ 2008 6-5-6

> (Required) A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.

Code examples

The following code example fails the check and will give a warning:

```
void example(void)
{
  int j;
  for (int i = 0; i < 10 || j > 5; ++i)
  {
    j = i;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void)
{
  bool found = false;
  for (int i = 0; i < 10 || found; ++i)
  {
    found = (i + 1) % 9;
  }
}
```

MISRAC++2008-6-6-1

Synopsis

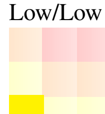
The target of the goto is a nested code block.

Enabled by default

Yes

Severity/Certainty

Low/Low



Full description

(Required) Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.

Coding standards

MISRA C++ 2008 6-6-1

(Required) Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.

Code examples

The following code example fails the check and will give a warning:

```
void f1 ( )
{
  int j = 0;
  goto L1;
  for (;;)
  {
L1: // Non-compliant
    j;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void f2()
{
  for(;;)
  {
    for(;;)
    {
      goto L1;
    }
  }
L1:
  return;
}
```

MISRAC++2008-6-6-2

Synopsis

Goto declared after target label.

Enabled by default

Yes

Severity/Certainty

Low/Low

Full description

(Required) The goto statement shall jump to a label declared later in the same function body.

Coding standards

MISRA C++ 2008 6-6-2

(Required) The goto statement shall jump to a label declared later in the same function body.

Code examples

The following code example fails the check and will give a warning:

```
void f1 ( )
{
  int j = 0;
  for ( j = 0; j < 10 ; ++j )
  {
L1: // Non-compliant
    j;
  }
  goto L1;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void f1 ( )
{
  int j = 0;
  goto L1;
  for ( j = 0; j < 10 ; ++j )
  {
    j;
  }
L1:
  return;
}
```

MISRAC++2008-6-6-4

Synopsis

Multiple break points from loop.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) For any iteration statement there shall be no more than one break or goto statement used for loop termination.

Coding standards

MISRA C++ 2008 6-6-4

(Required) For any iteration statement there shall be no more than one break or goto statement used for loop termination.

Code examples

The following code example fails the check and will give a warning:

```
void func()
{
  int x = 1;
  for ( int i = 0; i < 10; i++ )
  {
    if ( x )
    {
      break;
    }
    else if ( i )
    {
      break;  // Non-compliant - second jump from loop
    }
    else
    {
      // Code
    }
  }
}
int fn(void);

void example(void) {
   int i = fn();
   int j;
   int counter = 0;
   switch (i) {
      case 1:
         break;
      case 2:
      case 3:
         counter++;
         if (i==3) {
               break;
         }
         counter++;
         break;
      case 4:
         for (j = 0; j < 10; j++) {
               if (j == i) {
                     break;
               }
               if (j == counter) {
                     break;
               }
         }
         counter--;
         break;
```

```
        default:
          break;
    }
}
int fn(int i);

void example(void) {
    int counter = 0;
    int i = 0;
    for (i = 0; i < 100; i++) {
        switch (i % 9) {
          case 8:
                counter++;
                break;
          default:
                break;
        }
        if (fn(i)) {
          break;
        }
        if (fn(i)) {
          break;
        }
    }
}

int test1(int);
int test2(int);

void example(void)
{
    int i = 0;
    for (i = 0; i < 10; i++) {
        if (test1(i)) {
          break;
        } else if (test2(i)) {
          break;
        }
    }
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void)
{
  int i = 0;
  for (i = 0; i < 10 && i != 9; i++) {
    if (i == 9) {
      break;
    }
  }
}
void func()
{
  int x = 1;
  for ( int i = 0; i < 10; i++ )
  {
    if ( x )
    {
      break;
    }
    else if ( i )
    {
      while ( true )
      {
        if ( x )
        {
          break;
        }
        do
        {
          break;
        }
        while(true);
      }
    }
    else
    {
    }
  }
}
int fn(void);

void example(void) {
  int i = fn();
  int j;
  int counter = 0;
  switch (i) {
    case 1:
      break;
```

```
      case 2:
      case 3:
        counter++;
        if (i==3) {
            break;
        }
        counter++;
        break;
      case 4:
        for (j = 0; j < 10; j++) {
            if (j == i) {
                  break;
            }
        }
        counter--;
        break;
      default:
        break;
    }
}
int fn(int i);

void example(void) {
  int counter = 0;
  int i = 0;
  int stop = 0;
  for (i = 0; i < 100 && !stop; i++) {
    switch (i % 9) {
      case 8:
            counter++;
            break;
      default:
            break;
    }
    stop = fn(i);
  }
}
```

MISRAC++2008-6-6-5

Synopsis	A function shall have a single point of exit at the end of the function.
Enabled by default	Yes

Severity/Certainty

Low/Medium

Full description

(Required) A function shall have a single point of exit at the end of the function. function. This is required by IEC 61508, under good programming style.

Coding standards

MISRA C++ 2008 6-6-5

(Required) A function shall have a single point of exit at the end of the function.

Code examples

The following code example fails the check and will give a warning:

```
extern int errno;

void example(void) {
   if (errno) {
      return;
   }
   return;
}
```

The following code example passes the check and will not give a warning about this issue:

```
extern int errno;

void example(void) {
   if (errno) {
      goto end;
   }
end:
   {
      return;
   }
}
```

MISRAC++2008-7-1-1

Synopsis

A local variable is not modified after its initialization and so should be const qualified.

Enabled by default

Yes

Severity/Certainty   Low/Medium

Full description   (Required) A variable which is not modified shall be const qualified.

Coding standards   MISRA C++ 2008 7-1-1

(Required) A variable which is not modified shall be const qualified.

Code examples   The following code example fails the check and will give a warning:

```
int example( void ){
  int x = 7;
  return x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example( void ){
  int x = 7;
  ++x;
  return x;
}
```

MISRAC++2008-7-1-2

Synopsis   A function does not modify one of its parameters.

Enabled by default   Yes

Severity/Certainty   Low/Medium

Full description   (Required) A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.

Coding standards   MISRA C++ 2008 7-1-2

(Required) A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.

Code examples

The following code example fails the check and will give a warning:

```
int example(int* x) {  //x should be const
  if (*x > 5){
    return *x;
  } else {
    return 5;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(const int* x) {  //OK
  if (*x > 5){
    return *x;
  } else {
    return 5;
  }
}
```

MISRAC++2008-7-2-1

Synopsis

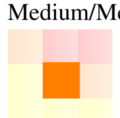
Conversions to enum that are out of range of the enumeration.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.

Coding standards

MISRA C++ 2008 7-2-1

(Required) An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.

Code examples

The following code example fails the check and will give a warning:

```
enum ens { ONE, TWO, THREE };

void example(void)
{
  ens one = (ens)10;
}
enum ens { ONE, TWO, THREE };

int func()
{
  return 10;
}

void example(void)
{
  ens one = (ens)func();
}
```

The following code example passes the check and will not give a warning about this issue:

```
enum ens { ONE, TWO, THREE };

int func()
{
  return 1;
}

void example(void)
{
  ens one = (ens)func();
}
enum ens { ONE, TWO, THREE };

void example(void)
{
  ens one = ONE;
  ens two = TWO;
  two = one;
}
```

MISRAC++2008-7-4-3

Synopsis
Inline asm statements that are not encapsulated in functions

Enabled by default
Yes

Severity/Certainty Low/Medium

Full description (Required) Assembly language shall be encapsulated and isolated.

Coding standards MISRA C++ 2008 7-4-3

(Required) Assembly language shall be encapsulated and isolated.

Code examples The following code example fails the check and will give a warning:

```
int ffs(int x)
{
        int r;
#if 0
#ifdef CONFIG_X86_64
        /*
         * AMD64 says BSFL won't clobber the dest reg if x==0;
Intel64 says the
         * dest reg is undefined if x==0, but their CPU architect
says its
         * value is written to set it to the same as before,
except that the
         * top 32 bits will be cleared.
         *
         * We cannot do this on 32 bits because at the very least
some
         * CPUs did not behave this way.
         */
        long tmp = -1;
        asm("bsfl %1,%0"
            : "=r" (r)
            : "rm" (x), "" (tmp));
#elif defined(CONFIG_X86_CMOV)
        asm("bsfl %1,%0\n\t"
            "cmovzl %2,%0"
            : "=&r" (r) : "rm" (x), "r" (-1));
#else
        asm("bsfl %1,%0\n\t"
            "jnz 1f\n\t"
            "movl $-1,%0\n"
            "1:" : "=r" (r) : "rm" (x));
#endif
#else
        asm("");
#endif
        return r + 1;
}
```

The following code example passes the check and will not give a warning about this issue:

```
unsigned int
bswap(unsigned int x)
{
    asm("");
    return x;
}
```

MISRAC++2008-7-5-1_a (C++ only)

Synopsis

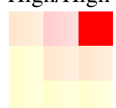
A stack object is returned from a function as a reference.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

(Required) A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function. Operations on the return value are illegal and a program crash, or memory corruption, is very likely. A safe alternative is for the function to return a copy of the object.

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 562

Return of Stack Variable Address

MISRA C++ 2008 7-5-1

(Required) A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.

Code examples

The following code example fails the check and will give a warning:

```
int& example(void) {
  int x;
  return x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(void) {
  int x;
  return x;
}
```

MISRAC++2008-7-5-1_b

Synopsis

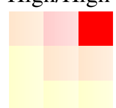
May return address on the stack.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

(Required) A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function. Depending on the circumstances, this code and subsequent memory accesses could appear to work, but the operations are illegal and a program crash, or memory corruption, is very likely. Returning a copy of the object, using a global variable, or dynamically allocating memory, are possible alternatives.

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 562

Return of Stack Variable Address

MISRA C++ 2008 7-5-1

(Required) A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.

Code examples

The following code example fails the check and will give a warning:

```
int *f() {
  int x;
  return &x;  //x is a local variable
}
int *example(void) {
  int a[20];
  return a;  //a is a local array
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

int* example(void) {
  int *p,i;
  p = (int *)malloc(sizeof(int));
  return p;  //OK - p is dynamically allocated

}
```

MISRAC++2008-7-5-2_a

Synopsis

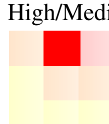
Store a stack address in a global pointer.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist. This is particularly dangerous because the program may appear to work normally, while it is in fact accessing illegal memory. Other results of this are a program crash, or the data changing unpredictably.

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 466

Return of Pointer Value Outside of Expected Range

MISRA C++ 2008 7-5-2

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Code examples

The following code example fails the check and will give a warning:

```
int *px;
void example() {
  int i = 0;
  px = &i; // assigning the address of stack
           // variable a to the global px
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(int *pz) {
  int x; int *px = &x;
  int *py = px; /* local variable */
  pz = px; /* parameter */
}
```

MISRAC++2008-7-5-2_b

Synopsis

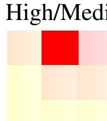
Store a stack address in the field of a global struct.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist. This is particularly dangerous because the program may appear to work normally, while it is in fact accessing illegal memory. Other results of this are a program crash, or the data changing unpredictably.

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 466

Return of Pointer Value Outside of Expected Range

MISRA C++ 2008 7-5-2

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Code examples

The following code example fails the check and will give a warning:

```
struct S{
  int *px;
} s;

void example() {
  int i = 0;
  s.px = &i; //storing local address in global struct
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

struct S{
  int *px;
} s;

void example() {
  int i = 0;
  s.px = &i; //OK - the field is written to later
  s.px = NULL;
}
```

MISRAC++2008-7-5-2_c

Synopsis

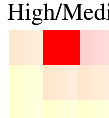
Store stack address outside function via parameter.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist. This is particularly dangerous because the program may appear to work normally, while it is in fact accessing illegal memory. Other results of this are a program crash, or the data changing unpredictably. Known false positives: this test checks for any expression refering to the store located by the parameter and so the assignment 'local[*parameter] = & local;' will invoke a warning.

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 466

Return of Pointer Value Outside of Expected Range

MISRA C++ 2008 7-5-2

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Code examples

The following code example fails the check and will give a warning:

```
void example(int **ppx) {
  int x;
  ppx[0] = &x;  //local address
}
```

The following code example passes the check and will not give a warning about this issue:

```
static int y = 0;
void example3(int **ppx){
  *ppx = &y;  //OK - static address
}
```

MISRAC++2008-7-5-2_d (C++ only)

Synopsis

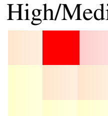
Store stack address via reference parameter.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist. The address of a local stack variable is assigned to a reference argument of its function. When the function ends, this memory address will become invalid. This is particularly dangerous because the program may appear to work normally, while it is in fact accessing illegal memory. Other results of this are a program crash, or the data changing unpredictably.

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 466

Return of Pointer Value Outside of Expected Range

MISRA C++ 2008 7-5-2

(Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Code examples

The following code example fails the check and will give a warning:

```
void example(int *&pxx) {
  int x;
  pxx = &x;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(int *p, int *&q) {
  int x;
  int *px= &x;
  p = px; // ok, pointer
  q = p; // ok, not local
}
```

MISRAC++2008-7-5-4_a

Synopsis

Functions that call themselves directly.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

(Advisory) Functions should not call themselves, either directly or indirectly.

Coding standards

MISRA C++ 2008 7-5-4

(Advisory) Functions should not call themselves, either directly or indirectly.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  example();
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC++2008-7-5-4_b

Synopsis

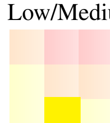
Functions that call themselves indirectly.

Enabled by default

No

Severity/Certainty

Low/Medium



Full description

(Advisory) Functions should not call themselves, either directly or indirectly.

Coding standards

MISRA C++ 2008 7-5-4

(Advisory) Functions should not call themselves, either directly or indirectly.

Code examples

The following code example fails the check and will give a warning:

```
void example(void);
void callee(void) {
    example();
}
void example(void) {
    callee();
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void);
void callee(void) {
    // example();
}
void example(void) {
    callee();
}
```

MISRAC++2008-8-0-1

Synopsis

Declarations shall only contain one variable or constant each.

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

(Required) An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.

Coding standards

MISRA C++ 2008 8-0-1

> (Required) An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.

Code examples

The following code example fails the check and will give a warning:

```
int foo(){
  int a,b,c;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int foo(){
  int a; int b; int c;
}
```

MISRAC++2008-8-4-1

Synopsis

Functions defined using ellipsis (...) notation

Enabled by default Yes

Severity/Certainty Low/High

Full description (Required) Functions shall not be defined using the ellipsis notation. Additionally, passing an argument with non-POD class type leads to undefined behaviour. Note that the rule specifies defined (and not declared) so as to permit the use of existing library functions.

Coding standards MISRA C++ 2008 8-4-1

> (Required) Functions shall not be defined using the ellipsis notation.

Code examples The following code example fails the check and will give a warning:

```
#include <stdarg.h>
int putchar(int c);

void
minprintf(const char *fmt, ...)
{
    va_list    ap;
    const char *p, *s;

    va_start(ap, fmt);
    for (p = fmt; *p != '\0'; p++) {
        if (*p != '%') {
            putchar(*p);
            continue;
        }
        switch (*++p) {
        case 's':
            for (s = va_arg(ap, const char *); *s != '\0'; s++)
                 putchar(*s);
            break;
        }
    }
    va_end(ap);
}
```

The following code example passes the check and will not give a warning about this issue:

```
int puts(const char *);

void
func(void)
{
    puts("Hello, world!");
}
```

MISRAC++2008-8-4-3

Synopsis

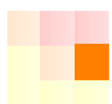
For some execution, no return statement is executed in a function with a non-void return type

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

(Required) All exit paths from a function with non-void return type shall have an explicit return statement with an expression. Checks whether all execution paths in non-void functions contain a return statement before they exit. If a non-void function has no return statement, it will return an undefined value. This will not pose a problem if the function is used as a void function, however, if the function return value is used it will cause unpredictable behavior. Note: This is a weaker check than the one performed by gcc. Its check allows more aggressive coding without violating the rule. However, a rule violation in gcc means there is no path leading to a return statement. non-`void` return type.

Coding standards

CERT MSC37-C

> Ensure that control never reaches the end of a non-void function

MISRA C++ 2008 8-4-3

> (Required) All exit paths from a function with non-void return type shall have an explicit return statement with an expression.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdio.h>

int example(void) {
  int x;

  scanf("%d",&x);

  if (x > 10) {
    return 10;
  }
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdio.h>

int example(void) {
  int x;

  scanf("%d",&x);

  if (x > 10) {
    return 10;
  }

  return 0;
}
```

MISRAC++2008-8-4-4

Synopsis	Function addresses taken without explicit &
Enabled by default	Yes
Severity/Certainty	Low/High
Full description	(Required) A function identifier shall either be used to call the function or it shall be preceded by &.
Coding standards	MISRA C++ 2008 8-4-4

(Required) A function identifier shall either be used to call the function or it shall be preceded by &.

Code examples

The following code example fails the check and will give a warning:

```
void func(void);

void
example(void)
{
    void (*pf)(void) = func;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void func(void);

void
example(void)
{
    void (*pf)(void) = &func;
}
```

MISRAC++2008-8-5-1_a

Synopsis

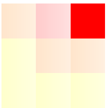
In all executions, a variable is read before it is assigned a value.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

(Required) All variables shall have a defined value before they are used. value. Different paths may result in reading a variable at different program points. Whichever path is executed, uninitialized data is read, and behavior may consequently be unpredictable.

Coding standards

CERT EXP33-C

Do not reference uninitialized memory

CWE 457

Use of Uninitialized Variable

MISRA C++ 2008 8-5-1

(Required) All variables shall have a defined value before they are used.

Code examples

The following code example fails the check and will give a warning:

```
int main(void) {
  int x;

  x++;  //x is uninitialized

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int main(void) {
  int x = 0;

  x++;

  return 0;
}
```

MISRAC++2008-8-5-1_b

Synopsis

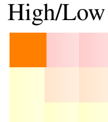
In some execution, a variable is read before it is assigned a value.

Enabled by default

Yes

Severity/Certainty

High/Low



Full description

(Required) All variables shall have a defined value before they are used. There may be some execution paths where the variable is assigned a value before it is read. In such cases behavior may be unpredictable.

Coding standards

CWE 457

Use of Uninitialized Variable

MISRA C++ 2008 8-5-1

(Required) All variables shall have a defined value before they are used.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

int main(void) {
  int x, y;

  if (rand()) {
    x = 0;
  }

  y = x;  //x may not be initialized

  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>

int main(void) {
  int x;

  if (rand()) {
    x = 0;
  }

  /* x never read */

  return 0;
}
```

MISRAC++2008-8-5-1_c

Synopsis

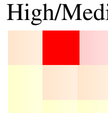
Dereference of an uninitialized or NULL pointer.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

(Required) All variables shall have a defined value before they are used. This will likely result in memory corruption or a program crash. Pointer values should always be initialized before being dereferenced, to avoid this.

Coding standards

CERT EXP33-C

Do not reference uninitialized memory

CWE 457

Use of Uninitialized Variable

CWE 824

Access of Uninitialized Pointer

MISRA C++ 2008 8-5-1

(Required) All variables shall have a defined value before they are used.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int *p;
  *p = 4;  //p is uninitialized
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int *p,a;
  p = &a;
  *p = 4;  //OK - p holds a valid address
}
```

MISRAC++2008-8-5-2

Synopsis

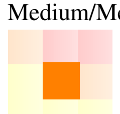
This check points out where a non-zero array initialisation does not exactly match the structure of the array declaration.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures.

Coding standards

MISRA C++ 2008 8-5-2

(Required) Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures.

Code examples

The following code example fails the check and will give a warning:

```
void example(void) {
  int y[3][4] = { { 1, 2, 3 }, { 4, 5, 6 } };
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
  int y[3][2] = { { 1, 2 }, { 3, 4 }, { 5, 6 } };
}
```

MISRAC++2008-9-3-1 (C++ only)

Synopsis

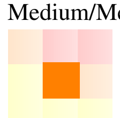
A member function qualified as `const` returns a pointer member variable.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) const member functions shall not return non-const pointers or references to class-data. This will not be identified by a compiler as the pointer being returned is a copy, even though the memory to which it refers is vulnerable.

Coding standards

MISRA C++ 2008 9-3-1

(Required) const member functions shall not return non-const pointers or references to class-data.

Code examples

The following code example fails the check and will give a warning:

```
class C{
  int* foo() const {
    return p;
  }
  int* p;
};
```

The following code example passes the check and will not give a warning about this issue:

```
class C{
  int* foo() {
    return p;
  }
  int* p;
};
```

MISRAC++2008-9-3-2 (C++ only)

Synopsis

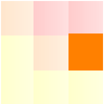
Member functions that return non-const handles to members

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

(Required) Member functions shall not return non-const handles to class-data. over how the object state can be modified and helps to allow a class to be maintained without affecting clients. Returning a handle to class-data allows for clients to modify the state of the object without using any interfaces.

Coding standards

CERT OOP35-CPP

Do not return references to private data

MISRA C++ 2008 9-3-2

(Required) Member functions shall not return non-const handles to class-data.

Code examples

The following code example fails the check and will give a warning:

```
class C{
  int x;
 public:
  int& foo();
  int* bar();
};

int& C::foo() {
  return x;  //returns a non-const reference to x
}

int* C::bar() {
  return &x;  //returns a non-const pointer to x
}
```

The following code example passes the check and will not give a warning about this issue:

```
class C{
  int x;
 public:
  const int& foo();
  const int* bar();
};

const int& C::foo() {
  return x;  //OK - returns a const reference
}

const int* C::bar() {
  return &x;  //OK - returns a const pointer
}
```

MISRAC++2008-9-5-1

Synopsis	All unions
Enabled by default	Yes
Severity/Certainty	Low/Medium
Full description	(Required) Unions shall not be used.

Coding standards

MISRA C++ 2008 9-5-1

(Required) Unions shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
union cheat {
  int   i;
  float f;
};

int example(float f) {
  union cheat u;
  u.f = f;
  return u.i;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int example(int x) {
  return x;
}
```

MISRAC++2008-9-6-2

Synopsis

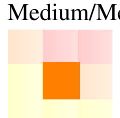
Bitfields with plain int type

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Bit-fields shall be either bool type or an explicitly unsigned or signed integral type.

Coding standards

MISRA C++ 2008 9-6-2

(Required) Bit-fields shall be either bool type or an explicitly unsigned or signed integral type.

Code examples

The following code example fails the check and will give a warning:

```
struct bad {
   int x:3;
};
#error "IGNORE_TEST: enum's are ok!"

enum digs { ONE, TWO, THREE, FOUR };

struct bad {
   digs d:3;
};
```

The following code example passes the check and will not give a warning about this issue:

```
struct good {
   signed int x:3;
};
struct good {
   unsigned int x:3;
};
```

MISRAC++2008-9-6-3

Synopsis

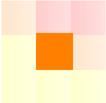
Bitfields with plain int type

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Bit-fields shall not have enum type.

Coding standards

MISRA C++ 2008 9-6-3

(Required) Bit-fields shall not have enum type.

Code examples

The following code example fails the check and will give a warning:

```
enum digs { ONE, TWO, THREE, FOUR };

struct bad {
   digs d:3;
};
```

The following code example passes the check and will not give a warning about this issue:

```
struct good {
  signed int x:3;
};
struct good {
  unsigned int x:3;
};
```

MISRAC++2008-9-6-4

Synopsis

Signed single-bit fields (excluding anonymous fields)

Enabled by default

Yes

Severity/Certainty

Low/Low

Full description

(Required) Named bit-fields with signed integer type shall have a length of more than one bit.

Coding standards

MISRA C++ 2008 9-6-4

> (Required) Named bit-fields with signed integer type shall have a length of more than one bit.

Code examples

The following code example fails the check and will give a warning:

```
struct S
{
  signed int a : 1; // Non-compliant
};
```

The following code example passes the check and will not give a warning about this issue:

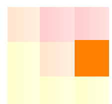
```
struct S
{
  signed int b : 2;
  signed int   : 0;
  signed int   : 1;
  signed int   : 2;
};
```

MISRAC++2008-12-1-1_a (C++ only)

Synopsis
A virtual member function is called in a class constructor.

Enabled by default
Yes

Severity/Certainty
Medium/High



Full description
(Required) An object's dynamic type shall not be used from the body of its constructor or destructor. When an instance is constructed, the virtual member function of its base class is called, rather than the function of the actual class being constructed. This might result in an incorrect function being called, and consequently erroneous data or uninitialized elements.

Coding standards
CERT OOP30-CPP

Do not invoke virtual functions from constructors or destructors

MISRA C++ 2008 12-1-1

(Required) An object's dynamic type shall not be used from the body of its constructor or destructor.

Code examples
The following code example fails the check and will give a warning:

```
#include <iostream>
#ifndef __embedded_cplusplus
  using namespace std;
#endif

class A {
public:
  A() { f(); }  //virtual member function is called
  virtual void f() const { cout << "A::f\n"; }
};

class B: public A {
public:
  virtual void f() const { cout << "B::f\n"; }
};

int main(void) {
  B *b = new B();
  delete b;
  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <iostream>
#ifndef __embedded_cplusplus
  using namespace std;
#endif

class A {
public:
  A() { }  //OK - contructor does not call any virtual
           //member functions
  virtual void f() const { cout << "A::f\n"; }
};

class B: public A {
public:
  virtual void f() const { cout << "B::f\n"; }
};

int main(void) {
  B *b = new B();
  delete b;
  return 0;
}
```

MISRAC++2008-12-1-1_b (C++ only)

Synopsis

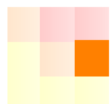
A virtual member function is called in a class destructor.

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

(Required) An object's dynamic type shall not be used from the body of its constructor or destructor. When an instance is destructed, the virtual member function of its base class is called, rather than the function of the actual class being destructed. This might result in an incorrect function being called, and consequently dynamic memory might not be properly deallocated, or some other unwanted behavior may occur.

Coding standards

CERT OOP30-CPP

> Do not invoke virtual functions from constructors or destructors

MISRA C++ 2008 12-1-1

> (Required) An object's dynamic type shall not be used from the body of its constructor or destructor.

Code examples

The following code example fails the check and will give a warning:

```
#include <iostream>
#ifndef __embedded_cplusplus
  using namespace std;
#endif

class A {
public:
  ~A() { f(); }  //virtual member function is called
  virtual void f() const { cout << "A::f\n"; }
};

class B: public A {
public:
  virtual void f() const { cout << "B::f\n"; }
};

int main(void) {
  B *b = new B();
  delete b;
  return 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <iostream>
#ifndef __embedded_cplusplus
  using namespace std;
#endif

class A {
public:
  ~A() { }  //OK - contructor does not call any virtual
            //member functions
  virtual void f() const { cout << "A::f\n"; }
};

class B: public A {
public:
  virtual void f() const { cout << "B::f\n"; }
};

int main(void) {
  B *b = new B();
  delete b;
  return 0;
}
```

MISRAC++2008-12-1-3 (C++ only)

Synopsis

All constructors that are callable with a single argument of fundamental type shall be declared `explicit`.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) All constructors that are callable with a single argument of fundamental type shall be declared explicit. fundamental type to the class type.

Coding standards

CERT OOP32-CPP

Ensure that single-argument constructors are marked "explicit"

MISRA C++ 2008 12-1-3

(Required) All constructors that are callable with a single argument of fundamental type shall be declared explicit.

Code examples

The following code example fails the check and will give a warning:

```
class C{
  C(double x){} //should be explicit
};
```

The following code example passes the check and will not give a warning about this issue:

```
class C{
  explicit C(double x){} //OK
};
```

MISRAC++2008-15-0-2

Synopsis

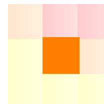
Throw of exceptions by pointer

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

(Advisory) An exception object should not have pointer type. object, then it may be unclear which function is responsible for destroying it, and when. This ambiguity does not exist if the object is caught by value or reference.

Coding standards

CERT ERR09-CPP

Throw anonymous temporaries and catch by reference

MISRA C++ 2008 15-0-2

(Advisory) An exception object should not have pointer type.

Code examples

The following code example fails the check and will give a warning:

```
class Except {};

Except *new_except();

void example(void)
{
    throw new Except();
}
```

The following code example passes the check and will not give a warning about this issue:

```
class Except {};

void example(void)
{
    throw Except();
}
```

MISRAC++2008-15-1-2

Synopsis

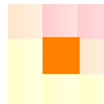
Throw of NULL integer constant

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) NULL shall not be thrown explicitly. only ever caught by an integer handler. This may be inconsistent with developer expectations, particularly if the program only has handlers for pointer-to-type exceptions.

Coding standards

MISRA C++ 2008 15-1-2

(Required) NULL shall not be thrown explicitly.

Code examples

The following code example fails the check and will give a warning:

```
typedef intint32_t;
typedefsigned charchar_t;
#defineNULL0

void example(void)
{
  try {
    throw ( NULL );           // Non-compliant
  }
  catch ( int32_t i ) {       // NULL exception handled here
    // ...
  }
  catch ( const char_t * ) { // Developer may expect it to be
caught here
    // ...
  }
}
```

The following code example passes the check and will not give a warning about this issue:

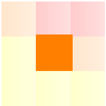
```
typedef intint32_t;
typedefsigned charchar_t;
#defineNULL0

void example(void)
{
  char_t * p = NULL;
  try {
    throw ( p );              // Compliant
  }
  catch ( int32_t i ) {
    // ...
  }
  catch ( const char_t * ) { // Exception handled here
    // ...
  }
}
```

MISRAC++2008-15-1-3 (C++ only)

Synopsis
Unsafe rethrow of exception.

Enabled by default
Yes

Severity/Certainty
Medium/Medium



Full description
(Required) An empty throw (throw;) shall only be used in the compound-statement of a catch handler.

Coding standards
MISRA C++ 2008 15-1-3

(Required) An empty throw (throw;) shall only be used in the compound-statement of a catch handler.

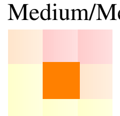
Code examples
The following code example fails the check and will give a warning:

```
void func()
{
  try
  {
    throw;
  }
  catch (...) {}
}
```

The following code example passes the check and will not give a warning about this issue:

```
void func()
{
  try
  {
    throw (42);
  }
  catch (int i)
  {
    if (i > 10)
    {
      throw;
    }
  }
}
```

MISRAC++2008-15-3-1 (C++ only)

Synopsis	Exceptions thrown without a handler in some call paths leading to that point
Enabled by default	Yes
Severity/Certainty	Medium/Medium 
Full description	(Required) Exceptions shall be raised only after start-up and before termination of the program. In particular, it is implementation-defined whether the call stack is unwound before termination, so the destructors of any automatic objects may or may not be invoked. If an exception is thrown as an object of a derived class, a compatible type may be either the derived class or any of its bases. The objective of this rule is that a program should catch all exceptions that it is expected to throw.

Coding standards

MISRA C++ 2008 15-3-1

(Required) Exceptions shall be raised only after start-up and before termination of the program.

Code examples

The following code example fails the check and will give a warning:

```
class C {
public:
    C ( ) { throw ( 0 ); }  // Non-compliant - thrown before main
starts
    ~C ( ) { throw ( 0 ); } // Non-compliant - thrown after main
exits
};

C c;  // An exception thrown in C's constructor or destructor
will
      // cause the program to terminate, and will not be caught
by
      // the handler in main

int main( ... )
{
    try {
        // program code
        return 0;
    }
    // The following catch-all exception handler can only
    // catch exceptions thrown in the above program code
    catch ( ... ) {
        // Handle exception
        return 0;
    }
}
```

The following code example passes the check and will not give a warning about this issue:

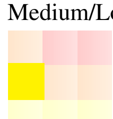
```
class C {
public:
    C ( ) {   }  // Compliant - doesn't throw exceptions
    ~C ( ) {   } // Compliant - doesn't throw exceptions
};

C c;

int main( ... )
{
    try {
        // program code
        return 0;
    }
    // The following catch-all exception handler can only
    // catch exceptions thrown in the above program code
    catch ( ... ) {
        // Handle exception
        return 0;
    }
}
```

MISRAC++2008-15-3-2 (C++ only)

Synopsis
No default exception handler for try.

Enabled by default
No

Severity/Certainty
Medium/Low



Full description
(Advisory) There should be at least one exception handler to catch all otherwise unhandled exceptions

Coding standards
MISRA C++ 2008 15-3-2

(Advisory) There should be at least one exception handler to catch all otherwise unhandled exceptions

Code examples
The following code example fails the check and will give a warning:

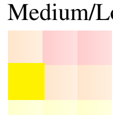
```
int main()
{
  try
  {
    throw (42);
  }
  catch (int i)
  {
    if (i > 10)
    {
      throw;
    }
  }
  return 1;
}
```

The following code example passes the check and will not give a warning about this issue:

```
int main()
{
  try
  {
    throw;
  }
  catch (...) {}
  // spacer
  try {}
  catch (int i) {}
  catch (...) {}
  return 0;
}
```

MISRAC++2008-15-3-3 (C++ only)

Synopsis	Exception handler in constructor or destructor accesses non-static member variable that may not exist.
Enabled by default	Yes
Severity/Certainty	Medium/Low



Full description

(Required) Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.

Coding standards

MISRA C++ 2008 15-3-3

> (Required) Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.

Code examples

The following code example fails the check and will give a warning:

```
int throws();

class C
{
public:
  int x;
  static char c;
  C ( )
  {
    x = 0;
  }

  ~C ( )
  {
    try
    {
      throws();
      // Action that may raise an exception
    }
    catch ( ... )
    {
      if ( 0 == x ) // Non-compliant - x may not exist at this
point
      {
        // Action dependent on value of x
      }
    }
  }
};
```

The following code example passes the check and will not give a warning about this issue:

```
class C
{
public:
  int x;
  static char c;
  C ( )
  {
    try
    {
      // Action that may raise an exception
    }
    catch ( ... )
    {
      if ( 0 == c )
      {
        // Action dependent on value of c
      }
    }
  }

  ~C ( )
  {
    try
    {
      // Action that may raise an exception
    }
    catch (int i) {}
    catch ( ... )
    {
      if ( 0 == c )
      {
        // Action dependent on value of c
      }
    }
  }
};
```

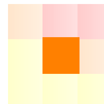
MISRAC++2008-15-3-4 (C++ only)

Synopsis | Calls to functions explicitly declared to throw an exception type that is not handled (or declared as thrown) by the caller

Enabled by default | Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point. In particular, it is implementation-defined whether the call stack is unwound before termination, so the destructors of any automatic objects may or may not be invoked. If an exception is thrown as an object of a derived class, a compatible type may be either the derived class or any of its bases. The objective of this rule is that a program should catch all exceptions that it is expected to throw.

Coding standards

MISRA C++ 2008 15-3-4

> (Required) Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.

Code examples

The following code example fails the check and will give a warning:

```
class E1{};

void foo(int i) throw (E1) {
  if (i<0)
    throw E1();
}

int bar() {
  foo(-3);
}
class E1{};

void foo(int i) throw (E1) {
  if (i<0)
    throw E1();
}

int bar() throw (E1) {  //warning about E1 because it is not 
EXPLICITLY caught
  foo(-3);
}
```

The following code example passes the check and will not give a warning about this issue:

```
class E1{};

void foo(int i) throw (E1) {
  if (i<0)
    throw E1();
}

int bar() {
  try {
    foo(-3);
  }
  catch (E1){
  }
}
```

MISRAC++2008-15-3-5 (C++ only)

Synopsis

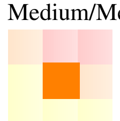
Catch of exception objects by value

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) A class type exception shall always be caught by reference. a derived class and is caught as the base, only the base class' functions (including virtual functions) can be called. Also, any additional member data in the derived class cannot be accessed. If the exception is caught by reference, slicing does not occur.

Coding standards

CERT ERR09-CPP

Throw anonymous temporaries and catch by reference

MISRA C++ 2008 15-3-5

(Required) A class type exception shall always be caught by reference.

Code examples

The following code example fails the check and will give a warning:

```
typedefcharchar_t;

// base class for exceptions
class ExpBase {
public:
    virtual const char_t *who ( ) { return "base"; }
};

class ExpD1: public ExpBase {
public:
    virtual const char_t *who ( ) { return "type 1 exception"; }
};

class ExpD2: public ExpBase {
public:
    virtual const char_t *who ( ) { return "type 2 exception"; }
};

void example()
{
    try {
        // ...
        throw ExpD1 ( );
        // ...
        throw ExpBase ( );
    }
    catch ( ExpBase b ) { // Non-compliant - derived type objects 
will be
                          // caught as the base type
        b.who();          // Will always be "base"
        throw b;          // The exception re-thrown is of the 
base class,
                          // not the original exception type
    }
}
```

The following code example passes the check and will not give a warning about this issue:

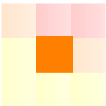
```
typedefcharchar_t;

// base class for exceptions
class ExpBase {
public:
    virtual const char_t *who ( ) { return "base"; }
};

class ExpD1: public ExpBase {
public:
    virtual const char_t *who ( ) { return "type 1 exception"; }
};

class ExpD2: public ExpBase {
public:
    virtual const char_t *who ( ) { return "type 2 exception"; }
};

void example()
{
    try {
        // ...
        throw ExpD1 ( );
        // ...
        throw ExpBase ( );
    }
    catch ( ExpBase &b ) { // Compliant - exceptions caught by
reference
        // ...
        b.who(); // "base", "type 1 exception" or "type 2
exception"
                 // depending upon the type of the thrown object
    }
}
```

MISRAC++2008-15-5-1 (C++ only)

Synopsis An exception is thrown, or may be thrown, in a class' destructor.

Enabled by default Yes

Severity/Certainty Medium/Medium



Full description

(Required) A class destructor shall not exit with an exception.

Coding standards

CERT ERR33-CPP

Destructors must not throw exceptions

MISRA C++ 2008 15-5-1

(Required) A class destructor shall not exit with an exception.

Code examples

The following code example fails the check and will give a warning:

```
class E{};

class C {
  ~C() {
    if (!p){
      throw E();  //may throw an exception here
    }
  }
  int* p;
};
class E{};

void do_something();

class C {
 ~C() throw (E) {  //may throw an exception
    if (!p){
      do_something();
    }
  }
  int* p;
};
```

The following code example passes the check and will not give a warning about this issue:

```
void do_something();

class C {
  ~C() {  //OK
    if (!p){
      do_something();
    }
  }
  int* p;
};
```

MISRAC++2008-16-0-3

Synopsis

All #undef's

Enabled by default

Yes

Severity/Certainty

Low/Low

Full description

(Required) #undef shall not be used. or meaning of a macro when it is used in the code.

Coding standards

MISRA C++ 2008 16-0-3

(Required) #undef shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
#defineSYM
#undef SYM
void example(void) {}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {}
```

MISRAC++2008-16-0-4

Synopsis

Function-like macros

Enabled by default

Yes

Severity/Certainty

Low/Low

Full description

(Required) Function-like macros shall not be defined. robust mechanism. This is particularly true with respect to the type checking of parameters, and the problem of function-like macros potentially evaluating parameters multiple times. Inline functions should be used instead.

Coding standards

MISRA C++ 2008 16-0-4

(Required) Function-like macros shall not be defined.

Code examples

The following code example fails the check and will give a warning:

```
#defineABS(x)((x) < 0 ? -(x) : (x))

void example(void) {
  int a;
  ABS (a);
}
```

The following code example passes the check and will not give a warning about this issue:

```
template <typename T>
inline T ABS(T x) { return x < 0 ? -x : x; }
```

MISRAC++2008-16-2-2 (C++ only)

Synopsis

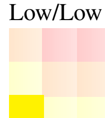
Definition of macros (except include guards)

Enabled by default

Yes

Severity/Certainty

Low/Low



Full description

(Required) C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers. functions and constant declarations.

Coding standards

MISRA C++ 2008 16-2-2

(Required) C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.

Code examples

The following code example fails the check and will give a warning:

```
#defineX(Y)(Y)// Non-compliant
```

The following code example passes the check and will not give a warning about this issue:

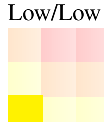
```
#include "header.h"/* contains #ifndef HDR #define HDR ... #endif
*/
void example(void) {}
```

MISRAC++2008-16-2-3

Synopsis: Header files without #include guards

Enabled by default: Yes

Severity/Certainty: Low/Low



Full description: (Required) Include guards shall be provided. particular header file to be included more than once. This can be, at best, a source of confusion. If this multiple inclusion leads to multiple or conflicting definitions, then this can result in undefined or erroneous behaviour.

Coding standards: MISRA C++ 2008 16-2-3

(Required) Include guards shall be provided.

Code examples: The following code example fails the check and will give a warning:

```
#include "unguarded_header.h"
void example(void) {}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <stdlib.h>
#include "header.h"/* contains #ifndef HDR #define HDR ... #endif
*/
void example(void) {}
```

MISRAC++2008-16-2-4

Synopsis: Illegal characters in header file names

Enabled by default: Yes

Severity/Certainty

Low/Low

Full description

(Required) The ', ", /* or // characters shall not occur in a header file name. ', \, /*, or // characters are used between the " delimiters in a header name preprocessing token.

Coding standards

MISRA C++ 2008 16-2-4

(Required) The ', ", /* or // characters shall not occur in a header file name.

Code examples

The following code example fails the check and will give a warning:

```
#include "fi'le.h"/* Non-compliant */
void example(void) {}
```

The following code example passes the check and will not give a warning about this issue:

```
#include "header.h"
void example(void) {}
```

MISRAC++2008-16-2-5

Synopsis

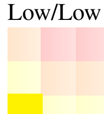
Illegal characters in header file names

Enabled by default

No

Severity/Certainty

Low/Low



Full description

(Advisory) The backslash character should not occur in a header file name. ', \, /*, or // characters are used between the " delimiters in a header name preprocessing token.

Coding standards

MISRA C++ 2008 16-2-5

(Advisory) The backslash character should not occur in a header file name.

Code examples

The following code example fails the check and will give a warning:

```
#include "fi\\le.h"/* Non-compliant */
```

The following code example passes the check and will not give a warning about this issue:

```
#include "header.h"
void example(void) {}
```

MISRAC++2008-16-3-1

Synopsis

Multiple # or ## operators in a macro definition

Enabled by default

Yes

Severity/Certainty

Medium/Low

Full description

(Required) There shall be at most one occurrence of the # or ## operators in a single macro definition. This problem can be avoided by having only one occurrence of either operator in any single macro definition (i.e. one #, or one ## or neither).

Coding standards

MISRA C++ 2008 16-3-1

> (Required) There shall be at most one occurrence of the # or ## operators in a single macro definition.

Code examples

The following code example fails the check and will give a warning:

```
#defineD(x, y, z, yz)x ## y ## z/* Non-compliant */
#define C(x, y)# x ## y/* Non-compliant */
```

The following code example passes the check and will not give a warning about this issue:

```
#define A(x)#x/* Compliant */
#defineB(x, y)x ## y/* Compliant */
```

MISRAC++2008-16-3-2

Synopsis

The # and ## operators should not be used

Enabled by default

No

Severity/Certainty Low/Low

Full description (Advisory) The # and ## operators should not be used.

Coding standards MISRA C++ 2008 16-3-2

(Advisory) The # and ## operators should not be used.

Code examples The following code example fails the check and will give a warning:

```
#defineA(X,Y)X##Y/* Non-compliant */

#define A(Y)#Y/* Non-compliant */
```

The following code example passes the check and will not give a warning about this issue:

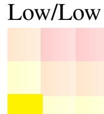
```
#define A(x)(x)/* Compliant */
```

MISRAC++2008-17-0-1

Synopsis #define or #undef of a reserved identifier in the standard library

Enabled by default Yes

Severity/Certainty Low/Low



Full description (Required) Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined. practice to #define a macro name that is a C/C++ reserved identifier, or C/C++ keyword or the name of any macro, object or function in the standard library. For example, there are some specific reserved words and function names that are known to give rise to undefined behaviour if they are redefined or undefined, including defined, __LINE__, __FILE__, __DATE__, __TIME__, __STDC__, errno and assert.

Coding standards MISRA C++ 2008 17-0-1

(Required) Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.

Code examples

The following code example fails the check and will give a warning:

```
#define__TIME__11111111/* Non-compliant */
```

The following code example passes the check and will not give a warning about this issue:

```
#define A(x)(x)/* Compliant */
```

MISRAC++2008-17-0-3

Synopsis

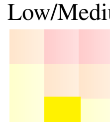
A library function is being overridden.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) The names of standard library functions shall not be overridden.

Coding standards

MISRA C++ 2008 17-0-3

(Required) The names of standard library functions shall not be overridden.

Code examples

The following code example fails the check and will give a warning:

```
extern "C" void strcpy(void);
void strcpy(void) {}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {}
```

MISRAC++2008-17-0-5

Synopsis

All uses of <setjmp.h>

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The setjmp macro and the longjmp function shall not be used.

Coding standards

CERT ERR34-CPP

Do not use longjmp

MISRA C++ 2008 17-0-5

(Required) The setjmp macro and the longjmp function shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
#include <setjmp.h>

jmp_buf ex;

void example(void) {
  setjmp(ex);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC++2008-18-0-1 (C++ only)

Synopsis

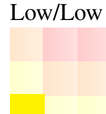
Uses of C library includes

Enabled by default

Yes

Severity/Certainty

Low/Low



Full description

(Required) The C library shall not be used. requires that the C++ version is used.

Coding standards

MISRA C++ 2008 18-0-1

(Required) The C library shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdio.h>
void example(void) {}
```

The following code example passes the check and will not give a warning about this issue:

```
#include <cstdio>
void example(void) {}
```

MISRAC++2008-18-0-2

Synopsis

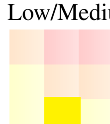
All uses of atof, atoi, atol and atoll

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) The library functions atof, atoi and atol from library <cstdlib> shall not be used.

Coding standards

CERT INT06-C

Use strtol() or a related function to convert a string token to an integer

MISRA C++ 2008 18-0-2

(Required) The library functions atof, atoi and atol from library <cstdlib> shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

int example(char buf[]) {
  return atoi(buf);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC++2008-18-0-3

Synopsis

All uses of abort, exit, getenv, and system

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.

Coding standards

MISRA C++ 2008 18-0-3

> (Required) The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

void example(void) {
  abort();
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC++2008-18-0-4

Synopsis

All uses of <time.h> functions: asctime, clock, ctime, difftime, gmtime, localtime, mktime, strftime, and time

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The time handling functions of library <ctime> shall not be used.

Coding standards

MISRA C++ 2008 18-0-4

(Required) The time handling functions of library <ctime> shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
#include <stddef.h>
#include <time.h>

time_t example(void) {
  return time(NULL);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC++2008-18-0-5

Synopsis

All uses of strcpy, strcmp, strcat, strchr, strspn, strcspn, strpbrk, strrchr, strstr, strtok, and strlen

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The unbounded functions of library <cstring> shall not be used. within the <cstring> library can read or write beyond the end of a buffer, resulting in undefined behaviour. Ideally, a safe string handling library should be used.

Coding standards

MISRA C++ 2008 18-0-5

(Required) The unbounded functions of library <cstring> shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
#include <string.h>

void example(void) {
  char buf[100];
  strcpy(buf, "Hello, world!\n");
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC++2008-18-2-1

Synopsis

All uses of the offsetof built-in function

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The macro offsetof shall not be used.

Coding standards

MISRA C++ 2008 18-2-1

(Required) The macro offsetof shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
#include <stddef.h>
//#include <sys/stat.h>
struct stat { int st_size; };
int example(void) {
  return offsetof(struct stat, st_size);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC++2008-18-4-1

Synopsis

All uses of malloc, calloc, realloc, and free

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Dynamic heap memory allocation shall not be used.

Coding standards

MISRA C++ 2008 18-4-1

(Required) Dynamic heap memory allocation shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
#include <stdlib.h>

void *example(void) {
  return malloc(100);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC++2008-18-7-1

Synopsis

All uses of <signal.h>

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The signal handling facilities of <csignal> shall not be used.

Coding standards

MISRA C++ 2008 18-7-1

(Required) The signal handling facilities of <csignal> shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
#include <signal.h>
#include <stddef.h>

void example(void) {
  signal(SIGFPE, NULL);
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC++2008-19-3-1

Synopsis

All uses of errno

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The error indicator errno shall not be used.

Coding standards

MISRA C++ 2008 19-3-1

(Required) The error indicator errno shall not be used.

Code examples

The following code example fails the check and will give a warning:

```
#include <errno.h>
#include <stdlib.h>
//int errno;

int example(char buf[]) {
  int i;
  errno = 0;
  i = atoi(buf);
  return (errno == 0) ? i : 0;
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```

MISRAC++2008-27-0-1

Synopsis
All uses of <stdio.h>

Enabled by default
Yes

Severity/Certainty
Low/Medium

Full description
(Required) The stream input/output library <cstdio> shall not be used.

Coding standards
MISRA C++ 2008 27-0-1

(Required) The stream input/output library <cstdio> shall not be used.

Code examples
The following code example fails the check and will give a warning:

```
#include <stdio.h>

void example(void) {
  printf("Hello, world!\n");
}
```

The following code example passes the check and will not give a warning about this issue:

```
void example(void) {
}
```