IAR Embedded Workbench

C-STAT® Static Analysis Guide

CSTAT-9

IAR SYSTEMS

EDITION NOTICE

Ninth edition: March 2018

Part number: CSTAT-9

This guide applies to version 5.0 and later of C-STAT.

Internal reference: M22/BB2.2, Hom7.5/Mym8.0, Skutt5.0, IJOA.

Contents

C-STAT for static analysis

Introduction to C-STAT and static analysis

These topics are described:

BRIEFLY ABOUT C-STAT AND THE CODING RULES

C-STAT is a static analysis tool that tries to find deviations from certain coding rules by performing one or more *checks* for the rule. The checks are grouped in *packages*. The various packages are:

- STDCHECKS

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

  Contains checks for CERT. In addition, some CERT rules and recommendations can be verified by checks for other standard rules, see *Mapping of CERT rules to C-STAT checks*, page 979.
- SECURITY

  Contains checks for rules from SANS Top25, OWASP and CWE.

- 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.

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 analysis of a specific file is terminated after a time limit that you can specify. When the time limit has been reached, the analysis will continue with the next file.

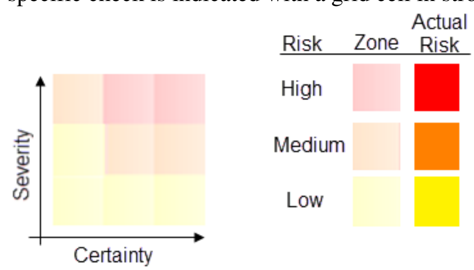
THE CHECKS AND THEIR DOCUMENTATION

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

- *Tag*, a unique identifier 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.

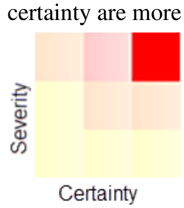
A grid shows the *severity* of the problems that code that does not conform to the rule (non-conformant code) can cause, and the level of *certainty* that the message reflects a true error in the source code. The grid is divided into three *zones*—indicated with pale colors—that reflect the *risks* based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

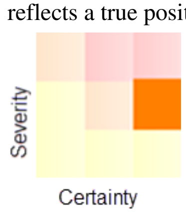
Example 1—high severity and high certainty = high risk

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.



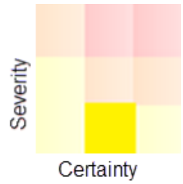
Example 2—medium severity and high certainty = medium risk

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.



THE SCOPE OF THE C-STAT CHECKS

The checks in C-STAT can be divided into checks performed on the source code and checks performed at link time.

Source code checks search for deviations from a coding rule in the C or C++ source code in the user project and any included user headers (included with `#include "xxx"`). System headers (included with `#include <xxx>`) and assembler source code are not searched.

Link time checks search for deviations from coding rules that specify how global and static objects (variables and functions) can be used. The search might be incomplete because the checks search the C or C++ source code for global and static objects and then C-STAT analyzes the code to see whether any deviations have occurred. If the user project contains assembler source code or third-party libraries, the search might yield false positives.

Also note that some MISRA C 2012 checks—MISRAC2012-Rule-5.2, MISRAC2012-Rule-5.3, MISRAC2012-Rule-5.4, MISRAC2012-Rule-5.5, and MISRAC2012-Rule-20.4—all have one variant for C89 and one for C99. The C89 variants are only used if the source code was compiled in C89 mode, otherwise the C99 variants are used.

Note: When you use C-STAT, the compiler options for each C/C++ source file must be the same as in the user project, otherwise the analysis might give incorrect results.

VARIOUS WAYS TO USE C-STAT

C-STAT is an integral part of the IAR Embedded Workbench IDE:

- You specify which packages of checks to perform in the **Select C-STAT Checks** dialog box.
- You perform a static analysis by choosing the appropriate commands from the **Project>C-STAT Static Analysis** menu.

- You can view the result of the performed analysis in the **C-STAT Messages** window.
- You can create a report in HTML format by choosing the appropriate commands from the **Project>C-STAT Static Analysis** menu.

C-STAT can also be used from the command line, which is useful if you build your project using a make file:

- `ichecks.exe`—use the `ichecks` tool to generate a *manifest file* that contains only the checks that you want to perform.
- `icstat.exe`—use the `icstat` tool to perform a C-STAT static analysis on a project, with the manifest file as input.
- `ireport.exe`—use the `ireport` tool to generate an HTML report of a previously performed analysis.

Finally, you can use C-STAT together with the IAR Command Line Build Utility (`iarbuild.exe`) for regression testing.

Using C-STAT

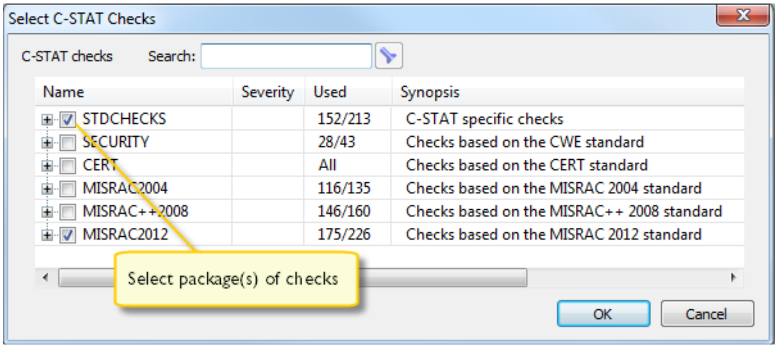
These tasks are described:

- *Getting started analyzing using C-STAT*, page 9
- *Generating an analysis report*, page 12
- *Performing regression testing*, page 13
- *Performing an analysis from the command line*, page 14

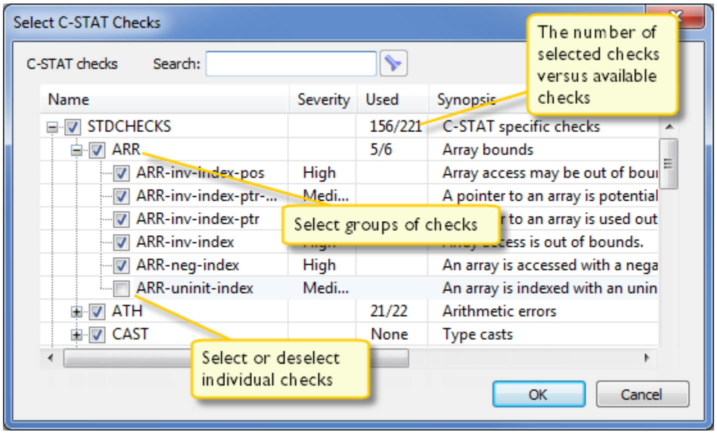
GETTING STARTED ANALYZING USING C-STAT

1. Before you perform a static analysis, make sure your project builds without errors. For information about how to build a project, see the *IDE Project Management and Building Guide*.
2. Choose **Project>Options** and select the **Static Analysis** category. On the **C-STAT Static Analysis** page, click **Select C-STAT Checks**.

3 In the **Select C-STAT Checks** dialog box, select the packages of checks you want to use. For example **STDCHECKS**.



4 For each package, select groups of checks or individual checks:



For information about a specific check, select it and press F1 to open the context-sensitive online help system.

When you have made your settings, click **OK** and then **OK** again.

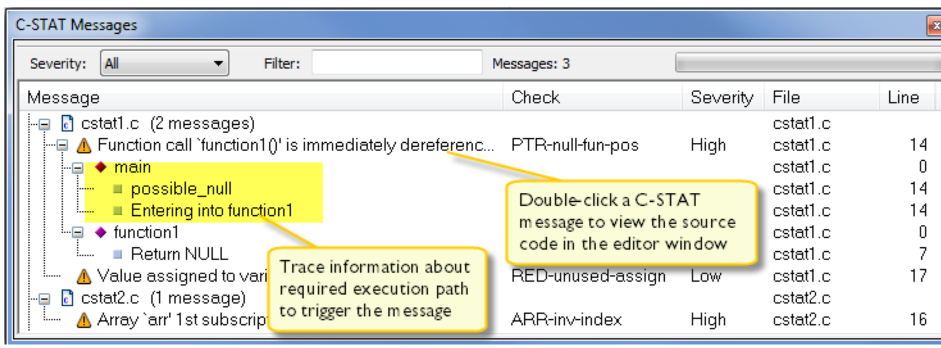
5 To perform an analysis, make sure the project is active and execute one of these steps:

- To analyze your project, select the project in the **Workspace** window and choose **Project>C-STAT Static Analysis>Analyze Project**.
- To analyze one or more individual files, select the file(s) in the **Workspace** window and choose **Project>C-STAT Static Analysis>Analyze File(s)**.

Alternatively, use the corresponding commands on the context menu in the **Workspace** window instead.

Note: The next time you perform an analysis and if you have made changes to your source code since the previous analysis, you should first clean the database to avoid problems due to mixing old and new data in the database. Choose **Project>C-STAT Static Analysis>Clear Analysis Results**.

6 The result of the performed analysis is listed in the **C-STAT Messages** window.

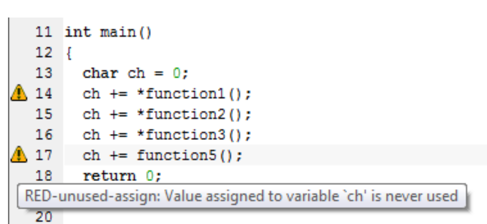


For information about a specific check, select it and press F1 to open the context-sensitive online help system.

For reference information, see *C-STAT Messages window*, page 17.

Note: If there are any problems when analyzing, the **Build Log** window displays detailed information.

7 Double-click a C-STAT message to view the corresponding source code in the editor window:



Point at a message with the mouse pointer to get tooltip information about which check that caused the message.

8 Correct the error and click the next message in the **C-STAT Messages** window. Continue until all messages have been processed.

Note: C-STAT has a predefined macro, `__CSTAT__`, that you can use to explicitly include or exclude specific parts of source code from the analysis, see *__CSTAT__*, page 25. There are also specific C-STAT pragma directives that suppress one or more checks for selected source lines, see *Descriptions of compiler extensions for C-STAT*, page 22.

GENERATING AN ANALYSIS REPORT

1 Perform your analysis, see *Getting started analyzing using C-STAT*, page 9.

2 To generate your report:

- In the IDE, choose **Project>C-STAT Static Analysis** and choose either **Generate HTML Summary** or **Generate Full HTML Report** depending on which type of report you want to produce.

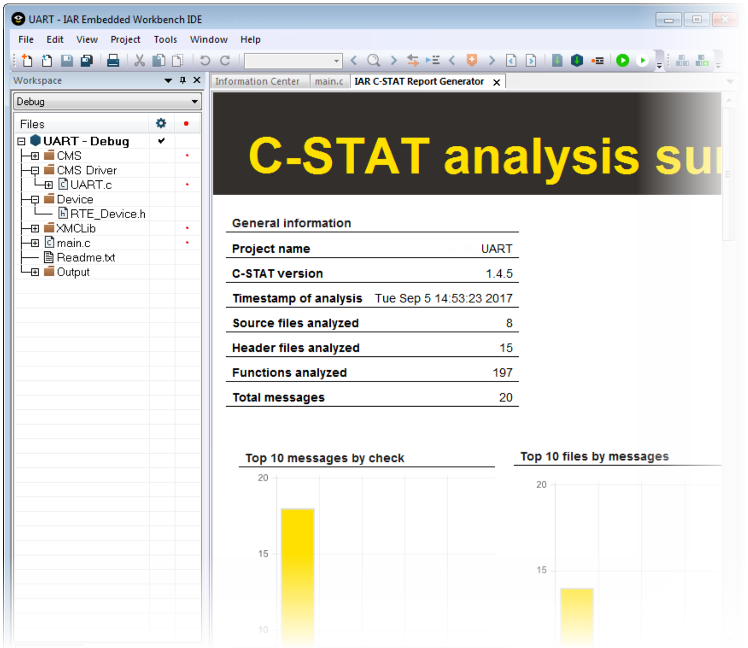
  The report will be based on the latest performed analysis. If you have modified your source code files after the latest analysis, you might want to update the analysis before you generate the report.

- On the command line, specify your `ireport` options, for example like this:

  ```
  ireport --db cstat.db --project project1 --output
  tutor_report.html
  ```

  This will generate a summary report named `tutor_report.html` from the database `cstat.db` with `project1` as an identifying name for the project. The report can be viewed in a web browser or in the IAR Embedded Workbench IDE.

3 This is an example of a summary report:



PERFORMING REGRESSION TESTING

Regression testing is a method for testing the whole or parts of your source code after you have modified it, to verify that no errors have been added as a result of the modifications.

1 After you have analyzed your project using C-STAT and possibly corrected some errors, it can be useful to perform regression testing using the IAR Command Line Build Utility (`iarbuild.exe`) located in the `common\bin` directory.

To clean the database from old errors, use a command line like this:

```
iarbuild.exe MyProject.ewp -cstat_clean Debug
```

To analyze all files in the project, use a command line like this:

```
iarbuild.exe MyProject.ewp -cstat_analyze Debug
```

2 C-STAT generates output information, for example:

```
Analyzing configuration: MyProject - Debug
Updating build tree...

Starting C-STAT analysis

Analysis completed. 164 message(s)
```

3 Compare the number of messages reported with the number of messages produced in previous builds. If the number has increased, new errors have been introduced as a result of earlier development.

4 In the IDE, open your project, perform the analysis, and locate the cause of the new message.

Alternatively, you can create an HTML report from the command line, for example like this:

```
ireport.exe --db cstat.db --project MyProject.ewp --full --output
MyProject.html
```

This creates a report in `MyProject.html`, see also *Generating an analysis report*, page 12.

5 Typically, you might want to repeat this process during nightly builds to continuously control that existing code is not affected by new code.

For more information about the IAR Command Line Build Utility, see the *IDE Project Management and Building Guide*.

PERFORMING AN ANALYSIS FROM THE COMMAND LINE

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

For information about the checks, see *C-STAT checks*, page 37.

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 performed (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 perform 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: `icc`*xxxxx* is the invocation of the compiler and `ilink`*xxxxx* is the invocation of the ILINK Linker. *xxxxx* should be replaced with an identifier that is unique to your IAR Embedded Workbench product package. Refer to the compiler documentation that was delivered with the product, for what to replace *xxxxx* with.

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` of the object file should be `r`*xx*, where *xx* is a numeric part that identifies your product package. Refer to the *IDE Project Management and Building Guide* for what to replace *xx* with.

In these example command lines, `--db` specifies a file where the resulting database 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 `commands` command which means that you collect all your commands in a specific file in combination with `--parallel`. 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 commands commands.txt
--parallel 4
```

`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.

Note: The next time you perform an analysis, you should first clean the database by using the `clear` command to avoid problems due to mixing old and new data in the database.

4 After running `icstat` on the `cstat1.c` file, these messages are listed on 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 parts of source code from the analysis. There are also specific C-STAT pragma directives that suppress one or more checks for selected source lines, see *Descriptions of compiler extensions for C-STAT*, page 22.

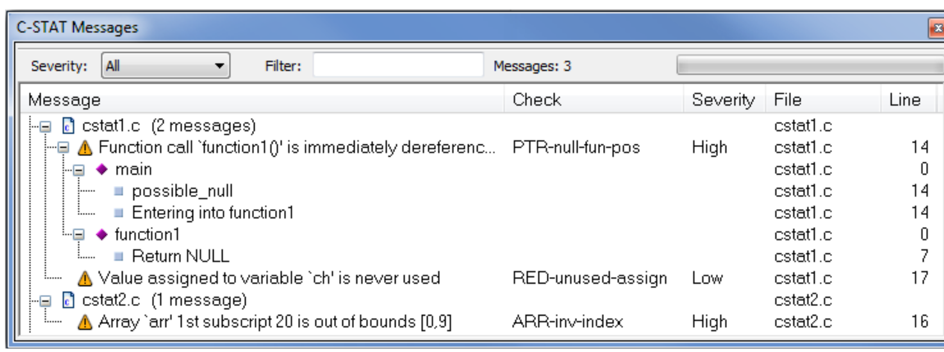
Reference information on the graphical environment

Reference information about:

- *C-STAT Messages window*, page 17
- *C-STAT Static Analysis options*, page 19
- *Extra Options*, page 20
- *Select C-STAT Checks dialog box*, page 21

C-STAT Messages window

The **C-STAT Messages** window is automatically displayed when you perform a C-STAT analysis.



This window displays the result of a performed C-STAT static analysis.

See also *Getting started analyzing using C-STAT*, page 9.

Toolbar menu

Severity

Selects which severity level of the messages to be displayed. Choose between **All** (shows all messages), **Medium/High** (shows messages of Medium and High severity), or **High** (shows only messages of High severity).

Filter

Filters the messages so that only messages that contain the text you specify will be listed (the filter is case-sensitive). This is useful if you want to search the message information.

Messages

Lists the number of C-STAT messages after a performed analysis.

Progress bar

Shows the progress of the ongoing analysis.

Display area

The display area shows messages per file and linkage. The messages can be expanded and collapsed. For each file, the number of messages and the number of C-STAT pragma messages are displayed.

Message

Lists the C-STAT message for the check.

Check

The name of the check.

Severity

The severity of the check, **High**, **Medium**, or **Low**.

File

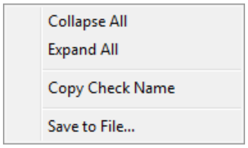
The name of the file where the non-conformant code construct is found.

Line

The line number of the non-conformant code construct.

Context menu

This context menu is available:



These commands are available:

Collapse All

Collapses all file nodes in the **C-STAT Messages** window.

Expand All

Expands all file nodes in the **C-STAT Messages** window.

Copy Check Name

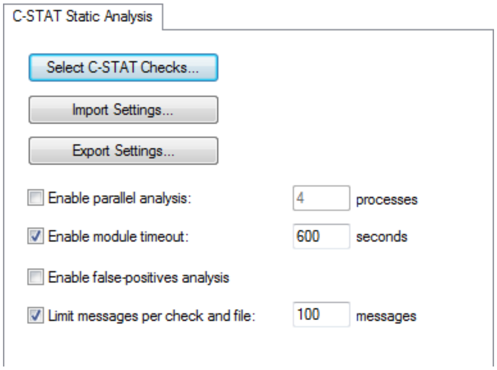
Copies the name of the selected check. Use the copied name in the **C-STAT Settings** dialog box to search for a specific check.

Save to File

Saves the result of a performed analysis to a text file.

C-STAT Static Analysis options

To open the **C-STAT Static Analysis** page, choose **Project>Options** and select the **Static Analysis** category.



Use this page to specify options for performing a static analysis using C-STAT.

Select C-STAT Checks

Opens the **Select C-STAT Checks** dialog box where you can select which checks to perform.

Import Settings

Opens a standard open dialog box to use for locating and opening an XML file that contains the checks to perform. The content of the file will be imported and can be modified in the **Select C-STAT Checks** dialog box.

Export Settings

Opens a standard save dialog box for locating and saving an XML file with your currently selected checks.

Enable parallel analysis

Enables C-STAT to perform analysis in parallel.

Enable module timeout

Specify the number of seconds after which the analysis terminates.

Processes

Specify the number of processes to be used by C-STAT for performing an analysis.

Enable false-positives analysis

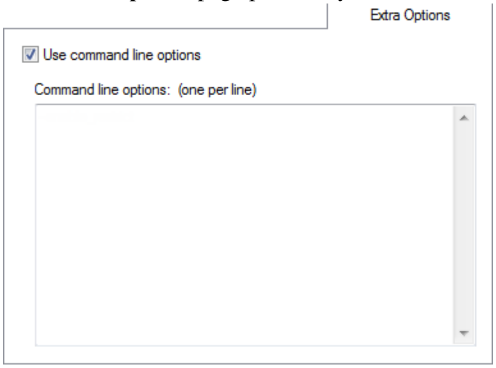
Attempts to remove false messages, commonly referred to as *false positives*.

Limit messages per check and file

Specify the maximum number of messages to be produced per check and file.

Extra Options

The **Extra Options** page provides you with a command line interface to the tool.

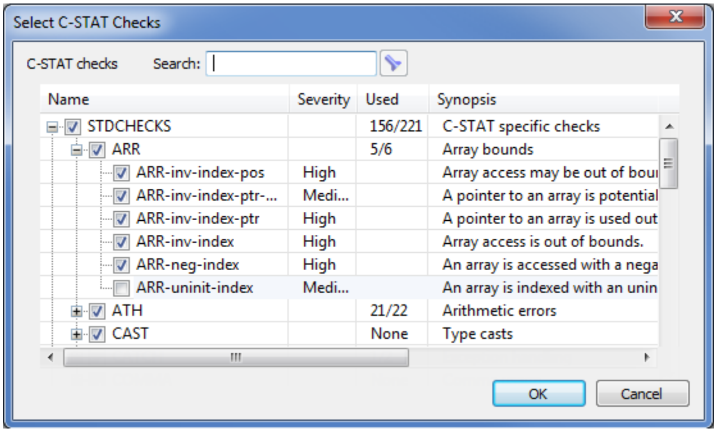


Use command line options

Specify additional command line arguments to be passed to the tool (not supported by the GUI).

Select C-STAT Checks dialog box

The **Select C-STAT Checks** dialog box is available from the **C-STAT Static Analysis** options page.



Use this dialog box to specify the checks to include during a C-STAT static analysis. You can select packages or groups of checks, or individual checks to perform by selecting the corresponding check boxes.

For reference information about individual checks, select a check and press F1 to open the context-sensitive help.

Search

Type a text string to be used as a filter.

Name

Lists all packages, groups, and checks. Select the ones you want to perform.

Severity

Shows the severity for each check, which can be **High**, **Medium**, or **Low**.

Used

Shows how many of the checks in the package or group that will performed during a C-STAT static analysis (only if the package or group actually is selected). The values can be **All**, **None**, or the number of selected checks out of the total amount.

Synopsis

Gives a short description of the packages, groups, and checks.

Descriptions of compiler extensions for C-STAT

Reference information about:

C-STAT directives in comments

Syntax

```
//cstat op [op op...]
/*cstat op [op op...]*/
```

Parameters

`op` is one of:

`-tag`	Disables the specified C-STAT check until the end of the compilation unit or until a matching `+tag` is found.
`+tag`	Reenables the specified C-STAT check until the end of the compilation unit or until a matching `-tag` is found.
`!tag`	Disables the specified C-STAT check for a single line. If the line of the specified directive consists of more than just the comment, the line where the directive is placed is used for disabling the specified C-STAT check. Otherwise, the next line that consists of more than just a comment is used.
`#tag`	Disables the specified C-STAT check for the immediately following function.
`tag`	`tag` to be replaced with the tag for a specific check, for example `MISRAC2012-Rule-4.2`.

Note that you can use the wildcard (`*`) character to match multiple tags and thus disable multiple checks.

Description

Use the comment characters (and the operators) to disable or enable C-STAT messages for specific checks.

Example

```
//cstat -MISRAC2004* -MISRAC2012-Rule-4.2
// ...
// Messages about MISRA C 2012 rule 4.2 and the whole MISRA C
// 2004 package suppressed here
// ...
//cstat +MISRAC2004* +MISRAC2012-Rule-4.2
// ...
// Messages about MISRA C 2012 rule 4.2 and the whole MISRA C
// 2004 package unsuppressed here
// ...

//cstat !MISRAC2004-6.3
int a;
```

or

```
int a; //cstat !MISRAC2004-6.3
```

will disable the message given by MISRA C 2004 6.3 regarding the `int a;` statement.

```
//cstat #ARR-inv-index
void f(...)
{
...// Messages about ARR-inv-index suppressed here
}
```

cstat_disable

Syntax

`#pragma cstat_disable="`*`tag`*`"[,"`*`tag`*`"...]`

Parameters

tag	The tag of a C-STAT check.

Description

Use this pragma directive to suppress the specified C-STAT check until the end of the compilation unit or until a matching `#pragma cstat_restore` directive is encountered.

Example

```
#pragma cstat_disable = "MISRAC2012-Rule-9.2",
"MISRAC2012-Rule-10.3"
  // ...
  // Messages about rules 9.2 and 10.3 suppressed here
  // ...
```

See also

cstat_restore, page 24

cstat_enable

Syntax

`#pragma cstat_enable="tag"[,"tag"...]`

Parameters

tag	The tag of a C-STAT check.

Description

Use this pragma directive to unsuppress the specified C-STAT check until the end of the compilation unit, or until a matching `#pragma cstat_restore` directive is encountered.

Example

```
#pragma cstat_enable = "MISRAC2012-Rule-10.3"
  // ...
  // Messages about rule 10.3 not suppressed here
  // ...
```

See also

cstat_enable, page 24

cstat_restore

Syntax

`#pragma cstat_restore="tag"[,"tag"...]`

Parameters

tag	The tag of a C-STAT check.

Description

Use this pragma directive to undo the effects of the most recent `cstat_enable` or `cstat_disable` directive for the same check(s).

Example

```
#pragma cstat_restore = "MISRAC2012-Rule-10.3"
  // ...
  // Messages about rule 10.3 suppressed here
  // ...
```

cstat_suppress

Syntax

`#pragma cstat_suppress="tag"[,"tag"...]`

Parameters

tag	The tag of a C-STAT check.

Description

Use this pragma directive to suppress the specified C-STAT check until the end of the immediately following line.

__CSTAT__

Description

A predefined macro that is defined when the code is processed for analysis. You can use it to explicitly include or exclude specific parts of source code from the analysis.

Example

```
#ifndef __CSTAT__
  /* Code here is not visible to the analysis */
#endif
```

Descriptions of C-STAT options

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

Rules for specifying a filename or directory as parameters

Description

These rules apply for options that take a filename or directory as parameters:

- Options that take a filename as a parameter can optionally take a file path. The path can be relative or absolute. For example, to generate a check manifest to the file `cstat_checks.txt` in the directory `..\checks`:

  ```
  ichecks --package misrac2012 --output
  ..\checks\cstat_checks.txt
  ```

- `/` can be used instead of `\` as the directory delimiter.
- By specifying `-`, input files and output files can be redirected to the standard input and output stream, respectively. For example:

  ```
  ichecks --package misrac2012 --output -
  ```

  For options where it is not relevant to direct files to standard input or output, `-` is not supported.

--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 perform 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 perform, 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 perform the specified check.

To set related options, choose:

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

--checks

Syntax `--checks` *`filename`*

Parameters

`filename`	The name of the manifest file that contains the checks that `icstat` will perform. See also *Rules for specifying a filename or directory as parameters*, page 25.

For use with `icstat`

Description Use this option to specify the file that contains the checks to perform. You create the file using `ichecks`, see *Performing an analysis from the command line*, page 14.

This option is not available in the IDE.

--db

Syntax `--db` *`filename`*

Parameters

`filename`	`icstat`: The name of the file where the analysis result will be stored as a database. `ireport`: The name of the database file that contains the result of a previously performed analysis. See also *Rules for specifying a filename or directory as parameters*, page 25.

For use with `icstat`, `ireport`

Description Use this option to specify the name of the 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`, `cert`, `security`, `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 perform the default checks.

To set related options, choose:

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

--exclude

Syntax `--exclude {filename|directory}`

Parameters

filename	The name of the source file to exclude. See also *Rules for specifying a filename or directory as parameters*, page 25.
directory	The name of the directory where the source files to exclude are stored. See also *Rules for specifying a filename or directory as parameters*, page 25.

Note that the string you specify can include the `*` and `?` characters, where `*` matches any sequence of characters (including the empty sequence) and `?` matches any single character.

For use with `icstat`

Description Use this option to exclude one or more source files (not, for example, header files) from the source file analysis (the command `analyze`); more specifically, files whose part of their absolute path completely matches the string you specify. The -- exclude option cannot exclude files from the application linking analysis (the command `link_analyze`). For more information on the analysis commands, see *Summary of icstat commands*, page 34.

Example `--exclude library`

Will for example, exclude `E:\project\library\libxml.c`, but will not exclude `E:\project\third_party_library\libxml.c` or `E:\project\library.c`.

`--exclude libxml*`

Will for example, exclude `E:\project\library\libxml-2.7.6.c\main.c` and `E:\project\libxml.c`, but will not exclude `E:\project\api_libxml.c`.

`--exclude library\libxml`

Will for example, exclude `E:\project\library\libxml\main.c`, but will not exclude `E:\project\libxml-2.7.6.c\main.c`.

This option is not available in the IDE.

--fpe

Syntax `--fpe`

For use with `icstat`

Description Use this option to make `icstat` attempt to remove false messages, commonly referred to as *false positives*.

Project>Options>Static Analysis>C-STAT Static Analysis>Enable false-positive analysis

--full

Syntax `--full`

For use with `ireport`

Description Use this option to make `ireport` generate a full report in HTML, which means that all checks (suppressed and non-suppressed) are included at the end of the report.

To set this option, choose:

Project>C-STAT Static Analysis>Generate Full HTML Report

--group

Syntax

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

Parameters

group	The group of checks that you want to perform, 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 perform the specified group of checks.

To set related options, choose:

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

--output

Syntax

`--output filename`

Parameters

filename	The name of the output file. See also *Rules for specifying a filename or directory as parameters*, page 25.

For use with

`ichecks`, `ireport`

Description

Use this option to explicitly specify a different output filename.

`ichecks`: By default, the generated output produced by `ichecks` is located in a file with the name `cstat_sel_checks.txt`.

`ireport`: By default, the generated output produced by `ireport` is located in a file with the name *`project_name`*`.html`.

For `ichecks`: This option is not available in the IDE.

For `ireport`: **Project>Options>Static Analysis>C-STAT Static Analysis>Generate Full HTML Report**

or

Project>Options>Static Analysis>C-STAT Static Analysis>Generate HTML Summary

--package

Syntax

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

Parameters

package	The package of checks that you want to perform. 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 perform the specified package of checks.

To set related options, choose:

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

--parallel

Syntax

`--parallel threads`

Parameters

threads	The maximum number of threads to use during parallel analysis.

For use with

`icstat`

Description

Use this option to specify the maximum number of threads to use during parallel analysis.

Note: This option might cause subsequently performed analyses to produce more or fewer messages. This is because the summary information for the source files might change depending on the order in which they are analyzed.

Project>Options>Static Analysis>Enable parallel analysis

--project

Syntax `--project` *`name`*

Parameters

`name`	A name to identify the project in the report.

For use with `ireport`

Description

Use this option to specify a name for the project in the report.

This option is mandatory.

This option is not available in the IDE.

--timeout

Syntax `--timeout` *`seconds`*

Parameters

`seconds`	The number of seconds before the analysis of a module terminates.

For use with `icstat`

Description

Use this option to specify the number of seconds that the analysis of a module is allowed to take before it terminates.

Project>Options>Static Analysis>Module timeout

--timeout_check

Syntax `--timeout_check` *`seconds`*

Parameters

`seconds`	The number of seconds that each check is allowed to take before the analysis terminates.

For use with `icstat`

Description

Use this option to specify the number of seconds that each check is allowed to take before the analysis terminates. This limit includes various internal operations performed during the analysis.

Project>Options>Static Analysis>Extra Options

Description of the C-STAT command line tools

Reference information about:

- *The icstat tool*, page 33
- *The ichecks tool*, page 35
- *The ireport tool*, page 35

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

THE ICSTAT TOOL

Use the `icstat` tool to perform a C-STAT static analysis on a project, with a previously produced manifest file as input. You produce the manifest file using the `ichecks` tool.

Invocation syntax for icstat

The invocation syntax for `icstat`:

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

The different parts are:

Syntax parts	Description
`commands`	Commands that define an operation to be performed, see *Summary of icstat commands*, page 34.
`options`	Command line options that define actions to be performed, see *Summary of icstat options*, page 34. 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 *Performing an analysis from the command line*, page 14.

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 analysis 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 *Performing an analysis from the command line*, page 14.

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 perform.
`--db`	Contains analysis information (mandatory).
`--exclude`	Excludes file(s) from the analysis.
`--fpe`	Makes `icstat` attempt to remove false messages (false positives).
`--parallel`	Specifies the number maximum number of threads to use during parallel analysis.
`--timeout`	Specifies the number of seconds that the analysis of a module is allowed to take before it terminates.
`--timeout_check`	Specifies the number of seconds that the each check is allowed to take before the analysis terminates.

Table 3: icstat options summary

For more information, see *Descriptions of C-STAT options*, page 25.

THE ICHECKS TOOL

Use the `ichecks` tool to generate a *manifest file* that contains only the checks that you want to perform. Use this file as input to the `icstat` tool.

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 *Performing an analysis from the command line*, page 14.

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 25.

THE IREPORT TOOL

Use the `ireport` tool to produce an HTML report of a previous analysis performed by C-STAT. The report presents statistics both in numbers and as tables. Two different types of reports that can be produced:

- A summary that includes information about, for example, project-wide enabled checks, the total amount of messages, suppressed checks (if any), messages for each check, etc.
- A full report that contains the same information as the summary, but also information about all suppressed and non-suppressed messages at the end of the report. The tables can be collapsed and expanded, and the columns can be sorted.

Invocation syntax for ireport

The invocation syntax for `ireport`:

`ireport` *`options`*

For an example, see *Performing an analysis from the command line*, page 14.

Summary of ireport options

This table summarizes the `ireport` options:

Command line option	Description
`--db`	Specifies the database that the report will be based on.
`--full`	Produces a full report, including information about suppressed and non-suppressed checks.
`--output`	Specifies the name of the produced report.
`--project`	Specifies a name for the project.

Table 5: ireport options summary

For more information, see *Descriptions of C-STAT options*, page 25.

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	An 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	An 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 6: 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 were found.
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

Table 6: Summary of checks

Check	Synopsis
CONST-member-ret (C++ only)	A member function qualified as `const` returns a pointer member variable.
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.

Table 6: Summary of checks

Check	Synopsis
CPU-delete-void (C++ only)	A pointer to void is used in delete, causing the destructor not to be called.
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.

Table 6: Summary of checks

Check	Synopsis
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.
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-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.

Table 6: Summary of checks

Check	Synopsis
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.
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.

Table 6: Summary of checks

Check	Synopsis
`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.
`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`	A memory location is freed 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.

Table 6: Summary of checks

Check	Synopsis
`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`	An allocation call tries to allocate memory based on a `sizeof` operator, but the destination type of the call is of a different type.
`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 type of the pointer 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-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`	A stack address is stored outside a function via a 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

Table 6: Summary of checks

Check	Synopsis
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.
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-alloc-zero-bytes	Checks that an allocation does not allocate zero bytes

Table 6: Summary of checks

Check	Synopsis
`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.
`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.

Table 6: Summary of checks

Check	Synopsis
`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.
`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-order`	Expressions that depend on order of evaluation were found.
`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.
`STRUCT-signed-bit`	There are signed single-bit fields (excluding anonymous fields).

Table 6: Summary of checks

Check	Synopsis
`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.
`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.
`CERT-EXP19-C`	No braces for the body of an if, for, or while statement
`CERT-FIO37-C`	A string returned by fgets() and fgetsws() might contain NULL characters.
`CERT-FIO38-C`	A FILE object is copied.
`CERT-SIG31-C`	Shared objects in a signal handler are accessed or modified.
`SEC-BUFFER-memory-leak-alias`	A memory leak is caused by incorrect deallocation.
`SEC-BUFFER-memory-leak`	A memory leak is caused by incorrect deallocation.
`SEC-BUFFER-memset-overrun-pos`	A call to memset might overrun the buffer.
`SEC-BUFFER-memset-overrun`	A call to memset overruns the buffer.
`SEC-BUFFER-qsort-overrun-pos`	Arguments passed to qsort might cause it to overrun.
`SEC-BUFFER-qsort-overrun`	Arguments passed to qsort cause it to overrun.
`SEC-BUFFER-sprintf-overrun`	A call to the sprintf function will overrun the target buffer.

Table 6: Summary of checks

Check	Synopsis
SEC-BUFFER-std-sort-overrun-pos (C++ only)	Use of std::sort might cause a buffer overrun.
SEC-BUFFER-std-sort-overrun (C++ only)	A buffer overrun is caused by use of std::sort.
SEC-BUFFER-strcat-overrun-pos	A call to the strcat function might overrun the target buffer.
SEC-BUFFER-strcat-overrun	A call to the strcat function will overrun the target buffer.
SEC-BUFFER-strcpy-overrun-pos	A call to the strcpy function might overrun the target buffer.
SEC-BUFFER-strcpy-overrun	A call to the strcpy function will overrun the target buffer.
SEC-BUFFER-strncat-overrun-pos	A buffer overrun might be caused by a call to strncat.
SEC-BUFFER-strncat-overrun	A call to strncat causes a buffer overrun.
SEC-BUFFER-strncmp-overrun-pos	A call to strncmp might cause a buffer overrun.
SEC-BUFFER-strncmp-overrun	A buffer overrun is caused by a call to strncmp.
SEC-BUFFER-strncpy-overrun-pos	The target buffer might be overrun by a call to the strncpy function.
SEC-BUFFER-strncpy-overrun	A call to the strncpy function will overrun the target buffer.
SEC-BUFFER-tainted-alloc-size	A user is able to control the amount of memory used in an allocation.
SEC-BUFFER-tainted-copy-length	A tainted value is used as the size of the memory copied from one buffer to another.
SEC-BUFFER-tainted-copy	User input is copied into a buffer.
SEC-BUFFER-tainted-index	An array is accessed with an index derived from user input.
SEC-BUFFER-tainted-offset	A user-controlled variable is used as an offset to a pointer without proper bounds checking.
SEC-BUFFER-use-after-free-all	A pointer is used after it has been freed, on all execution paths.
SEC-BUFFER-use-after-free-some	A pointer is used after it has been freed, on some execution paths.

Table 6: Summary of checks

Check	Synopsis
SEC-DIV-0-compare-after	After a successful comparison with 0, a variable is used as a divisor.
SEC-DIV-0-compare-before	A variable is first used as a divisor, then compared with 0.
SEC-DIV-0-tainted	User input is used as a divisor without validation.
SEC-FILEOP-open-no-close	All file pointers obtained dynamically by means of Standard Library functions must be explicitly released.
SEC-FILEOP-path-traversal	User input is used as a file path, or used to derive a file path.
SEC-FILEOP-use-after-close	A file resource is used after it has been closed.
SEC-INJECTION-sql	User input is improperly used in an SQL statement
SEC-INJECTION-xpath	User input is improperly used as an XPath expression
SEC-LOOP-tainted-bound	A user-controlled value is used as part of a loop condidition.
SEC-NULL-assignment-fun-pos	A pointer that might have been assigned the value NULL is dereferenced.
SEC-NULL-assignment	A pointer is assigned the value NULL, then dereferenced.
SEC-NULL-cmp-aft	A pointer is dereferenced, then compared with NULL.
SEC-NULL-cmp-bef-fun	A pointer is compared with NULL, then dereferenced by a function.
SEC-NULL-cmp-bef	A pointer is compared with NULL, then dereferenced.
SEC-NULL-literal-pos	A literal pointer expression (e.g. NULL) is dereferenced by a function call.
SEC-STRING-format-string	User input is used as a format string.
SEC-STRING-hard-coded-credentials	The application hard codes a username or password to connect to an external component.
MISRAC2004-1.1	Code was found that does not conform to the ISO/IEC 9899:1990 standard.

Table 6: Summary of checks

Check	Synopsis
MISRAC2004-1.2_a	There are read accesses from local buffers that are not preceded by write accesses.
MISRAC2004-1.2_b	On all execution paths, one or more fields are read from a struct before they are initialized.
MISRAC2004-1.2_c	An expression resulting in 0 is used as a divisor.
MISRAC2004-1.2_d	A variable was found that is assigned the value 0, and then used as a divisor.
MISRAC2004-1.2_e	A variable is used as a divisor after a successful comparison with 0.
MISRAC2004-1.2_f	A variable used as a divisor is subsequently compared with 0.
MISRAC2004-1.2_g	A value that is determined using interval analysis to be 0 is used as a divisor.
MISRAC2004-1.2_h	An expression that might 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 assembler statements were found that are not encapsulated in functions.
MISRAC2004-2.2	Uses of // comments were found.
MISRAC2004-2.3	The character sequence /* was found inside comments.
MISRAC2004-2.4	Code sections in comments were found, where the comment ends in ;, {, or } characters.
MISRAC2004-5.2	An identifier name was found that is not distinct in the first 31 characters from other names in an outer scope.
MISRAC2004-5.3	A typedef declaration was found with a name already used for a previously declared typedef.
MISRAC2004-5.4	A class, struct, union, or enum declaration was found that clashes with a previous declaration.
MISRAC2004-5.5	An identifier is used that might clash with another static identifier.
MISRAC2004-5.6	Identifier reuse in different namespaces

Table 6: Summary of checks

Check	Synopsis
MISRAC2004-5.7	An identifier in a variable, enumeration, struct, #define, or union definition is reused.
MISRAC2004-6.1	Arithmetic is performed on objects of type plain char, without an explicit signed or unsigned qualifier.
MISRAC2004-6.2	A signed or unsigned char is used on character data.
MISRAC2004-6.3	One or more of the basic types char, int, short, long, double, and float are used without a typedef.
MISRAC2004-6.4	Bitfields of plain int type were found.
MISRAC2004-6.5	Signed bitfields consisting of a single bit (excluding anonymous fields) were found.
MISRAC2004-7.1	Uses of octal integer constants were found.
MISRAC2004-8.1	Functions were found that are used despite not having a valid prototype.
MISRAC2004-8.2	An implicit `int` was found in a declaration.
MISRAC2004-8.3	A declaration and definition for a function were found that use different type qualifiers.
MISRAC2004-8.5_a	A global variable is declared in a header file.
MISRAC2004-8.5_b	One or more non-inlined functions are defined in header files.
MISRAC2004-8.6	A function declaration was found at block scope.
MISRAC2004-8.7	A global object was found that is only referenced from a single function.
MISRAC2004-8.8_a	Multiple declarations of the same external object or function were found.
MISRAC2004-8.8_b	Multiple declarations of the same external object or function were found.
MISRAC2004-8.9	Multiple definitions or no definition were found for an external object or function.
MISRAC2004-8.10	An externally linked object or function was found referenced in only one translation unit.

Table 6: Summary of checks

Check	Synopsis
MISRAC2004-8.12	External arrays are declared without their size being stated explicitly or defined implicitly by initialization.
MISRAC2004-9.1_a	A variable is read before it is assigned a value, on all execution paths.
MISRAC2004-9.1_b	On some execution paths, a variable is read before it is assigned a value.
MISRAC2004-9.1_c	An uninitialized or NULL pointer that is dereferenced was found.
MISRAC2004-9.2	A non-zero array initialization was found that does not exactly match the structure of the array declaration.
MISRAC2004-9.3	Partially initialized enum.
MISRAC2004-10.1_a	An expression of integer type was found that is implicitly converted to a narrower or differently signed underlying type.
MISRAC2004-10.1_b	A complex expression of integer type was found that is implicitly converted to a different underlying type.
MISRAC2004-10.1_c	A non-constant expression of integer type was found that is implicitly converted to a different underlying type in a function argument.
MISRAC2004-10.1_d	A non-constant expression of integer type was found that is implicitly converted to a different underlying type in a return expression.
MISRAC2004-10.2_a	An expression of floating type was found that is implicitly converted to a narrower underlying type.
MISRAC2004-10.2_b	An expression of floating type was found that is implicitly converted to a narrower underlying type.
MISRAC2004-10.2_c	A non-constant expression of floating type was found that is implicitly converted to a different underlying type in a function argument.
MISRAC2004-10.2_d	A non-constant expression of floating type was found that is implicitly converted to a different underlying type in a return expression.

Table 6: Summary of checks

Check	Synopsis
MISRAC2004-10.3	A complex expression of integer type was found that is cast to a wider or differently signed underlying type.
MISRAC2004-10.4	A complex expression of floating type was found that is cast to a wider or different underlying type.
MISRAC2004-10.5	Detected a bitwise operation on unsigned char or unsigned short, that are not immediately cast to this type to ensure consistent truncation.
MISRAC2004-10.6	Constants of unsigned type were found that do not have a `U` suffix.
MISRAC2004-11.1	Conversions were found between a pointer to a function and a type other than an integral type.
MISRAC2004-11.3	A cast between a pointer type and an integral type was found.
MISRAC2004-11.4	A pointer to object type was found that is cast to a pointer to different object type.
MISRAC2004-11.5	Casts were found that that remove any const or volatile qualification.
MISRAC2004-12.1	Expressions were found without parentheses, making the operator precedence implicit instead of explicit.
MISRAC2004-12.2_a	Expressions were found that depend on the order of evaluation.
MISRAC2004-12.2_b	More than one read access with volatile-qualified type was found within one sequence point.
MISRAC2004-12.2_c	More than one modification access with volatile-qualified type was found within one sequence point.
MISRAC2004-12.3	Sizeof expressions were found that contain side effects.
MISRAC2004-12.4	Right-hand operands of && or || were found that contain side effects.
MISRAC2004-12.5	The operands of a logical && or || is not an identifier, a constant, a parenthesized expression or a sequence of the same logical operator.

Table 6: Summary of checks

Check	Synopsis
MISRAC2004-12.6_a	Operands of logical operators (&&, ||, and !) were found that are not effectively Boolean.
MISRAC2004-12.6_b	Uses of arithmetic operators on Boolean operands were found.
MISRAC2004-12.7	Applications of bitwise operators to signed operands were found.
MISRAC2004-12.8	Shifts were found where the right-hand operand might be negative, or too large.
MISRAC2004-12.9	Uses of unary minus on unsigned expressions were found.
MISRAC2004-12.10	Uses of the comma operator were found.
MISRAC2004-12.11	Found a constant unsigned integer expression that overflows.
MISRAC2004-12.12_a	Found a read access to a field of a union following a write access to a different field, which effectively re-interprets the bit pattern with a different type.
MISRAC2004-12.12_b	An expression was found that provides access to the bit representation of a floating-point variable.
MISRAC2004-12.13	Uses of the increment (++) and decrement (--) operators werew found mixed with other operators in an expression.
MISRAC2004-13.1	Assignment operators were found in expressions that yield a Boolean value.
MISRAC2004-13.2_a	Non-Boolean termination conditions were found in do ... while statements.
MISRAC2004-13.2_b	Non-boolean termination conditions were found in `for` loops.
MISRAC2004-13.2_c	Non-Boolean conditions were found in `if` statements.
MISRAC2004-13.2_d	Non-Boolean termination conditions were found in `while` statements.
MISRAC2004-13.2_e	Non-Boolean operands to the conditional (`? :`) operator were found.

Table 6: Summary of checks

Check	Synopsis
`MISRAC2004-13.3`	Floating-point comparisons using == or != were found.
`MISRAC2004-13.4`	Floating-point values were found 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 was found that is modified in the body of the loop.
`MISRAC2004-13.7_a`	A comparison using `==`, `<`, `<=`, `>`, or `>=` was found that always evaluates to true.
`MISRAC2004-13.7_b`	A comparison using `==`, `<`, `<=`, `>`, or `>=` was found that always evaluates to false.
`MISRAC2004-14.1`	A part of the application is not executed on any of the execution paths.
`MISRAC2004-14.2`	A statement was found that potentially contains no side effects.
`MISRAC2004-14.3`	There are stray semicolons on the same line as other code.
`MISRAC2004-14.4`	Uses of the goto statement were found.
`MISRAC2004-14.5`	Uses of the continue statement were found.
`MISRAC2004-14.6`	Multiple termination points were found in a loop.
`MISRAC2004-14.7`	More than one point of exit was found in a function, or an exit point before the end of the function.
`MISRAC2004-14.8_a`	There are missing braces in one or more do ... while statements.
`MISRAC2004-14.8_b`	There are missing braces in one or more `for` statements.
`MISRAC2004-14.8_c`	There are missing braces in one or more switch statements.
`MISRAC2004-14.8_d`	There are missing braces in one or more while statements.
`MISRAC2004-14.9`	There are missing braces in one or more `if`, `else`, or `else if` statements.

Table 6: Summary of checks

Check	Synopsis
`MISRAC2004-14.10`	One or more `if ... else if` constructs were found that are not terminated with an `else` clause.
`MISRAC2004-15.0`	Switch statements were found that do not conform to the MISRA C switch syntax.
`MISRAC2004-15.1`	Switch labels were found in nested blocks.
`MISRAC2004-15.2`	Non-empty switch cases were found that are not terminated by a break statement.
`MISRAC2004-15.3`	Switch statements were found without a default clause, or with a default clause that is not the final clause.
`MISRAC2004-15.4`	A switch expression was found that represents a value that is effectively Boolean.
`MISRAC2004-15.5`	Switch statements without case clauses were found.
`MISRAC2004-16.1`	Functions that are defined using ellipsis (...) notation were found.
`MISRAC2004-16.2_a`	Functions were found that call themselves directly.
`MISRAC2004-16.2_b`	Functions were found that call themselves indirectly.
`MISRAC2004-16.3`	Function prototypes were found that do not give all parameters a name.
`MISRAC2004-16.4`	The parameter names between the function declaration and definition does not match.
`MISRAC2004-16.5`	Functions were found that are declared with an empty () parameter list that does not form a valid prototype.
`MISRAC2004-16.7`	A function was found that does not modify one of its parameters.
`MISRAC2004-16.8`	For some execution paths, no return statement is executed in a function with a non-void return type.
`MISRAC2004-16.9`	One or more function addresses are taken without an explicit &.

Table 6: Summary of checks

Check	Synopsis
MISRAC2004-16.10	A return value for a library function that might return an error value is not used.
MISRAC2004-17.1_a	A direct access to a field of a struct was found, that uses an offset from the address of the struct.
MISRAC2004-17.1_b	Detected pointer arithmetic applied to a pointer that references a stack address.
MISRAC2004-17.1_c	Detected invalid pointer arithmetic with an automatic variable that is neither an array nor a pointer.
MISRAC2004-17.2	A subtraction was found between pointers that address elements of different arrays.
MISRAC2004-17.3	A relational operator was found applied to an object of pointer type that does not point into the same object.
MISRAC2004-17.4_a	Pointer arithmetic that is not array indexing was detected.
MISRAC2004-17.4_b	Array indexing was detected applied to an object defined as a pointer type.
MISRAC2004-17.5	One or more declarations of objects were found that contain more than two levels of pointer indirection.
MISRAC2004-17.6_a	Detected the return of a stack address.
MISRAC2004-17.6_b	Detected a stack address stored in a global pointer.
MISRAC2004-17.6_c	Detected a stack address stored in the field of a global struct.
MISRAC2004-17.6_d	Detected a stack address stored outside a function via a parameter.
MISRAC2004-18.1	Structs and unions were found that are used without being defined.
MISRAC2004-18.2	Assignments from one field of a union to another were found.
MISRAC2004-18.4	Unions were detected.
MISRAC2004-19.1	#include directives were found that are not first in the source file.

Table 6: Summary of checks

Check	Synopsis
MISRAC2004-19.2	There are illegal characters in header file names.
MISRAC2004-19.4	A macro definition was found that is not permitted.
MISRAC2004-19.5	A #define or #undef was found inside a block.
MISRAC2004-19.6	#undef directives were found.
MISRAC2004-19.7	Function-like macros were detected.
MISRAC2004-19.10	A macro parameter was not enclosed in parentheses or used as the operand of # or ##.
MISRAC2004-19.12	Multiple # or ## preprocessor operators were found in a macro definition.
MISRAC2004-19.13	Uses were found of the # and ## operators.
MISRAC2004-19.15	Header files were found without #include guards.
MISRAC2004-20.1	Detected a #define or #undef of a reserved identifier in the standard library.
MISRAC2004-20.2	One or more library functions are being overridden.
MISRAC2004-20.3_a	A parameter value (<=0) might cause a domain or range error.
MISRAC2004-20.3_b	A parameter value (<0) might cause a domain or range error.
MISRAC2004-20.3_c	A parameter value (==0) might cause a domain or range error.
MISRAC2004-20.3_d	A parameter value (>1) might cause domain or range error.
MISRAC2004-20.3_e	A parameter value (>=1) might cause domain or range error.
MISRAC2004-20.3_f	A parameter value (<-1) might cause a domain or range error.
MISRAC2004-20.3_g	A parameter value (<=-1) might cause a domain or range error.
MISRAC2004-20.3_h	A parameter value (>255) might cause a domain or range error.
MISRAC2004-20.3_i	A parameter value (min) might cause a domain or range error.

Table 6: Summary of checks

Check	Synopsis
MISRAC2004-20.4	Detected use of malloc, calloc, realloc, or free.
MISRAC2004-20.5	Detected use of the error indicator errno.
MISRAC2004-20.6	Detected use of the built-in function offsetof.
MISRAC2004-20.7	Detected use of setjmp.h.
MISRAC2004-20.8	Use of signal.h was detected.
MISRAC2004-20.9	Use of stdio.h was detected.
MISRAC2004-20.10	Use of the functions atof, atoi, atol, or atoll was detected.
MISRAC2004-20.11	Use of the functions abort, exit, getenv, or system was detected.
MISRAC2004-20.12	Use of the time.h functions was detected: asctime, clock, ctime, difftime, gmtime, localtime, mktime, strftime, or time.
MISRAC2012-Dir-4.3	Inline assembler statements were found that are not encapsulated in functions.
MISRAC2012-Dir-4.4	Code sections in comments were found where the comment ends with a ';', '{', or '}' character.
MISRAC2012-Dir-4.5	Identifiers in the same namespace, with overlapping visibility, should be typographically unambiguous.
MISRAC2012-Dir-4.6_a	The basic types char, int, short, long, double, and float are used without a typedef.
MISRAC2012-Dir-4.6_b	Typedefs of basic types were found with names that do not indicate the size or signedness.
MISRAC2012-Dir-4.7_a	Returned error information should be tested.
MISRAC2012-Dir-4.7_b	Returned error information should be tested.
MISRAC2012-Dir-4.7_c	Returned error information should be tested.
MISRAC2012-Dir-4.8	The implementation of a structure is unnecessarily exposed to a translation unit.
MISRAC2012-Dir-4.9	Function-like macros were detected.
MISRAC2012-Dir-4.10	Header files were found without #include guards.
MISRAC2012-Dir-4.11_a	A parameter value (<=0) might cause a domain or range error.

Table 6: Summary of checks

Check	Synopsis
MISRAC2012-Dir-4.11_b	A parameter value (<0) might cause a domain or range error.
MISRAC2012-Dir-4.11_c	A parameter value (==0) might cause a domain or range error.
MISRAC2012-Dir-4.11_d	A parameter value (>1) might cause domain or range error.
MISRAC2012-Dir-4.11_e	A parameter value (>=1) might cause domain or range error.
MISRAC2012-Dir-4.11_f	A parameter value (<-1) might cause a domain or range error.
MISRAC2012-Dir-4.11_g	A parameter value (<=-1) might cause a domain or range error.
MISRAC2012-Dir-4.11_h	A parameter value (>255) might cause a domain or range error.
MISRAC2012-Dir-4.11_i	A parameter value (min) might cause a domain or range error.
MISRAC2012-Dir-4.12	Dynamic memory allocation found.
MISRAC2012-Dir-4.13_b	Incorrect deallocation causes memory leak.
MISRAC2012-Dir-4.13_c	A file pointer is never closed.
MISRAC2012-Dir-4.13_d	A pointer is used after it has been freed.
MISRAC2012-Dir-4.13_e	A pointer is used after it has been freed.
MISRAC2012-Dir-4.13_f	A file resource is used after it has been closed.
MISRAC2012-Dir-4.13_g	A pointer is freed without having been allocated.
MISRAC2012-Dir-4.13_h	A struct field is deallocated without first having been allocated.
MISRAC2012-Rule-1.3_a	An expression resulting in `0` is used as a divisor.
MISRAC2012-Rule-1.3_b	A variable was found that is assigned the value `0`, and then used as a divisor.
MISRAC2012-Rule-1.3_c	A variable is used as a divisor after a successful comparison with `0`.
MISRAC2012-Rule-1.3_d	A variable used as a divisor is subsequently compared with `0`.
MISRAC2012-Rule-1.3_e	A value that is determined using interval analysis to be `0` is used as a divisor.

Table 6: Summary of checks

Check	Synopsis
MISRAC2012-Rule-1.3_f	An expression that might 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-1.3_i	Expressions found that depend on order of evaluation.
MISRAC2012-Rule-1.3_j	A variable is read before it is assigned a value.
MISRAC2012-Rule-1.3_k	A variable is read before it is assigned a value.
MISRAC2012-Rule-1.3_m	A function pointer is used in an invalid context.
MISRAC2012-Rule-1.3_n	The left-hand side of a right shift operation might be a negative value.
MISRAC2012-Rule-1.3_o	A pointer is used after it has been freed.
MISRAC2012-Rule-1.3_p	A pointer is used after it has been freed.
MISRAC2012-Rule-1.3_q	Might return an address on the stack.
MISRAC2012-Rule-1.3_r	A stack address is stored in a global pointer.
MISRAC2012-Rule-1.3_s	A stack address is stored outside a function via a parameter.
MISRAC2012-Rule-1.3_t	A call to memcpy or memmove causes the memory to overrun.
MISRAC2012-Rule-1.3_u	A call to memset causes a buffer overrun.
MISRAC2012-Rule-1.3_v	A call to strcpy causes a destination buffer overrun.
MISRAC2012-Rule-1.3_w	A call to strcat causes a destination buffer overrun.
MISRAC2012-Rule-2.1_a	A case statement within a switch statement cannot be reached.
MISRAC2012-Rule-2.1_b	A part of the application is never executed.
MISRAC2012-Rule-2.2_a	A statement potentially contains no side effects.
MISRAC2012-Rule-2.2_b	A field in a struct is assigned a non-trivial value that is never used.
MISRAC2012-Rule-2.2_c	A variable is assigned a value that is never used.
MISRAC2012-Rule-2.3	Unused type declaration.

Table 6: Summary of checks

Check	Synopsis
MISRAC2012-Rule-2.4	Unused tag declarations were found.
MISRAC2012-Rule-2.5	An unused macro declaration was found.
MISRAC2012-Rule-2.6	A function was found that contains an unused label declaration.
MISRAC2012-Rule-2.7	A function parameter is declared but not used.
MISRAC2012-Rule-3.1	The character sequences /* and // were found within a comment.
MISRAC2012-Rule-3.2	Line-splicing was found in // comments.
MISRAC2012-Rule-5.1	An external identifier was found that is not unique for the first 31 characters, but still not identical to another identifier.
MISRAC2012-Rule-5.2_c89	Identifier names were found that are not distinct in their first 31 characters from other names in the same scope.
MISRAC2012-Rule-5.2_c99	Identifier names were found that are not distinct in their first 63 characters from other names in the same scope.
MISRAC2012-Rule-5.3_c89	Identifier names were found that are not distinct in their first 31 characters from other names in an outer scope.
MISRAC2012-Rule-5.3_c99	Identifier names were found that are not distinct in their first 63 characters from other names in an outer scope.
MISRAC2012-Rule-5.4_c89	Macro names were found that are not distinct in their first 31 characters from their macro parameters or other macro names.
MISRAC2012-Rule-5.4_c99	Macro names were found 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 were found that are not distinct in their first 31 characters from macro names.
MISRAC2012-Rule-5.5_c99	Non-macro identifiers were found that are not distinct in their first 63 characters from macro names.

Table 6: Summary of checks

Check	Synopsis
`MISRAC2012-Rule-5.6`	A typedef with this name has already been declared.
`MISRAC2012-Rule-5.7`	A class, struct, union, or enum declaration clashes with a previous declaration.
`MISRAC2012-Rule-5.8`	One or more external identifier names were found that are not unique.
`MISRAC2012-Rule-5.9`	An internal identifier name was found that is not unique.
`MISRAC2012-Rule-6.1`	Bitfields of plain int type were found.
`MISRAC2012-Rule-6.2`	Signed single-bit bitfields (excluding anonymous fields) were found.
`MISRAC2012-Rule-7.1`	Octal integer constants are used.
`MISRAC2012-Rule-7.2`	There are unsigned integer constants without a `U` suffix.
`MISRAC2012-Rule-7.3`	The lower case character `l` was found used as a suffix on numeric constants.
`MISRAC2012-Rule-7.4_a`	A string literal was found assigned to a variable that is not declared as constant.
`MISRAC2012-Rule-7.4_b`	Part of a string literal was found that is modified via the array subscript operator [].
`MISRAC2012-Rule-8.1`	An object or function of the type `int` is declared or defined, but its type is not explicitly stated.
`MISRAC2012-Rule-8.2_a`	There are functions declared with an empty () parameter list that does not form a valid prototype.
`MISRAC2012-Rule-8.2_b`	Function prototypes were found with unnamed parameters.
`MISRAC2012-Rule-8.3_b`	Multiple declarations of an object or function were found that use different names and type qualifiers.
`MISRAC2012-Rule-8.4`	An extern definition is missing a compatible declaration.
`MISRAC2012-Rule-8.5_a`	Multiple declarations of the same external object or function were found.

Table 6: Summary of checks

Check	Synopsis
MISRAC2012-Rule-8.5_b	Multiple declarations of the same external object or function were found.
MISRAC2012-Rule-8.6	Multiple definitions or no definition were found for an external object or function.
MISRAC2012-Rule-8.7	An externally linked object or function was found referenced in only one translation unit.
MISRAC2012-Rule-8.9_a	A global object was found that is only referenced from a single function.
MISRAC2012-Rule-8.9_b	A global object was found that is only referenced from a single function.
MISRAC2012-Rule-8.10	Inline functions were found that are not declared as static.
MISRAC2012-Rule-8.11	One or more external arrays are declared without their size being stated explicitly or defined implicitly by initialization.
MISRAC2012-Rule-8.12	A duplicated implicit enumeration constant was found.
MISRAC2012-Rule-8.13	A pointer was found that is not const-qualified.
MISRAC2012-Rule-8.14	The `restrict` type qualifier was found used in function parameters.
MISRAC2012-Rule-9.1_a	A possible dereference of an uninitialized or `NULL` pointer was found.
MISRAC2012-Rule-9.1_b	Read accesses from local buffers were found that are not preceded by writes.
MISRAC2012-Rule-9.1_c	On all execution paths, there is a struct that 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	On all execution paths, there is a variable that is read before it is assigned a value.
MISRAC2012-Rule-9.1_f	A variable was found that might read before it is assigned a value.
MISRAC2012-Rule-9.2	An initializer for an aggregate or union was found that is not enclosed in braces.
MISRAC2012-Rule-9.3	Arrays were found that are partially initialized.

Table 6: Summary of checks

Check	Synopsis
MISRAC2012-Rule-9.4	An object field was found that is initialized more than once. The last initialization will overwrite previous value(s).
MISRAC2012-Rule-9.5_a	Arrays, initialized with designated initializers but with no fixed length, were found.
MISRAC2012-Rule-9.5_b	A flexible array member was found that is initialized with a designated initializer.
MISRAC2012-Rule-10.1_R2	An operand was found that is not of essentially Boolean type, despite being interpreted as a Boolean value.
MISRAC2012-Rule-10.1_R3	An operand was found that is of essentially Boolean type, despite being interpreted as a numeric value.
MISRAC2012-Rule-10.1_R4	An operand was found that is of essentially character type, despite being interpreted as a numeric value.
MISRAC2012-Rule-10.1_R5	An operand that is of essentially enum type is used in an arithmetic operation, because an enum object uses an implementation-defined integer type.
MISRAC2012-Rule-10.1_R6	Shift and bitwise operations were found performed on operands of essentially signed type.
MISRAC2012-Rule-10.1_R7	The right-hand operand of a shift operator is not of essentially unsigned type.
MISRAC2012-Rule-10.1_R8	An operand of essentially unsigned typed is used as the operand to the unary minus operator.
MISRAC2012-Rule-10.2	Expressions of essentially character type were found used inappropriately in addition and subtraction operations.
MISRAC2012-Rule-10.3	The value of an expression was found assigned to an object with a narrower essential type or a different essential type category.
MISRAC2012-Rule-10.4_a	Operands of an operator in which the usual arithmetic conversions are performed were found, that do not have the same essential type category.

Table 6: Summary of checks

Check	Synopsis
MISRAC2012-Rule-10.4_b	The second and third operands of the ternary operator do not have the same essential type category.
MISRAC2012-Rule-10.5	A value of an expression was found that is cast to an inappropriate essential type.
MISRAC2012-Rule-10.6	The value of a composite expression is assigned to an object with wider essential type.
MISRAC2012-Rule-10.7	An operator in which the usual arithmetic conversions are performed was found, where a composite expression is used as one of the operands, but the other operand is of wider essential type.
MISRAC2012-Rule-10.8	A composite expression was found whose value is cast to a different essential type category or a wider essential type.
MISRAC2012-Rule-11.1	Conversion between a pointer to a function and another type were found.
MISRAC2012-Rule-11.2	A conversion from or to an incomplete type pointer was found.
MISRAC2012-Rule-11.3	A pointer to object type is cast to a pointer to a different object type.
MISRAC2012-Rule-11.4	A cast between a pointer type and an integral type was found.
MISRAC2012-Rule-11.5	A conversion from a pointer to void into a pointer to object was found.
MISRAC2012-Rule-11.6	A conversion between a pointer to void and an arithmetic type was found.
MISRAC2012-Rule-11.7	A cast between a pointer to object and a non-integer arithmetic type was found.
MISRAC2012-Rule-11.8	A cast that removes a const or volatile qualification was found.
MISRAC2012-Rule-11.9	An integer constant was found where the NULL macro should be.
MISRAC2012-Rule-12.1	Implicit operator precedence was detected, without parenthesis to make it explicit.
MISRAC2012-Rule-12.2	Out of range shifts were found

Table 6: Summary of checks

Check	Synopsis
`MISRAC2012-Rule-12.3`	There are uses of the comma operator.
`MISRAC2012-Rule-13.1`	The initalization list of an array contains side effects.
`MISRAC2012-Rule-13.2_a`	Expressions that depend on order of evaluation were found.
`MISRAC2012-Rule-13.2_b`	There are multiple read accesses with volatile-qualified type within one and the same sequence point.
`MISRAC2012-Rule-13.2_c`	There are multiple write accesses with volatile-qualified type within one and the same sequence point.
`MISRAC2012-Rule-13.3`	The increment (++) and decrement (--) operators are being used mixed with other operators in an expression.
`MISRAC2012-Rule-13.4_a`	An assignment might be mistakenly used as the condition for an `if`, `for`, `while`, or `do` statement.
`MISRAC2012-Rule-13.4_b`	Assignments were found in a sub-expression.
`MISRAC2012-Rule-13.5`	There are right-hand operands of && or || operators that contain side effects.
`MISRAC2012-Rule-13.6`	The operand of the sizeof operator contains an expression that has potential side effects.
`MISRAC2012-Rule-14.1_a`	Floating-point values were found in the controlling expression of a for statement.
`MISRAC2012-Rule-14.1_b`	A variable of essentially float type that is used in the loop condition, is then modified in the loop body.
`MISRAC2012-Rule-14.2`	A malformed `for` loop was found.
`MISRAC2012-Rule-14.3_a`	The condition in an if, for, while, do-while, or ternary operator will always be true.
`MISRAC2012-Rule-14.3_b`	The condition in if, for, while, do-while, or ternary operator will never be true.
`MISRAC2012-Rule-14.4_a`	Non-Boolean termination conditions were found in do ... while statements.
`MISRAC2012-Rule-14.4_b`	Non-Boolean termination conditions were found in `for` loops.

Table 6: Summary of checks

Check	Synopsis
MISRAC2012-Rule-14.4_c	Non-Boolean conditions were found in `if` statements.
MISRAC2012-Rule-14.4_d	Non-Boolean termination conditions were found in while statements.
MISRAC2012-Rule-15.1	Uses of the goto statement were found.
MISRAC2012-Rule-15.2	A goto statement is declared after the destination label.
MISRAC2012-Rule-15.3	The destination of a goto statement is a nested code block.
MISRAC2012-Rule-15.4	One or more iteration statements are terminated by more than one break or goto statements.
MISRAC2012-Rule-15.5	One or more functions have multiple exit points or an exit point that is not at the end of the function.
MISRAC2012-Rule-15.6_a	There are missing braces in `do ... while` statements.
MISRAC2012-Rule-15.6_b	There are missing braces in `for` statements.
MISRAC2012-Rule-15.6_c	There are missing braces in `if`, `else`, or `else if` statements.
MISRAC2012-Rule-15.6_d	There are missing braces in `switch` statements.
MISRAC2012-Rule-15.6_e	There are missing braces in `while` statements.
MISRAC2012-Rule-15.7	`If ... else if` constructs that are not terminated with an `else` clause were detected.
MISRAC2012-Rule-16.1	Detected switch statements that do not conform to the MISRA C switch syntax.
MISRAC2012-Rule-16.2	Switch labels were found in nested blocks.
MISRAC2012-Rule-16.3	Non-empty switch cases were found that are not terminated by a break.
MISRAC2012-Rule-16.4	Switch statements without a default clause were found.
MISRAC2012-Rule-16.5	A switch was found whose default label is neither the first nor the last label of the switch.
MISRAC2012-Rule-16.6	Switch statements without case clauses were found.

Table 6: Summary of checks

Check	Synopsis
`MISRAC2012-Rule-16.7`	A switch expression was found that represents a value that is effectively Boolean.
`MISRAC2012-Rule-17.1`	Inclusion of the stdarg header file was detected.
`MISRAC2012-Rule-17.2_a`	There are functions that call themselves directly.
`MISRAC2012-Rule-17.2_b`	There are functions that call themselves indirectly.
`MISRAC2012-Rule-17.3`	Functions are used without prototyping.
`MISRAC2012-Rule-17.4`	For some execution paths, no return statement is executed in a function with a non-`void` return type.
`MISRAC2012-Rule-17.5`	A function call is made with the wrong array type argument.
`MISRAC2012-Rule-17.6`	There are array parameters with the `static` keyword between the [].
`MISRAC2012-Rule-17.7`	There are unused function return values (other than overloaded operators).
`MISRAC2012-Rule-17.8`	A function parameter was found that is modified.
`MISRAC2012-Rule-18.1_a`	An array access is out of bounds.
`MISRAC2012-Rule-18.1_b`	An array access might 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-18.2`	A subtraction was found between pointers that address elements of different arrays.
`MISRAC2012-Rule-18.3`	A relational operator was found applied to an object of pointer type that does not point into the same object.
`MISRAC2012-Rule-18.4`	A +, -, +=, or -= operator was found applied to an expression of pointer type.
`MISRAC2012-Rule-18.5`	Declarations that contain more than two levels of pointer indirection have been found.
`MISRAC2012-Rule-18.6_a`	Might return address on the stack.

Table 6: Summary of checks

Check	Synopsis
MISRAC2012-Rule-18.6_b	A stack address is stored in a global pointer.
MISRAC2012-Rule-18.6_c	A stack address is stored in the field of a global struct.
MISRAC2012-Rule-18.6_d	A stack address is stored outside a function via a parameter.
MISRAC2012-Rule-18.7	Flexible array members are declared.
MISRAC2012-Rule-18.8	There are arrays declared with a variable length.
MISRAC2012-Rule-19.1	Assignments from one field of a union to another were found.
MISRAC2012-Rule-19.2	Unions were found.
MISRAC2012-Rule-20.1	#include directives were found that are not first in the source file.
MISRAC2012-Rule-20.2	Illegal characters were found in the names of header files.
MISRAC2012-Rule-20.4_c89	A macro was found defined with the same name as a keyword.
MISRAC2012-Rule-20.4_c99	A macro was found defined with the same name as a keyword.
MISRAC2012-Rule-20.5	Found occurrences of #undef.
MISRAC2012-Rule-20.7	An expansion of macro parameters was found that is not enclosed in parentheses.
MISRAC2012-Rule-20.10	# and ## operators were found in macro definitions.
MISRAC2012-Rule-21.1	Detected a #define or #undef of a reserved identifier in the standard library.
MISRAC2012-Rule-21.2	One or more library functions are being overridden.
MISRAC2012-Rule-21.3	Uses of malloc, calloc, realloc, or free were found.
MISRAC2012-Rule-21.4	Found uses of setjmp.h.
MISRAC2012-Rule-21.5	Uses of signal.h were found.
MISRAC2012-Rule-21.6	Uses of stdio.h were found.
MISRAC2012-Rule-21.7	Uses of atof, atoi, atol, and atoll were found.
MISRAC2012-Rule-21.8	Uses of abort, exit, getenv, and system were found.

Table 6: Summary of checks

Check	Synopsis
MISRAC2012-Rule-21.9	Uses of the library functions bsearch and qsort in stdlib.h were found.
MISRAC2012-Rule-21.10	Use of the following time.h functions was found: asctime, clock, ctime, difftime, gmtime, localtime, mktime, strftime, and time.
MISRAC2012-Rule-21.11	Use of the standard header file tgmath.h was found.
MISRAC2012-Rule-21.12_a	The exception-handling features of <fenv.h> are used.
MISRAC2012-Rule-21.12_b	Macros are used in <fenv.h>.
MISRAC2012-Rule-22.1_a	A memory leak due to incorrect deallocation was detected.
MISRAC2012-Rule-22.1_b	A file pointer is never closed.
MISRAC2012-Rule-22.2_a	A memory location is freed 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 might be freed.
MISRAC2012-Rule-22.3	A file was found that is open for read and write access at the same time on different streams.
MISRAC2012-Rule-22.4	A file opened as read-only is written to.
MISRAC2012-Rule-22.5_a	A pointer to a FILE object is dereferenced.
MISRAC2012-Rule-22.5_b	A file pointer was found that is implicitly dereferenced by a library function.
MISRAC2012-Rule-22.6	A file pointer was found that is used after it has been closed.
MISRAC++2008-0-1-1	A part of the application is never executed.
MISRAC++2008-0-1-2_a	The condition in if, for, while, do-while statement sequences and the ternary operator is always met.
MISRAC++2008-0-1-2_b	The condition in if, for, while, do-while statement sequences and the ternary operator will never be met.
MISRAC++2008-0-1-2_c	A case statement within a switch statement is unreachable.

Table 6: Summary of checks

Check	Synopsis
MISRAC++2008-0-1-3	A variable is never read or written during execution.
MISRAC++2008-0-1-4_a	A variable is only used once.
MISRAC++2008-0-1-4_b	A global variable is only used once.
MISRAC++2008-0-1-6	A variable is assigned a value that is never used.
MISRAC++2008-0-1-7	There are unused function return values (excluding overloaded operators)
MISRAC++2008-0-1-8	There are functions with no effect. A function with no return type and no side effects effectively does nothing.
MISRAC++2008-0-1-9	A part of the application is never executed.
MISRAC++2008-0-1-11	A function parameter is declared but not used.
MISRAC++2008-0-2-1	There are assignments from one field of a union to another.
MISRAC++2008-0-3-2	The return value for a library function that might return an error value is not used.
MISRAC++2008-2-7-1	Detected /* inside comments
MISRAC++2008-2-7-2	Commented-out code has been detected. (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	Commented-out code has been detected. (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++2008-2-10-1	Two identifiers have names that can be confused with each other.
MISRAC++2008-2-10-2 (C++ only)	There are identifier names that are not distinct from other names in an outer scope.
MISRAC++2008-2-10-3	A typedef with this name has already been declared.
MISRAC++2008-2-10-4	A class, struct, union, or enum declaration clashes with a previous declaration.

Table 6: Summary of checks

Check	Synopsis
MISRAC++2008-2-10-5	An identifier is used that might clash with another static identifier.
MISRAC++2008-2-10-6 (C++ only)	There is a clash with type names.
MISRAC++2008-2-13-2	Octal integer constants are used.
MISRAC++2008-2-13-3	There are unsigned integer constants without a U suffix.
MISRAC++2008-2-13-4_a	Suffixes on floating-point constants are lower case.
MISRAC++2008-2-13-4_b	Suffixes on integer constants are lower case.
MISRAC++2008-3-1-1	Non-inline functions have been defined in header files.
MISRAC++2008-3-1-3	One or more external arrays are declared without their size being stated explicitly or defined implicitly by initialization.
MISRAC++2008-3-9-2	There are uses of the basic types char, int, short, long, double, and float without a typedef.
MISRAC++2008-3-9-3	An expression provides access to the bit-representation of a floating-point variable.
MISRAC++2008-4-5-1	Arithmetic operators are used on boolean operands.
MISRAC++2008-4-5-2	Unsafe operators are used on variables of enumeration type.
MISRAC++2008-4-5-3	Arithmetic is performed on objects of type plain char, without an explicit signed or unsigned qualifier.
MISRAC++2008-5-0-1_a	There are expressions that depend on the order of evaluation.
MISRAC++2008-5-0-1_b	There are more than one read access with volatile-qualified type within a single sequence point.
MISRAC++2008-5-0-1_c	There are more than one modification access with volatile-qualified type within a single sequence point.
MISRAC++2008-5-0-2	Parentheses to avoid implicit operator precedence are missing.

Table 6: Summary of checks

Check	Synopsis
MISRAC++2008-5-0-3	One or more cvalue expressions have been implicitly converted to a different underlying type.
MISRAC++2008-5-0-4	One or more implicit integral conversions have been found that change the signedness of the underlying type.
MISRAC++2008-5-0-5	One or more implicit floating-integral conversions were found.
MISRAC++2008-5-0-6 (C++ only)	One or more implicit integral or floating-point conversion were found that reduce the size of the underlying type.
MISRAC++2008-5-0-7	One or more explicit floating-integral conversions of a cvalue expression were found.
MISRAC++2008-5-0-8	One or more explicit integral or floating-point conversions were found that increase the size of the underlying type of a cvalue expression.
MISRAC++2008-5-0-9	One or more explicit integral conversions were found that change the signedness of the underlying type of a cvalue expression.
MISRAC++2008-5-0-10	A bitwise operation on unsigned char or unsigned short was found, that was not immediately cast to this type to ensure consistent truncation.
MISRAC++2008-5-0-13_a	Non-Boolean termination conditions were found in do ... while statements.
MISRAC++2008-5-0-13_b	Non-boolean termination conditions were found in `for` loops.
MISRAC++2008-5-0-13_c	Non-boolean conditions were found in if statements.
MISRAC++2008-5-0-13_d	Non-boolean termination conditions were found in while statements.
MISRAC++2008-5-0-14	Non-boolean operands to the conditional (? :) operator were found.
MISRAC++2008-5-0-15_a	Pointer arithmetic that is not array indexing was found.
MISRAC++2008-5-0-15_b	Array indexing applied to objects not defined as an array type was found.

Table 6: Summary of checks

Check	Synopsis
MISRAC++2008-5-0-16_a	Pointer arithmetic applied to a pointer that references a stack address was found.
MISRAC++2008-5-0-16_b	Invalid pointer arithmetic with an automatic variable that is neither an array nor a pointer was found.
MISRAC++2008-5-0-16_c	An array access is out of bounds.
MISRAC++2008-5-0-16_d	An array access might be out of bounds for some execution paths.
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 might be used outside the array bounds.
MISRAC++2008-5-0-19	Declarations that contain more than two levels of pointer indirection have been found.
MISRAC++2008-5-0-21	Applications of bitwise operators to signed operands were found.
MISRAC++2008-5-2-4 (C++ only)	Old style casts (other than void casts) were found.
MISRAC++2008-5-2-5	Casts that remove a const or volatile qualification were found.
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.
MISRAC++2008-5-2-7	A pointer to object type is cast to a pointer to a different object type.
MISRAC++2008-5-2-9	A cast from a pointer type to an integral type was found.
MISRAC++2008-5-2-10	The increment (++) and decrement (--) operators are being used mixed with other operators in an expression.
MISRAC++2008-5-2-11_a (C++ only)	Overloaded && and || operators were found.
MISRAC++2008-5-2-11_b (C++ only)	Overloaded comma operators were found.
MISRAC++2008-5-3-1	Operands of the logical operators (&&, ||, and !) were found that are not of type bool.
MISRAC++2008-5-3-2_a	Uses of unary minus on unsigned expressions were found.

Table 6: Summary of checks

Check	Synopsis
MISRAC++2008-5-3-2_b	Uses of unary minus on unsigned expressions were found.
MISRAC++2008-5-3-3 (C++ only)	Occurances of overloaded & operators were found.
MISRAC++2008-5-3-4	There are sizeof expressions that contain side effects.
MISRAC++2008-5-8-1	Possible out-of-range shifts were found.
MISRAC++2008-5-14-1	There are right-hand operands of && or || operators that contain side effects.
MISRAC++2008-5-18-1	There are uses of the comma operator.
MISRAC++2008-5-19-1	A constant unsigned integer expression overflows.
MISRAC++2008-6-2-1	One or more assignment operators are used in sub-expressions.
MISRAC++2008-6-2-2	There are floating-point comparisons that use the == or != operators.
MISRAC++2008-6-2-3	There are stray semicolons on the same line as other code.
MISRAC++2008-6-3-1_a	There are missing braces in `do ... while` statements.
MISRAC++2008-6-3-1_b	There are missing braces in `for` statements.
MISRAC++2008-6-3-1_c	There are missing braces in `switch` statements.
MISRAC++2008-6-3-1_d	There are missing braces in `while` statements.
MISRAC++2008-6-4-1	There are missing braces in `if`, `else`, or `else if` statements.
MISRAC++2008-6-4-2	`If ... else if` constructs that are not terminated with an `else` clause were detected.
MISRAC++2008-6-4-3	Detected switch statements that do not conform to the MISRA C++ switch syntax.
MISRAC++2008-6-4-4	Switch labels were found in nested blocks.
MISRAC++2008-6-4-5	Non-empty switch cases were found that are not terminated by a break.

Table 6: Summary of checks

Check	Synopsis
MISRAC++2008-6-4-6	Switch statements without a default clause, or with a default clause that is not the final clause, were found.
MISRAC++2008-6-4-7	A switch expression was found that represents a value that is effectively Boolean.
MISRAC++2008-6-4-8	One or more switch statements without a case clause were found.
MISRAC++2008-6-5-1_a	Floating-point values were found in the controlling expression of a for statement.
MISRAC++2008-6-5-2	A loop counter was found that might not match the loop condition test.
MISRAC++2008-6-5-3	A `for` loop counter variable was found that is modified in the body of the loop.
MISRAC++2008-6-5-4	A potentially inconsistent loop counter modification was found.
MISRAC++2008-6-5-6	A non-boolean variable was detected that is modified in the loop and used as loop condition.
MISRAC++2008-6-6-1	The destination of a goto statement is a nested code block.
MISRAC++2008-6-6-2	A goto statement is declared after the destination label.
MISRAC++2008-6-6-4	One or more loops have more than one termination point.
MISRAC++2008-6-6-5	One or more functions have multiple exit points or an exit point that is not at the end of the function.
MISRAC++2008-7-1-1	A local variable that is not modified after its initialization is not `const` qualified.
MISRAC++2008-7-1-2	A parameter in a function that is not modified by the function is not `const` qualified.
MISRAC++2008-7-2-1	There are conversions to enum type that are out of range of the enumeration.
MISRAC++2008-7-4-3	There are inline assembler 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.

Table 6: Summary of checks

Check	Synopsis
`MISRAC++2008-7-5-1_b`	A function might return an address on the stack.
`MISRAC++2008-7-5-2_a`	Detected a stack address stored in a global pointer.
`MISRAC++2008-7-5-2_b`	Detected a stack address in the field of a global struct.
`MISRAC++2008-7-5-2_c`	Detected a stack address stored in a parameter of pointer or array type.
`MISRAC++2008-7-5-2_d (C++ only)`	Detected a stack address stored via a reference parameter.
`MISRAC++2008-7-5-4_a`	There are functions that call themselves directly.
`MISRAC++2008-7-5-4_b`	There are functions that call themselves indirectly.
`MISRAC++2008-8-0-1`	There are declarations that contain more than one variable or constant each.
`MISRAC++2008-8-4-1`	There are functions defined using the ellipsis (...) notation.
`MISRAC++2008-8-4-3`	For some execution paths, no return statements are executed in functions with a non-void return type.
`MISRAC++2008-8-4-4`	The addresses of one or more functions are taken without an explicit `&`.
`MISRAC++2008-8-5-1_a`	In all execution paths, variables are read before they are assigned a value.
`MISRAC++2008-8-5-1_b`	In some execution paths, variables might be read before they are assigned a value.
`MISRAC++2008-8-5-1_c`	One or more uninitialized or NULL pointers are dereferenced.
`MISRAC++2008-8-5-2`	There are one or more non-zero array initializations that do 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 return non-const handles to members.
`MISRAC++2008-9-5-1`	Unions were found.
`MISRAC++2008-9-6-2`	Bitfields of plain int type were found.

Table 6: Summary of checks

Check	Synopsis
MISRAC++2008-9-6-3	Bitfields of plain int type were found.
MISRAC++2008-9-6-4	Signed single-bit bitfields (excluding anonymous fields) were found.
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)	Constructors that can be called with a single argument of fundamental type are not declared `explicit`.
MISRAC++2008-15-0-2	Throw of exceptions by pointer.
MISRAC++2008-15-1-2	Throw of NULL integer constant.
MISRAC++2008-15-1-3 (C++ only)	Unsafe rethrow of exception.
MISRAC++2008-15-3-1 (C++ only)	There are exceptions thrown without a handler in some call paths that lead to that point.
MISRAC++2008-15-3-2 (C++ only)	There are no default exception handlers for try.
MISRAC++2008-15-3-3 (C++ only)	One or more exception handlers in a constructor or destructor accesses a non-static member variable that might not exist.
MISRAC++2008-15-3-4 (C++ only)	There are calls to functions that are explicitly declared to throw an exception type that are not handled (or declared as thrown) by the caller.
MISRAC++2008-15-3-5 (C++ only)	Exception objects are caught by value, not by reference.
MISRAC++2008-15-5-1 (C++ only)	An exception is thrown, or might be thrown, in a class destructor.
MISRAC++2008-16-0-3	Found occurrences of #undef.
MISRAC++2008-16-0-4	Definitions of function-like macros were found.
MISRAC++2008-16-2-2 (C++ only)	Definitions of macros that are not include guards were found.
MISRAC++2008-16-2-3	Header files without #include guards were found.
MISRAC++2008-16-2-4	There are illegal characters in header file names.
MISRAC++2008-16-2-5	There are illegal characters in header file names.

Table 6: Summary of checks

Check	Synopsis
MISRAC++2008-16-3-1	There are multiple # or ## operators in a macro definition.
MISRAC++2008-16-3-2	# and ## operators were found in macro definitions.
MISRAC++2008-17-0-1	Detected a #define or #undef of a reserved identifier in the standard library.
MISRAC++2008-17-0-3	One or more library functions are being overridden.
MISRAC++2008-17-0-5	Found uses of setjmp.h.
MISRAC++2008-18-0-1 (C++ only)	C library includes were found.
MISRAC++2008-18-0-2	Uses of atof, atoi, atol and atoll were found.
MISRAC++2008-18-0-3	Uses of abort, exit, getenv, and system were found.
MISRAC++2008-18-0-4	Uses of time.h functions: asctime, clock, ctime, difftime, gmtime, localtime, mktime, strftime, and time were found.
MISRAC++2008-18-0-5	Uses of strcpy, strcmp, strcat, strchr, strspn, strcspn, strpbrk, strrchr, strstr, strtok, or strlen were found.
MISRAC++2008-18-2-1	Uses of the built-in function offsetof were found.
MISRAC++2008-18-4-1	Uses of malloc, calloc, realloc, or free were found.
MISRAC++2008-18-7-1	Uses of signal.h were found.
MISRAC++2008-19-3-1	Uses of errno were found.
MISRAC++2008-27-0-1	Uses of stdio.h were found.

Table 6: Summary of checks

Descriptions of checks

The following section gives detailed reference information about each check.

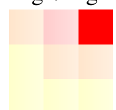
ARR-inv-index-pos

Synopsis An 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. This check is identical to MISRAC++2008-5-0-16_d, MISRAC2012-Rule-18.1_b

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

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:

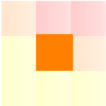
```
int cond;

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. This check is identical to MISRAC++2008-5-0-16_f, MISRAC2012-Rule-18.1_d

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

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

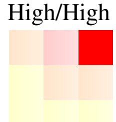
A pointer to an array is used outside the array bounds.

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. This check is identical to MISRAC++2008-5-0-16_e, MISRAC2012-Rule-18.1_c

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

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

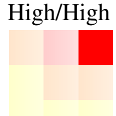
An array access is out of bounds.

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. This check is identical to MISRAC++2008-5-0-16_c, MISRAC2012-Rule-18.1_a

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

Code examples

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

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

  a[7] = 0;  //7 is out of bounds, since
             //a only has an interval of [0,3]
  return 0;
}
```

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

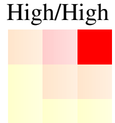
An array is accessed with a negative subscript value.

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

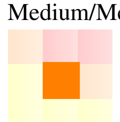
An array is indexed with an uninitialized variable

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

CWE 665

Improper Initialization

CWE 457

Use of Uninitialized Variable

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

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. This check is identical to MISRAC2004-13.3, MISRAC++2008-6-2-2

Coding standards

CERT FLP00-C

Understand the limitations of floating point numbers

CERT FLP35-CPP

Take granularity into account when comparing floating point values

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

CWE 570

Expression is Always False

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

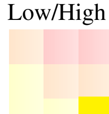
An unsigned value is compared to see whether it is greater than or equal to `0`.

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

CWE 571

Expression is Always True

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

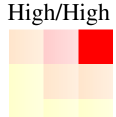
A variable is assigned the value 0, then used as a divisor.

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. This check is identical to MISRAC2004-1.2_d, MISRAC2012-Rule-1.3_b

Coding standards

CERT INT33-C

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

CWE 369

Divide By Zero

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;
}
```

ATH-div-0-cmp-aft

Synopsis

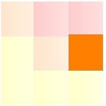
After a successful comparison with `0`, a variable is used as a divisor.

Enabled by default

No

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. This check is identical to MISRAC2004-1.2_e, MISRAC2012-Rule-1.3_c

Coding standards

CERT INT33-C

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

CWE 369

Divide By Zero

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. This check is identical to MISRAC2004-1.2_f, MISRAC2012-Rule-1.3_d

Coding standards

CERT INT33-C

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

CWE 369

> Divide By Zero

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

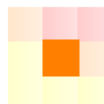
Interval analysis has found a value that is `0` and used as a divisor.

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. This check is identical to MISRAC2004-1.2_g, MISRAC2012-Rule-1.3_e

Coding standards

CERT INT33-C

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

CWE 369

Divide By Zero

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 */
}
```

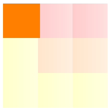
ATH-div-0-pos

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. This check is identical to MISRAC2004-1.2_h, MISRAC2012-Rule-1.3_f

Coding standards — CERT INT33-C

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

CWE 369

Divide By Zero

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
}
```

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`. This check is identical to MISRAC2004-1.2_i, MISRAC2012-Rule-1.3_g

Coding standards

CWE 369

Divide By Zero

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`. This check is identical to MISRAC2004-1.2_j, MISRAC2012-Rule-1.3_h

Coding standards

CWE 369

Divide By Zero

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

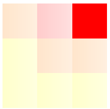
An expression that results in `0` is used as a divisor.

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. This check is identical to MISRAC2004-1.2_c, MISRAC2012-Rule-1.3_a

Coding standards

CERT INT33-C

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

CWE 369

Divide By Zero

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
}
```

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

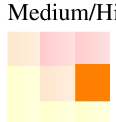
Deprecated operation on `bool`.

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

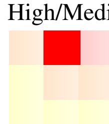
The size of memory passed to malloc to allocate overflows.

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

CWE 122

Heap-based Buffer Overflow

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 680

Integer Overflow to Buffer Overflow

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;
}
```

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

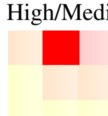
An arithmetic overflow is caused by an allocation using new[].

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

CWE 122

Heap-based Buffer Overflow

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 680

Integer Overflow to Buffer Overflow

Code examples

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

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

void example(void) {
  unsigned int b = (UINT_MAX / 4) + 1;
  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

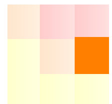
An expression is implicitly converted to a narrower type, resulting in an overflow or underflow of its value.

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

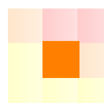
Out of range shifts were found.

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. This check is identical to MISRAC2004-12.8, MISRAC++2008-5-8-1, MISRAC2012-Rule-12.2

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

Code examples

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

```
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;
}
```

ATH-shift-neg

Synopsis

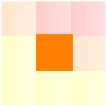
The left-hand side of a right shift operation might be a negative value.

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

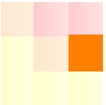
Multiplying `sizeof` by `sizeof`.

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

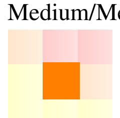
Old style casts (other than void casts) are used

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. This check is identical to MISRAC++2008-5-2-4

Coding standards

CERT EXP05-CPP

Do not use C-style casts

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

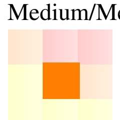
Exception objects are caught by value

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's 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. This check is identical to MISRAC++2008-15-3-5

Coding standards

CERT ERR09-CPP

Throw anonymous temporaries and catch by reference

Code examples

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

```
typedef char char_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:

```
typedef char char_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. This check is identical to MISRAC++2008-15-3-3

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 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

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 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. This check is identical to MISRAC2004-2.3, MISRAC++2008-2-7-1

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) {
  /* 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-member-ret (C++ only)

Synopsis

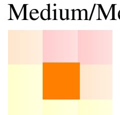
A member function qualified as `const` returns a pointer member variable.

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. This check is identical to MISRAC++2008-9-3-1

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 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;
};
```

COP-alloc-ctor (C++ only)

Synopsis

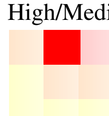
A class member is deallocated in the class' destructor, but not allocated in a constructor or assignment operator.

Enabled by default

No

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

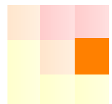
Assignment operator does not check for self-assignment before allocating member functions

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

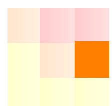
There is no assignment operator defined for a class whose destructor deallocates memory.

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:

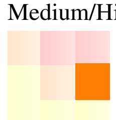
```
class MyClass{
  int* p;

  ~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

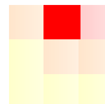
A class member has memory allocated in a constructor or an assignment operator, that is not released in the destructor.

Enabled by default

No

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

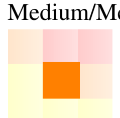
An exception is thrown, or might be thrown, in a class destructor.

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. This check is identical to MISRAC++2008-15-5-1

Coding standards

CERT ERR33-CPP

Destructors must not throw exceptions

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;
};
```

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

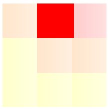
A class which dynamically allocates memory in its copy control functions does not have a destructor.

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

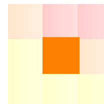
Data members are initialized with other data members that are in the same initialization list.

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

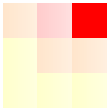
An initializer list reads the values of still uninitialized members.

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

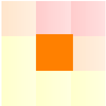
A member of a class is not initialized in one of the class constructors.

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
};
```

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

Synopsis

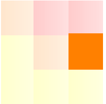
A virtual member function is called in a class constructor.

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. This check is identical to MISRAC++2008-12-1-1_a

Coding standards

CERT OOP30-CPP

Do not invoke virtual functions from constructors or destructors

Code examples

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

```
#include <iostream>

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

class B: public A {
public:
  virtual void f() const { std::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>

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

class B: public A {
public:
  virtual void f() const { std::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. This check is identical to MISRAC++2008-12-1-3

Coding standards

CERT OOP32-CPP

Ensure that single-argument constructors are marked "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

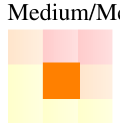
An exception is thrown, or might be thrown, in an overloaded `delete` or `delete[]` operator.

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;
};
```

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

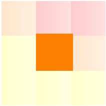
A pointer to void is used in `delete`, causing the destructor not to be called.

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

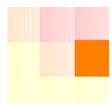
A virtual member function is called in a class destructor.

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. This check is identical to MISRAC++2008-12-1-1_b

Coding standards

CERT OOP30-CPP

Do not invoke virtual functions from constructors or destructors

Code examples

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

```
#include <iostream>

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

class B: public A {
public:
  virtual void f() const { std::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:

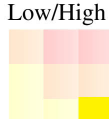
```
#include <iostream>

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

class B: public A {
public:
  virtual void f() const { std::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:

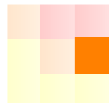
```
#include <stdlib.h>

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>

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

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

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

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

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

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

Synopsis

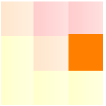
Member functions return non-`const` handles to members.

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. This check is identical to MISRAC++2008-9-3-2

Coding standards

CERT OOP35-CPP

Do not return references to private 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

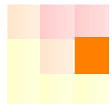
An object or function of the type `int` is declared or defined, but its type is not explicitly stated.

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. This check is identical to MISRAC2004-8.2, MISRAC2012-Rule-8.1

Coding standards

CERT DCL31-C

Declare identifiers before using them

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

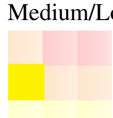
Multiple `#` or `##` operators in a macro definition.

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). This check is identical to MISRAC2004-19.12, MISRAC++2008-16-3-1

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:

```
#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 */
```

ENUM-bounds

Synopsis

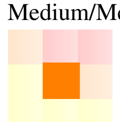
Conversions to `enum` that are out of range of the enumeration.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

There are conversions to `enum` that are out of range of the enumeration. This check is identical to MISRAC++2008-7-2-1

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:

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

void example(void)
{
  ens one = (ens)10;
}
```

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

```
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. This check is identical to MISRAC2012-Rule-13.4_a

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

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

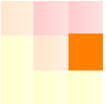
An `else` branch might be connected to an unexpected `if` statement.

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

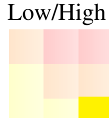
The return type of `main()` is not `int`.

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

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 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

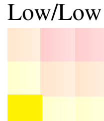
Stray semicolons on the same line as other code

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. This check is identical to MISRAC2004-14.3, MISRAC++2008-6-2-3

Coding standards

CERT EXP15-C

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

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

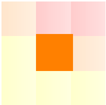
A constant unsigned integer expression overflows.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

A constant unsigned integer expression overflows. This check is identical to MISRAC2004-12.11, MISRAC++2008-5-19-1

Coding standards

CWE 190

Integer Overflow or Wraparound

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

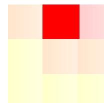
A function pointer that refers to a literal address is dereferenced.

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

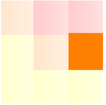
Code examples
The following code example fails the check and will give a warning:

```
/* 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. This check is identical to MISRAC2004-8.1, MISRAC2012-Rule-17.3

Coding standards

CERT DCL31-C

Declare identifiers before using them

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

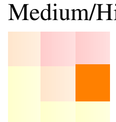
Functions are declared with an empty () parameter list that does not form a valid prototype.

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. This check is identical to MISRAC2004-16.5, MISRAC2012-Rule-8.2_a

Coding standards

CERT DCL20-C

Always specify void even if a function accepts no arguments

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:

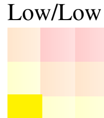
```
void func(void);
void func2(void)
{
    func();
}
```

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

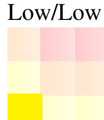
A .c file includes one or more .c files.

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

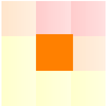
A negative signed integer is implicitly cast to an unsigned integer.

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

CWE 195

Signed to Unsigned Conversion Error

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

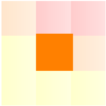
A negative signed integer is implicitly cast to an unsigned integer.

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

CWE 195

Signed to Unsigned Conversion Error

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

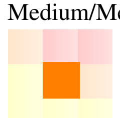
An iterator is used, then compared with `end()`

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:

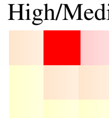
```
#include <vector>

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>

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>

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>

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

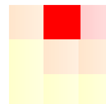
An iterator assigned to point into a container is used or dereferenced even though it might be invalidated.

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>

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:

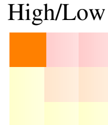
```
#include <vector>

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
No

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 <stdlib.h>
#include <vector>
#include <algorithm>

void example(void) {
  std::vector<int> v, w;
  for (int i=0; i != 10; ++i) {
    v.push_back(rand() % 100);
    w.push_back(rand() % 100);
  }
  std::sort(v.begin(), w.end());  //v and w are different 
containers
}
```

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

```
#include <stdlib.h>
#include <vector>
#include <algorithm>

void example(void) {
  std::vector<int> v;
  for (int i=0; i != 10; ++i) {
    v.push_back(rand() % 100);
  }
  std::sort(v.begin(), v.end());  //OK
}
```

ITR-store (C++ only)

Synopsis

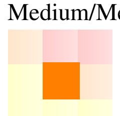
A container's `begin()` or `end()` iterator is stored and subsequently used.

Enabled by default

No

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>

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>

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

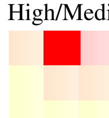
An iterator is dereferenced or incremented before it is assigned to point into a container.

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>

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>

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

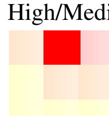
Arguments passed to `bsearch` might cause it to overrun.

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

CWE 676

Use of Potentially Dangerous Function

CWE 122

Heap-based Buffer Overflow

CWE 121

Stack-based Buffer Overflow

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

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>
#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

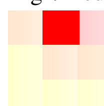
Arguments passed to `bsearch` cause it to overrun.

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

CWE 676

Use of Potentially Dangerous Function

CWE 122

Heap-based Buffer Overflow

CWE 121

Stack-based Buffer Overflow

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

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>
#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-fn-unsafe

Synopsis

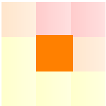
A potentially unsafe library function is used.

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

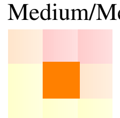
A call to `fread` might cause a buffer overrun.

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

CWE 676

Use of Potentially Dangerous Function

CWE 122

Heap-based Buffer Overflow

CWE 121

Stack-based Buffer Overflow

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 805

Buffer Access with Incorrect Length Value

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

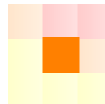
A call to `fread` causes a buffer overrun.

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

CWE 676

Use of Potentially Dangerous Function

CWE 122

Heap-based Buffer Overflow

CWE 121

Stack-based Buffer Overflow

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 805

Buffer Access with Incorrect Length Value

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

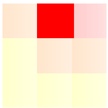
A call to `memchr` might cause a buffer overrun.

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

CWE 676

Use of Potentially Dangerous Function

CWE 122

Heap-based Buffer Overflow

CWE 121

Stack-based Buffer Overflow

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

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 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

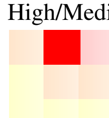
A call to `memchr` causes a buffer overrun.

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

CWE 676

Use of Potentially Dangerous Function

CWE 122

Heap-based Buffer Overflow

CWE 121

Stack-based Buffer Overflow

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

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 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

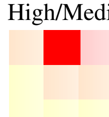
A call to `memcpy` might cause the memory to overrun.

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

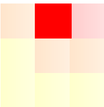
A call to `memcpy` or `memmove` causes the memory to overrun.

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

CWE 676

Use of Potentially Dangerous Function

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, sizeof(int) * (size + 1));
}
```

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

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

void func()
{
  int arr[10];
  int * ptr = (int *)malloc(sizeof(int) * 10);
  memcpy(ptr, arr, sizeof(int) * 10);
}
```

LIB-memset-overrun-pos

Synopsis

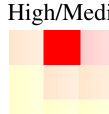
A call to `memset` might cause a buffer overrun.

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

CWE 676

Use of Potentially Dangerous Function

CWE 122

Heap-based Buffer Overflow

CWE 121

Stack-based Buffer Overflow

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

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 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

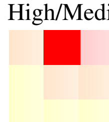
A call to `memset` causes a buffer overrun.

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

CWE 676

Use of Potentially Dangerous Function

CWE 122

Heap-based Buffer Overflow

CWE 121

Stack-based Buffer Overflow

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

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 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

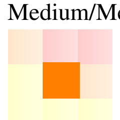
putenv used to set environment variable values.

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

CWE 676

Use of Potentially Dangerous Function

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

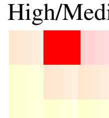
Arguments passed to `qsort` might cause it to overrun.

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

CWE 676

Use of Potentially Dangerous Function

CWE 122

Heap-based Buffer Overflow

CWE 121

Stack-based Buffer Overflow

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

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>
#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

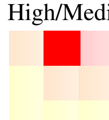
Arguments passed to `qsort` cause it to overrun.

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

CWE 676

Use of Potentially Dangerous Function

CWE 122

Heap-based Buffer Overflow

CWE 121

Stack-based Buffer Overflow

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

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>
#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

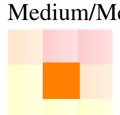
The return value for a library function that might return an error value is not used.

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()`. This check is identical to MISRAC2004-16.10, MISRAC++2008-0-3-2

Coding standards

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:

```
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

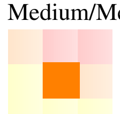
A variable assigned using a library function that can return -1 as an error value is subsequently used where the value must be non-negative.

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
  }
}
```

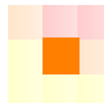
LIB-return-null

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

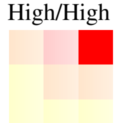
A call to `sprintf` causes a destination buffer overrun.

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

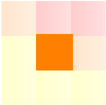
Using `std::sort` might cause buffer overrun.

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

CWE 676

Use of Potentially Dangerous Function

CWE 122

Heap-based Buffer Overflow

CWE 121

Stack-based Buffer Overflow

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

Code examples

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

```
#include <algorithm>

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>

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

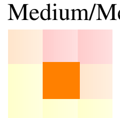
A buffer overrun is caused by use of `std::sort`.

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

CWE 676

Use of Potentially Dangerous Function

CWE 122

Heap-based Buffer Overflow

CWE 121

Stack-based Buffer Overflow

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

Code examples

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

```
#include <algorithm>

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>

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

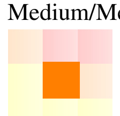
A call to `strcat` might cause destination buffer overrun.

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

CWE 122

Heap-based Buffer Overflow

CWE 676

Use of Potentially Dangerous Function

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

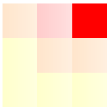
A call to `strcat` causes a destination buffer overrun.

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

CWE 122

Heap-based Buffer Overflow

CWE 676

Use of Potentially Dangerous Function

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:

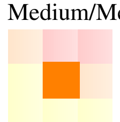
```
#include <string.h>
#include <stdlib.h>

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

CWE 676

Use of Potentially Dangerous Function

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

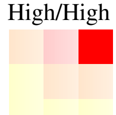
A call to `strcpy` causes a destination buffer overrun.

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

CWE 676

Use of Potentially Dangerous Function

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:

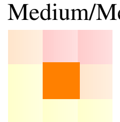
```
#include <string.h>
#include <stdlib.h>

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
CWE 676

Use of Potentially Dangerous Function

CWE 122

Heap-based Buffer Overflow

CWE 121

Stack-based Buffer Overflow

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

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(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

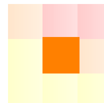
A call to `strncat` causes a destination buffer overrun.

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

CWE 676

Use of Potentially Dangerous Function

CWE 122

Heap-based Buffer Overflow

CWE 121

Stack-based Buffer Overflow

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

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 * 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);
}
```

LIB-strncmp-overrun-pos

Synopsis

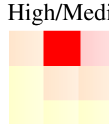
A call to `strncmp` might cause a buffer overrun.

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

CWE 676

Use of Potentially Dangerous Function

CWE 122

Heap-based Buffer Overflow

CWE 121

Stack-based Buffer Overflow

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

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>
#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:

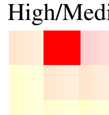
```
#include <stdlib.h>
#include <string.h>

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

CWE 676

Use of Potentially Dangerous Function

CWE 122

Heap-based Buffer Overflow

CWE 121

Stack-based Buffer Overflow

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

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>
#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:

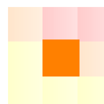
```
#include <stdlib.h>
#include <string.h>

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

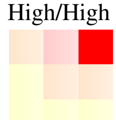
A call to `strncpy` causes a destination buffer overrun.

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:

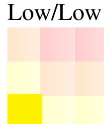
```
#include <string.h>
#include <stdlib.h>

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
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 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

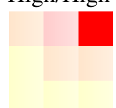
A memory location allocated with `new` is deleted with `delete[]`

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

CWE 763

Release of Invalid Pointer or Reference

CWE 404

Improper Resource Shutdown or Release

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

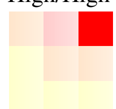
A memory location allocated with `new []` is deleted with `delete` or `free`.

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

CWE 763

Release of Invalid Pointer or Reference

CWE 404

Improper Resource Shutdown or Release

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

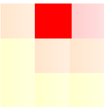
Freeing a memory location more than once.

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

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

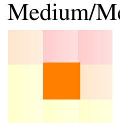
A memory location is freed more than once on some paths but not on others.

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. This check is identical to MISRAC2012-Rule-22.2_b

Coding standards

CERT MEM31-C

Free dynamically allocated memory exactly once

CWE 415

Double Free

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

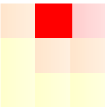
A memory location is freed more than once.

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. This check is identical to MISRAC2012-Rule-22.2_a

Coding standards

CERT MEM31-C

Free dynamically allocated memory exactly once

CWE 415

Double Free

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

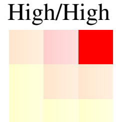
A struct or a class field is possibly freed.

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

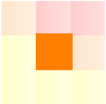
A function pointer is deallocated.

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

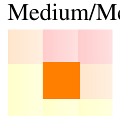
A struct field is deallocated without first having been allocated.

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

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 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

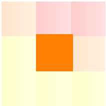
A pointer is freed without having been allocated.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

A pointer is freed without having been allocated.

Coding standards

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 *p;
  // Do stuff
  free(p);
}
```

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

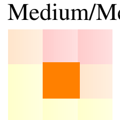
```
#include <stdlib.h>

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>
int * foo() {
  return (int *) 0xF0000000;
}
void example(void) {
  int *p = malloc(sizeof(int));
  *p = 1;
  free(p);
  p = foo();
  free(p);
}
```

MEM-free-op

Synopsis

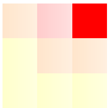
Memory allocated with `malloc` deallocated using `delete`.

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

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 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

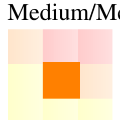
A struct's field is deallocated, but is not dynamically allocated.

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

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 test {
  int a[10];
};

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:

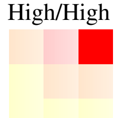
```
#include <stdlib.h>

struct test {
  int *a;
};

void example(void) {
  struct test t;
  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

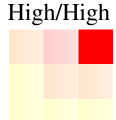
A stack address might be freed.

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. This check is identical to MISRAC2012-Rule-22.2_c

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-leak-alias

Synopsis

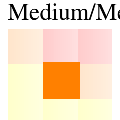
Incorrect deallocation causes memory leak.

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>

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;
}
```

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. This check is identical to MISRAC2012-Rule-22.1_a

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>

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;
}
```

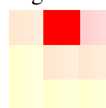
MEM-malloc-arith

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:

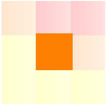
```
#include <stdlib.h>

int example(void) {
  int *p;

  p = (int *)malloc(255);

  return 0;
}
```

MEM-malloc-diff-type

Synopsis
: An allocation call 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

CWE 131

Incorrect Calculation of Buffer Size

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

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);
}
```

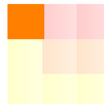
MEM-malloc-sizeof-ptr

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

CWE 131

Incorrect Calculation of Buffer Size

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

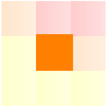
Dangerous arithmetic with `strlen` in argument to `malloc`.

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

CWE 131

Incorrect Calculation of Buffer Size

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

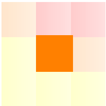
The type of the pointer that stores the result of `realloc` does not match the type of the first argument.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

The type of the pointer 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

CWE 131

Incorrect Calculation of Buffer Size

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

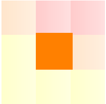
A function deallocates memory, then returns a pointer to that memory.

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

CWE 416

Use After Free

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

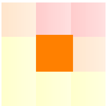
A function that allocates memory's return value is not stored.

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

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:

```
#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-global-field

Synopsis

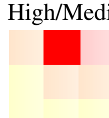
A stack address is stored in the field of a global struct.

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. This check is identical to MISRAC2004-17.6_c, MISRAC++2008-7-5-2_b, MISRAC2012-Rule-18.6_c

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 466

Return of Pointer Value Outside of Expected Range

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

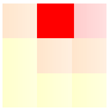
A stack address is stored in a global pointer.

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. This check is identical to MISRAC2004-17.6_b, MISRAC++2008-7-5-2_a, MISRAC2012-Rule-18.6_b

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 466

Return of Pointer Value Outside of Expected Range

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

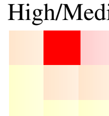
Stack address is stored via reference parameter.

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. This check is identical to MISRAC++2008-7-5-2_d

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 466

Return of Pointer Value Outside of Expected Range

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

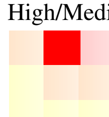
A stack address is stored outside a function via a parameter.

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. This check is identical to MISRAC2004-17.6_d, MISRAC++2008-7-5-2_c, MISRAC2012-Rule-18.6_d

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 466

Return of Pointer Value Outside of Expected Range

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

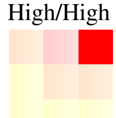
Might return address on the stack.

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

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:

```
int g;
int *example(int *a) {
    int i;
    int *p;
    if (a) {
  p = a;
    } else {
        p = &g;
    }
    return p;
}
```

MEM-stack-ref (C++ only)

Synopsis

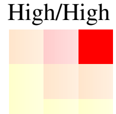
A stack object is returned from a function as a reference.

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. This check is identical to MISRAC++2008-7-5-1_a

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 562

Return of Stack Variable Address

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

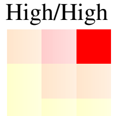
Might return address on the stack.

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 stack frame 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. This check is identical to MISRAC2004-17.6_a, MISRAC++2008-7-5-1_b, MISRAC2012-Rule-18.6_a

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 562

Return of Stack Variable Address

Code examples

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

```
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. This check is identical to SEC-BUFFER-use-after-free-all

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. This check is identical to SEC-BUFFER-use-after-free-some

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

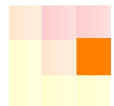
Direct access to a field of a struct, using an offset from the address of the struct.

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. This check is identical to MISRAC2004-17.1_a

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

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

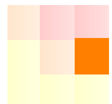
Pointer arithmetic applied to a pointer that references a stack address

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

A pointer is assigned a stack-based address and then used in pointer arithmetic. This check is identical to MISRAC2004-17.1_b, MISRAC++2008-5-0-16_a

Coding standards

CWE 120

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

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

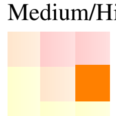
Invalid pointer arithmetic with an automatic variable that is neither an array nor a pointer.

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. This check is identical to MISRAC2004-17.1_c, MISRAC++2008-5-0-16_b

Coding standards

CWE 120

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

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

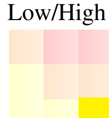
A variable is tested for equality with a string literal.

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

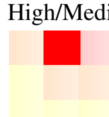
Possible `NULL` pointer dereferenced by a function.

Enabled by default

No

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

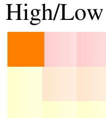
A pointer is assigned a value that might be `NULL`, and then dereferenced.

Enabled by default

No

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

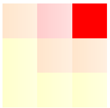
A pointer is assigned the value `NULL`, then dereferenced.

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

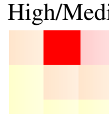
A pointer is dereferenced, then compared with `NULL`.

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;
}
```

PTR-null-cmp-bef-fun

Synopsis

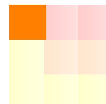
A pointer is compared with `NULL`, then dereferenced by a function.

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 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

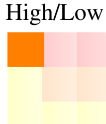
A pointer is compared with `NULL`, then dereferenced.

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

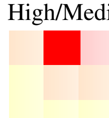
A possible `NULL` pointer is returned from a function, and immediately dereferenced without checking.

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

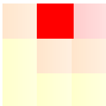
A literal pointer expression (like `NULL`) is dereferenced by a function call.

Enabled by default

No

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);
}
```

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

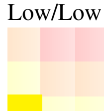
An `&` operator is overloaded.

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

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 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

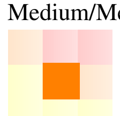
Pointer arithmetic might be performed on a pointer that points to a single object.

Enabled by default

No

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

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

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

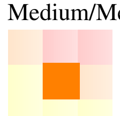
Pointer arithmetic is performed on a pointer that points to a single object.

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

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

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

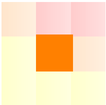
A pointer is dereferenced after being determined not to be `NULL` on some paths, but not checked on others.

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

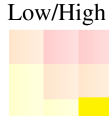
A pointer parameter is not compared to `NULL`

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

No

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. This check is identical to MISRAC2012-Rule-9.1_a

Coding standards

CERT EXP33-C

Do not reference uninitialized memory

CWE 457

Use of Uninitialized Variable

CWE 824

Access of Uninitialized Pointer

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

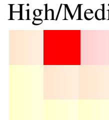
Dereference of an uninitialized or `NULL` pointer.

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. This check is identical to MISRAC2004-9.1_c, MISRAC++2008-8-5-1_c

Coding standards

CERT EXP33-C

Do not reference uninitialized memory

CWE 457

Use of Uninitialized Variable

CWE 824

Access of Uninitialized Pointer

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-alloc-zero-bytes

Synopsis

Checks that an allocation does not allocate zero bytes

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

Checks that an allocation does not allocate zero bytes. Allocation functions checked: malloc/calloc/valloc/alloca/operator new[]/calloc/realloc/memalign/posix_memalign.

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 foo(void) {
  int * x = (int *) malloc(0);
}
```

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

```
#include<stdlib.h>

void foo(int n) {
  int *x = (int *) malloc(n);
}

void bar(int m) {
  int n = 4;
  int *x;
  x = (int *) malloc(m);
  x = (int *) malloc(sizeof(int));
  x = (int *) realloc(0, n);
  posix_memalign(0, 4, n + 4);
  foo(n);
}
```

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. This check is identical to MISRAC++2008-0-1-2_c, MISRAC2012-Rule-2.1_a

Coding standards

CERT MSC07-C

Detect and remove dead 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 :
    ;
  }
}
```

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. This check is identical to MISRAC2004-13.7_a

Coding standards

CWE 571

Expression is Always True

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 on 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. This check is identical to MISRAC2004-13.7_b

Coding standards

CWE 570

Expression is Always False

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

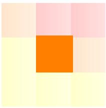
The condition in an if, for, while, do-while, or ternary operator will always be true.

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

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 = 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

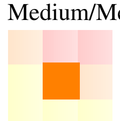
The condition in if, for, while, do-while, or ternary operator will never be true.

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. This check is identical to MISRAC++2008-0-1-2_b, MISRAC2012-Rule-14.3_b

Coding standards

CERT EXP17-C

Do not perform bitwise operations in conditional expressions

CWE 570

Expression is Always False

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. This check is identical to MISRAC2004-14.1, MISRAC++2008-0-1-1, MISRAC++2008-0-1-9, MISRAC2012-Rule-2.1_b

Coding standards

CERT MSC07-C

Detect and remove dead code

CWE 561

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:

```
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

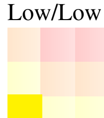
A function is declared that has no return type and creates no side effects.

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. This check is identical to MISRAC++2008-0-1-8

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 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

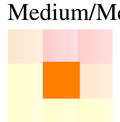
The definition of a local variable hides a global definition.

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

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

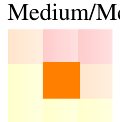
The definition of a local variable hides a previous local definition.

Enabled by default

No

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

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;
}
```

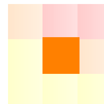
RED-local-hides-member (C++ only)

Synopsis The definition of a local variable hides a member of the class.

Enabled by default No

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

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

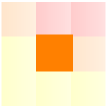
A variable declaration hides a parameter of the function

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

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. This check is identical to MISRAC2004-14.2, MISRAC2012-Rule-2.2_a

Coding standards

CERT MSC12-C

Detect and remove code that has no effect

CWE 482

Comparing instead of Assigning

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>

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;
}
```

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

```
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. Make sure that it is not an error. This check is identical to MISRAC++2008-0-1-11, MISRAC2012-Rule-2.7

Coding standards

CWE 563

Unused Variable

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. This check is identical to MISRAC++2008-0-1-7, MISRAC2012-Rule-17.7

Coding standards

CWE 252

Unchecked Return Value

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

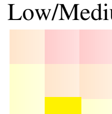
A variable is assigned a value that is never used.

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. This check is identical to MISRAC++2008-0-1-6, MISRAC2012-Rule-2.2_c

Coding standards

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;
}
```

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

```
int example(void) {
  int x;
  x = 20;
  return x;
}
```

RED-unused-var-all

Synopsis

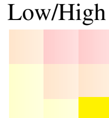
A variable is neither read nor written for any execution path.

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. This check is identical to MISRAC++2008-0-1-3

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;  //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

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 *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

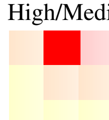
A file resource is closed multiple times

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

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 <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

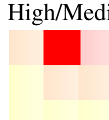
A file pointer is never closed.

Enabled by default

Yes

Severity/Certainty

High/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. This check is identical to MISRAC2012-Rule-22.1_b, SEC-FILEOP-open-no-close

Coding standards

CERT FIO42-C

Ensure files are properly closed when they are no longer needed

CWE 404

Improper Resource Shutdown or Release

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);
}
```

RESOURCE-file-pos-neg

Synopsis

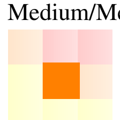
A file handler might be negative

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 <LowLevelIOInterface.h>

void example(void) {
  int a = __open("test.txt", _LLIO_WRONLY);
  write(a, "Hello", 5);
}
```

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

```
#include <LowLevelIOInterface.h>

void example(void) {
  int a = __open("test.txt", _LLIO_WRONLY);
  if (a > 0) {
    write(a, "Hello", 5);
  }
}
```

RESOURCE-file-use-after-close

Synopsis

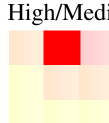
A file resource is used after it has been closed.

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

CERT FIO46-C

Do not access a closed file

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:

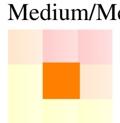
```
#include <stdio.h>

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. This check is identical to MISRAC2012-Rule-22.5_b

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>
#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:

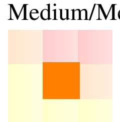
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

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. This check is identical to MISRAC2012-Rule-22.4

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) {
  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

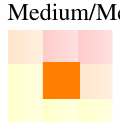
`sizeof` expressions containing side effects

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. This check is identical to MISRAC2004-12.3, MISRAC++2008-5-3-4

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

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-order

Synopsis

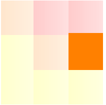
Expressions that depend on order of evaluation were found.

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. This check is identical to MISRAC2004-12.2_a, MISRAC++2008-5-0-1_a, MISRAC2012-Rule-13.2_a

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

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

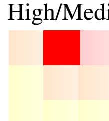
Reads from local buffers are not preceded by writes.

Enabled by default

No

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. This check is identical to MISRAC2004-1.2_a, MISRAC2012-Rule-9.1_b

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:

```
void example() {
  int a[20];
  int b = a[1];
}
```

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

```
extern void f(int*);
void example() {
  int a[20];
  f(a);
  int b = a[1];
}
```

SPC-uninit-struct-field-heap

Synopsis

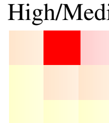
A field of a dynamically allocated struct is read before it is initialized.

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

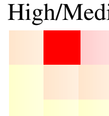
The following code example fails the check and will give a warning:

```
#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	No
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. This check is identical to MISRAC2012-Rule-9.1_d

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:

```
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

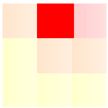
A struct has one or more fields read before they are initialized.

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. This check is identical to MISRAC2004-1.2_b, MISRAC2012-Rule-9.1_c

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:

```
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

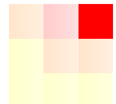
A variable is read before it is assigned a value.

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. This check is identical to MISRAC2004-9.1_a, MISRAC++2008-8-5-1_a, MISRAC2012-Rule-9.1_e

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:

```
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

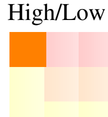
A variable is read before it is assigned a value.

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 read before it is assigned a value. This might cause unpredictable application behavior. This check is identical to MISRAC2004-9.1_b, MISRAC++2008-8-5-1_b, MISRAC2012-Rule-9.1_f

Coding standards

CWE 457

Use of Uninitialized Variable

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

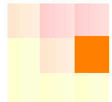
There are multiple read accesses with volatile-qualified type within one and the same sequence point.

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. This check is identical to MISRAC2004-12.2_b, MISRAC++2008-5-0-1_b, MISRAC2012-Rule-13.2_b

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

Code examples

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

```
void example(void) {
  int x;
  volatile int v;
  x = v + v;
}
```

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

```
int main(void) {
  volatile int i = 0;
  int x = i;
  i++;
  x = x * i;  //OK - statement is broken up
  return 0;
}
```

SPC-volatile-writes

Synopsis

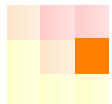
There are multiple write accesses with volatile-qualified type within one and the same sequence point.

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. This check is identical to MISRAC2004-12.2_c, MISRAC++2008-5-0-1_c, MISRAC2012-Rule-13.2_c

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

Code examples

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

```
void example(void) {
  int x;
  volatile int v, w;
  v = w = x;
}
```

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

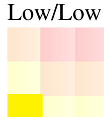
```
#include <stdbool.h>

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;
}
```

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. This check is identical to MISRAC2004-6.5, MISRAC++2008-9-6-4, MISRAC2012-Rule-6.2

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:

```
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

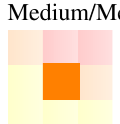
There are non-empty switch cases not terminated by break and without 'fallthrough' comment.

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) {

  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;
  }

}
```

THROW-empty (C++ only)

Synopsis

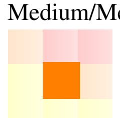
Unsafe rethrow of exception.

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. This check is identical to MISRAC++2008-15-1-3

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 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

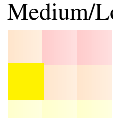
No default exception handler for `try`.

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. This check is identical to MISRAC++2008-15-3-2

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 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

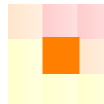
Throw of NULL integer constant

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. This check is identical to MISRAC++2008-15-1-2

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:

```
typedef int int32_t;
typedef signed char char_t;
#defineNULL 0

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 int int32_t;
typedefsigned char char_t;
#defineNULL 0

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

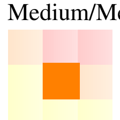
Throw of exceptions by pointer

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. This check is identical to MISRAC++2008-15-0-2

Coding standards

CERT ERR09-CPP

Throw anonymous temporaries and catch by reference

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

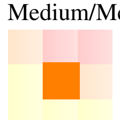
Exceptions thrown without a handler in some call paths that lead to that point.

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. This check is identical to MISRAC++2008-15-3-1

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 C {
public:
  C ( ) { throw ( 0 ); } // Non-compliant – thrown before main
starts
  ~C ( ) { throw ( 0 ); } // Non-compliant – thrown after main
exits
};

// 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
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;
  }
}
```

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

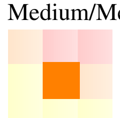
There are calls to functions explicitly declared to throw an exception type that is not handled (or declared as thrown) by the caller.

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. This check is identical to MISRAC++2008-15-3-4

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 E1{};

void foo(int i) throw (E1) {
  if (i<0)
    throw E1();
}

int bar() {
  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

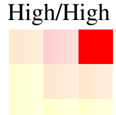
Assignments from one field of a union to another.

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. This check is identical to MISRAC2004-18.2, MISRAC++2008-0-2-1, MISRAC2012-Rule-19.1

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)
{
  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;
}
```

UNION-type-punning

Synopsis

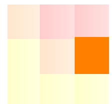
Writing to a field of a union after reading from a different field, effectively re-interpreting the bit pattern with a different type.

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. This check is identical to MISRAC2004-12.12_a

Coding standards

CERT EXP39-C

> Do not access a variable through a pointer of an incompatible type

CWE 188

> Reliance on Data/Memory Layout

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:

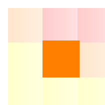
```
union name {
   int int_field;
   float float_field;
};

void example(void) {
   union name u;
   u.int_field = 10;
   float f = u.int_field;
}
```

CERT-EXP19-C

Synopsis
No braces for the body of an if, for, or while statement

Enabled by default
Yes

Severity/Certainty
Medium/Medium



Full description
The body of an if, for, or while statement is missing opening and closing braces. Opening and closing braces for if, for, and while statements should always be used even if the statement's body contains only a single statement

Coding standards
CERT EXP19-C

Use braces for the body of an if, for, or while statement

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

```
void example(void) {
  int login;

  if (invalid_login())
    login = 0;
  else
    login = 1;
}
```

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

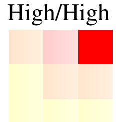
```
#define ADMINISTRATOR 0
#define GUEST 1

void example(void) {
  int privileges;

  if (invalid_login()) {
    if (allow_guests()) {
      privileges = GUEST;
    }
  } else {
    privileges = ADMINISTRATOR;
  }
}
```

CERT-FIO37-C

Synopsis
A string returned by fgets() and fgetsws() might contain NULL characters.

Enabled by default
Yes

Severity/Certainty
High/High



Full description
A string returned by fgets() and fgetsws() might contain NULL characters. If the length of this string is then used to access the buffer, it might result in an unexpect integer wrap around leading to an out-of-bounds memory write.

Coding standards
CERT FIO37-C

Do not assume that fgets() returns a nonempty string when successful

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 241

Improper Handling of Unexpected Data Type

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

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

enum { BUFFER_SIZE = 1024 };

void func(void) {
  char buf[BUFFER_SIZE];

  if (fgets(buf, sizeof(buf), stdin) == NULL) {
    /* Handle error */
  }
  buf[strlen(buf) - 1] = '\0';
}
```

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

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

enum { BUFFER_SIZE = 1024 };

void func(void) {
  char buf[BUFFER_SIZE];
  char *p;

  if (fgets(buf, sizeof(buf), stdin)) {
    p = strchr(buf, '\n');
    if (p) {
      *p = '\0';
    }
  } else {
    /* Handle error */
  }
}
```

CERT-FIO38-C

Synopsis	A FILE object is copied.
Enabled by default	Yes

Severity/Certainty

Low/Medium

Full description

A FILE object is copied. In some C implementations, the address of a FILE object might be used to identify a stream. Using a copy of FILE object might result in unexpected behavior or a crash.

Coding standards

CERT FIO38-C

Do not use a copy of a FILE object for input and output

Code examples

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

```
#include <stdio.h>

void example(FILE file) {
  FILE my_file = file;
}
```

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

```
#include <stdio.h>

void example(FILE * file_ptr) {
  FILE * my_file_ptr = file_ptr;
}
```

CERT-SIG31-C

Synopsis

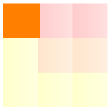
Shared objects in a signal handler are accessed or modified.

Enabled by default

Yes

Severity/Certainty

High/Low



Full description

Accessing or modifying shared objects (not of the type `volatile sig_atomic_t`) in a signal handler might result in race conditions that can leave data in an inconsistent state.

Coding standards

CERT SIG31-C

Do not access or modify shared objects in signal handlers

CWE 662

Improper Synchronization

Code examples

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

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>

enum { MAX_MSG_SIZE = 24 };
char *err_msg;

void handler(int signum) {
  strcpy(err_msg, "SIGINT encountered.");
}

int main(void) {
  signal(SIGINT, handler);

  err_msg = (char *)malloc(MAX_MSG_SIZE);
  if (err_msg == NULL) {
    /* Handle error */
  }
  strcpy(err_msg, "No errors yet.");
  /* Main code loop */
  return 0;
}
```

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

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>

enum { MAX_MSG_SIZE = 24 };
volatile sig_atomic_t e_flag = 0;

void handler(int signum) {
  e_flag = 1;
}

int main(void) {
  char *err_msg = (char *)malloc(MAX_MSG_SIZE);
  if (err_msg == NULL) {
    /* Handle error */
  }

  signal(SIGINT, handler);
  strcpy(err_msg, "No errors yet.");
  /* Main code loop */
  if (e_flag) {
    strcpy(err_msg, "SIGINT received.");
  }
  return 0;
}
```

SEC-BUFFER-memory-leak-alias

Synopsis

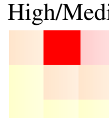
A memory leak is caused by incorrect deallocation.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

Memory has been allocated, then the pointer value is lost because it is reassigned or its scope ends, without a guarantee that the value will be propagated or the memory be freed. The value must be freed, returned, or passed to another function as an argument, before it is lost, on all possible execution paths. Before a pointer is reassigned or its scope ends, the memory it points to must be freed, or a new pointer must be assigned to the memory.

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>

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;
}
```

SEC-BUFFER-memory-leak

Synopsis

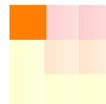
A memory leak is caused by incorrect deallocation.

Enabled by default

No

Severity/Certainty

High/Low



Full description

Memory has been allocated, then the pointer value is lost because it is reassigned or its scope ends, without a guarantee that the value will be propagated or the memory be freed. The value must be freed, returned, or passed to another function as an argument, before it is lost, on all possible execution paths. Before a pointer is reassigned or its scope ends, the memory it points to must be freed, or a new pointer must be assigned to the memory. This check is identical to MISRAC2012-Rule-22.1_a, 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

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>

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;
}
```

SEC-BUFFER-memset-overrun-pos

Synopsis

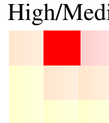
A call to memset might overrun the buffer.

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 exceeding the size of the allocated buffer, it will overrun. This might cause a runtime error. Make sure that the size of the buffer passed to memset does not exceed the destination buffer's size. You might need to add a condition before the call to memset.

Coding standards

CWE 121

Stack-based Buffer Overflow

CWE 122

Heap-based Buffer Overflow

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

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);
}
```

SEC-BUFFER-memset-overrun

Synopsis

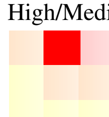
A call to memset overruns the buffer.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

A buffer overrun is caused by a call to memset. If memset is called with a size exceeding the size of the allocated buffer, it will overrun. This might cause a runtime error. Make sure that the size of the buffer passed to memset does not exceed the destination buffer's size. You might need to add a condition before the call to memset.

Coding standards

CWE 121

Stack-based Buffer Overflow

CWE 122

Heap-based Buffer Overflow

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

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);
}
```

SEC-BUFFER-qsort-overrun-pos

Synopsis

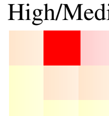
Arguments passed to qsort might cause it to overrun.

Enabled by default

No

Severity/Certainty

High/Medium



Full description

A call to qsort might cause a buffer overrun. An overrun might be caused by passing a buffer length that exceeds that of the buffer passed to either function, as their first argument. Make sure that a correct buffer length and size is passed to qsort. The call to qsort might need to be preceded with a comparison of the buffer length and element size.

Coding standards

CWE 122

Heap-based Buffer Overflow

CWE 121

Stack-based Buffer Overflow

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

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);
}
```

SEC-BUFFER-qsort-overrun

Synopsis

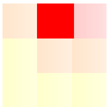
Arguments passed to qsort cause it to overrun.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

A buffer overrun is caused by a call to qsort. An overrun is caused by passing a buffer length that exceeds that of the buffer passed to either function, as their first argument. Make sure that a correct buffer length and size is passed to qsort. The call to qsort might need to be preceded with a comparison of the buffer length and element size.

Coding standards

CWE 122

Heap-based Buffer Overflow

CWE 121

Stack-based Buffer Overflow

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

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);
}
```

SEC-BUFFER-sprintf-overrun

Synopsis

A call to the sprintf function will overrun the target buffer.

Enabled by default

Yes

Severity/Certainty

High/High

Full description

A call to the sprintf function will overrun the target buffer. Consider using a function that allows you to set the buffer length, such as snprintf. Alternatively, you might be able to compare the lenghts of the source and destination buffer before calling sprintf.

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:

```
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:

```
char buf[14];

void example(void) {
  sprintf(buf, "Hello World!\n");
}
```

SEC-BUFFER-std-sort-overrun-pos (C++ only)

Synopsis

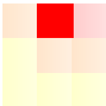
Use of std::sort might cause a buffer overrun.

Enabled by default

No

Severity/Certainty

High/Medium



Full description

std::sort can take a pointer to an array and a pointer to the end of the array as arguments. However, if the pointers do not point into the same array, or if the end pointer is so far away that some elements outside the array are included, a buffer overrun might occur. Ensure that both pointers passed to std::sort point within the same buffer.

Coding standards

CWE 122

Heap-based Buffer Overflow

CWE 121

Stack-based Buffer Overflow

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

Code examples

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

```
#include <algorithm>

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>

void example(void) {
  int a[10] = {0,1,2,3,4,5,6,7,8,9};
  std::sort(a, a+5);
}
```

SEC-BUFFER-std-sort-overrun (C++ only)

Synopsis

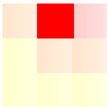
A buffer overrun is caused by use of std::sort.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

std::sort can take a pointer to an array and a pointer to the end of the array as arguments. However, if the pointers do not point into the same array, or if the end pointer is so far away that some elements outside the array are included, a buffer overrun might occur. Ensure that both pointers passed to std::sort point within the same buffer.

Coding standards

CWE 122

Heap-based Buffer Overflow

CWE 121

Stack-based Buffer Overflow

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

Code examples

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

```
#include <algorithm>

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>

void example(void) {
  int a[10] = {0,1,2,3,4,5,6,7,8,9};
  std::sort(a, a+5);
}
```

SEC-BUFFER-strcat-overrun-pos

Synopsis

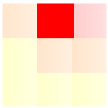
A call to the strcat function might overrun the target buffer.

Enabled by default

No

Severity/Certainty

High/Medium



Full description

A call to the strcat function might overrun the target buffer. strcat appends to the target the contents of the source string up until a null character. If the length of the source buffer is longer than the amount allocated in the destination buffer, a buffer overflow occurs. Alternatively, if the source string is not null terminated, strcat could read past the intended bytes and overflow the destination buffer. If possible, use strncat instead of strcat to set an upper bound on the number of bytes to append. You should also try to check the length of source and destination buffer before calling strcat.

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);
}
```

SEC-BUFFER-strcat-overrun

Synopsis

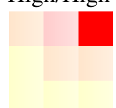
A call to the strcat function will overrun the target buffer.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

A call to the strcat function will overrun the target buffer. strcat appends to the target the contents of the source string up until a null character. If the length of the source buffer is longer than the amount allocated in the destination buffer, a buffer overflow occurs. Alternatively, if the source string is not null terminated, strcat could read past the intended bytes and overflow the destination buffer. If possible, use strncat instead of strcat to set an upper bound on the number of bytes to append. You should also try to check the length of source and destination buffer before calling strcat.

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:

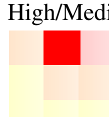
```
#include <string.h>
#include <stdlib.h>

void example(void)
{
  char *str1 = "Hello World!\n";
  char *str2 = (char *)malloc(14);
  strcpy(str2, "");
  strcat(str2, str1);
}
```

SEC-BUFFER-strcpy-overrun-pos

Synopsis
A call to the strcpy function might overrun the target buffer.

Enabled by default
No

Severity/Certainty
High/Medium



Full description
A call to the strcpy function might overrun the target buffer. strcpy will copy the contents of the source string, up until the null character. If the length of the source string exceeds the intended destination, a buffer overflow occurs which might overwrite memory you did not intend to. Alternatively, if the null character is not present, strcpy might continue past the intended end of the string and read unintended memory into the buffer. If possible, use strncpy to set an upper limit on the number of bytes copied into the destination buffer. The number of bytes should be the length of the destination buffer.

Alternatively, you might be able to check the length of both the source and destination buffers before calling strcpy.

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);
}
```

SEC-BUFFER-strcpy-overrun

Synopsis

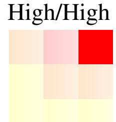
A call to the strcpy function will overrun the target buffer.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

A call to the strcpy function will overrun the target buffer. strcpy will copy the contents of the source string, up until the null character. If the length of the source string exceeds the intended destination, a buffer overflow occurs which might overwrite memory you did not intend to. Alternatively, if the null character is not present, strcpy might continue past the intended end of the string and read unintended memory into the buffer. If possible, use strncpy to set an upper limit on the number of bytes copied into the destination buffer. The number of bytes should be the length of the destination buffer. Alternatively, you might be able to check the length of both the source and destination buffers before calling strcpy.

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);
}
```

SEC-BUFFER-strncat-overrun-pos

Synopsis

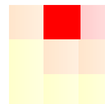
A buffer overrun might be caused by a call to strncat.

Enabled by default

No

Severity/Certainty

High/Medium



Full description

Calling strncat with a destination buffer that is too small causes 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 be appended, as determined by the position of the null terminator in the source buffer or the size passed as the third argument to strncat, then an overflow might occur resulting in undefined behavior and potential runtime errors. Make sure that the length passed to strncat is correct. You might need to perform an comparison before calling strncat.

Coding standards

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 121

Stack-based Buffer Overflow

CWE 122

Heap-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(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);
}
```

SEC-BUFFER-strncat-overrun

Synopsis

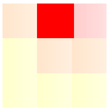
A call to strncat causes a buffer overrun.

Enabled by default

Yes

Severity/Certainty

High/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 be appended, as determined by the position of the null terminator in the source buffer or the size passed as the third argument to strncat, then an overflow might occur resulting in undefined behavior and potential runtime errors. Make sure that the length passed to strncat is correct. You might need to perform an comparison before calling strncat.

Coding standards

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 121

Stack-based Buffer Overflow

CWE 122

Heap-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 * 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);
}
```

SEC-BUFFER-strncmp-overrun-pos

Synopsis

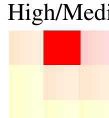
A call to strncmp might cause a buffer overrun.

Enabled by default

No

Severity/Certainty

High/Medium



Full description

Passing an incorrect string length to strncmp might cause a buffer overrun. Strncmp limits the number of characters it compares to the number of characters passed as its third argument, to prevent buffer overruns with non-null terminated strings. However, if the number of characters passed exceeds the length of the two strings, and none of these strings is null terminated, then it will overrun. Make sure the length passed to strncmp is correct. You might need to perform an comparison before calling strncmp.

Coding standards

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

CWE 121

Stack-based Buffer Overflow

CWE 122

Heap-based Buffer Overflow

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);
}
```

SEC-BUFFER-strncmp-overrun

Synopsis

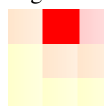
A buffer overrun is caused by a call to strncmp.

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 of characters passed as its third argument, to prevent buffer overruns with non-null terminated strings. However, if the number of characters passed exceeds the length of the two strings, and none of these strings is null terminated, then it will overrun. Make sure the length passed to strncmp is correct. You might need to perform an comparison before calling strncmp.

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);
}
```

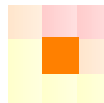
SEC-BUFFER-strncpy-overrun-pos

Synopsis

The target buffer might be overrun by a call to the strncpy function.

Enabled by default No

Severity/Certainty Medium/Medium



Full description The target buffer might be overrun by a call to the strncpy function. If the supplied buffer length exceeds the actual length of the destination buffer, strncpy might write past the bounds of the destination buffer. Make sure the length passed to strncpy is correct. You might need to perform a comparison before calling strncpy.

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);
}
```

SEC-BUFFER-strncpy-overrun

Synopsis

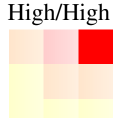
A call to the strncpy function will overrun the target buffer.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

A call to the strncpy function will overrun the target buffer. If the supplied buffer length exceeds the actual length of the destination buffer, strncpy might write past the bounds of the destination buffer. Make sure the length passed to strncpy is correct. You might need to perform a comparison before calling strncpy.

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);
}
```

SEC-BUFFER-tainted-alloc-size

Synopsis

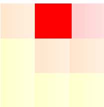
A user is able to control the amount of memory used in an allocation.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

The size of an allocation is derived from user input. User input should be bounds-checked before it is used as an argument to a memory allocation function. If the size being passed to an allocation function is not checked properly, an attacker might cause an application crash via an out-of-memory condition, or cause the application to consume large amounts of memory on a system. Any size derived from user input that is passed to an allocation function should be checked to make sure it is not too large.

Coding standards

CERT INT04-C

Enforce limits on integer values originating from untrusted sources

CWE 789

Uncontrolled Memory Allocation

CWE 770

Allocation of Resources Without Limits or Throttling

CWE 20

Improper Input Validation

Code examples

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

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

int main(char* argc, char** argv) {
  int num;
  char buffer[50];
  char *other_string = "Hello World!";
  gets(buffer);
  sscanf(buffer, "%d", &num);
  if (num > 100) return -1;
  char *string = (char *)malloc(num);
  strcpy(string, other_string);
}
```

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

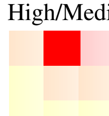
```
#include <stdio.h>
#include <string.h>

int main(char* argc, char** argv) {
  int num;
  char buffer[50];
  char *other_string = "Hello World!";
  gets(buffer);
  sscanf(buffer, "%d", &num);
  if (num < strlen(other_string) || num > 100) return -1;
  char *string = (char *)malloc(num);
  strcpy(string, other_string);
}
```

SEC-BUFFER-tainted-copy-length

Synopsis
A tainted value is used as the size of the memory copied from one buffer to another.

Enabled by default
Yes

Severity/Certainty
High/Medium



Full description
A value derived from user input is used as the size of the memory when contents is copied from one buffer to another. An attacker could supply a value that causes a buffer overrun, which might expose sensitive data stored in memory or cause an application

crash. Buffer sizes taken from user input should be properly bounds-tested before they are used.

Coding standards

CERT INT04-C

Enforce limits on integer values originating from untrusted sources

CWE 126

Buffer Over-read

CWE 120

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

Code examples

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

```
#include <stdio.h>

int main(int argc, char **argv) {
  char dest[50], src[50];
  int size = getchar();
  int size2 = 10;
  int size3 = 20;
  int size4 = 30;
  int i;
  for (i = 0; i < 4; i++) {
    memcpy(dest, src, size4);
    size4 = size3;
    size3 = size2;
    size2 = size;
  }
}
```

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

```
#include <stdio.h>

int main(int argc, char **argv) {
  char dest[50], src[50];
  int size = getchar();
  int size2 = 10;
  int size3 = 20;
  int size4 = 30;
  int i;
  for (i = 0; i < 4; i++) {
    if (size4 >= 0 && size4 <= 50)
      memcpy(dest, src, size4);
    size4 = size3;
    size3 = size2;
    size2 = size;
  }
}
```

SEC-BUFFER-tainted-copy

Synopsis

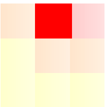
User input is copied into a buffer.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

An unbounded copying function is used to copy the contents of a buffer that contains user input, into another buffer. If the length of the user input is not checked before it is copied, an attacker could input data longer than the intended destination. This data could overwrite other values stored in memory, causing unexpected (and potentially dangerous) behavior and could lead to arbitrary code execution. The length of user input should be checked before it is used in an unbounded copy function, or such functions should be avoided altogether.

Coding standards

CWE 120

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

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

Code examples

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

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

int main(int argc, char **argv) {
  char passwd[10];
  char *input = getenv("PASSWORD");
  int accept;

  strcpy(passwd, input);

  if (accept)
    printf("Login Successful\n");
  else
    printf("Unsuccessful Login\n");
}
```

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

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

int main(int argc, char **argv) {
  char passwd[10];
  int accept;

  if (strlen(argv[1]) < 10)
    strcpy(passwd, argv[1]);

  if (accept)
    printf("Login Successful\n");
  else
    printf("Unsuccessful Login\n");
}
```

SEC-BUFFER-tainted-index

Synopsis

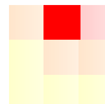
An array is accessed with an index derived from user input.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

An array is accessed with an index that is unchecked and derived from user input. An attacker could create input that might cause a buffer overrun. Such an attack might cause an application crash, corruption of data, or exposure of sensitive information in memory. All input from users should be bounds-checked before it is used to access an array.

Coding standards

CERT INT04-C

Enforce limits on integer values originating from untrusted sources

CWE 129

Improper Validation of Array Index

CWE 126

Buffer Over-read

Code examples

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

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

int *main(int argc, char *argv[]) {
  int *options[10];
  char buffer[1024];
  int index, success, socket;
  success = recv(socket, buffer, sizeof(buffer) - 1, 0);
  if (!success) return 0;
  sscanf(buffer, "%d", &index);
  return options[index]; /* Index could be any integer */
}
```

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

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

int *main(int argc, char *argv[]) {
  int *options[10];
  char buffer[1024];
  int index, success, socket;
  success = recv(socket, buffer, sizeof(buffer) - 1, 0);
  if (!success) return 0;
  sscanf(buffer, "%d", &index);
  if (index >= 0 && index < 10)
    return options[index]; /* Index is between 0 and 9 */
}
```

SEC-BUFFER-tainted-offset

Synopsis

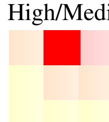
A user-controlled variable is used as an offset to a pointer without proper bounds checking.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

In an arithmetic operation involving a pointer, a variable is used that is under user control. Without checking the bounds of this variable, an attacker could send a value to the application that might cause a buffer overrun, corruption of data, or exposure of sensitive information stored in memory. The bounds of all tainted variables must be properly checked before used in pointer arithmetic.

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 *p) {
  int a = atoi(getenv("TEST"));
  p + a;
}
```

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 *p) {
  int a = atoi(getenv("TEST"));
  if (a > 0 && a < 10)
    p + a;
}
```

SEC-BUFFER-use-after-free-all

Synopsis

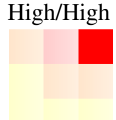
A pointer is used after it has been freed, on all execution paths.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

Memory is being accessed after it has been deallocated. The application might seem to work, but the operation is illegal. This will probably cause an application crash, or the program might continue operating with erroneous or corrupt data. A pointer should be assigned to a different and valid memory location (either by aliasing another pointer, or by performing another allocation) before being used. This check is identical to MEM-use-free-all, MISRAC2012-Dir-4.13_d, MISRAC2012-Rule-1.3_o

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
}
```

SEC-BUFFER-use-after-free-some

Synopsis

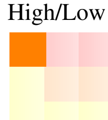
A pointer is used after it has been freed, on some execution paths.

Enabled by default

Yes

Severity/Certainty

High/Low



Full description

A pointer is used after it has been freed, on some execution paths. This might cause data corruption or an application crash. A pointer should be assigned to a different and valid memory location (either by aliasing another pointer, or by performing another allocation) before being used. This check is identical to MEM-use-free-some, MISRAC2012-Dir-4.13_e, MISRAC2012-Rule-1.3_p

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++;
}
```

SEC-DIV-0-compare-after

Synopsis

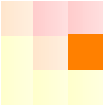
After a successful comparison with 0, a variable is used as a divisor.

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

A variable is compared to 0, then used as a divisor before being written to. The comparison implies that the variable's value is 0 for all following statements. Using it as a divisor afterwards causes a 'divide by zero' runtime error. This check is identical to MISRAC2004-1.2_e, MISRAC2012-Rule-1.3_c, ATH-div-0-cmp-aft

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 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;
}
```

SEC-DIV-0-compare-before

Synopsis

A variable is first used as a divisor, then 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. The comparison 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 (which would cause a 'divide by zero' runtime error), the execution can never reach the comparison when the value is 0, making it meaningless. This check is identical to MISRAC2004-1.2_f, MISRAC2012-Rule-1.3_d, ATH-div-0-cmp-bef

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 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;
}
```

SEC-DIV-0-tainted

Synopsis

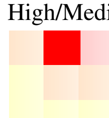
User input is used as a divisor without validation.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

User input is used as a divisor without first checking that it is within a range. This means that an attacker can send a value that might trigger a division by zero error, for example as part of a denial of service attack.

Coding standards

CWE 369

Divide By Zero

Code examples

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

```
int main(int argc, char **argv) {
  return 10 / argc;
}
```

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

```
int main(int argc, char **argv) {
  if (argc > 0 && argc < 10)
    return 10 / argc;
  else
    return 1;
}
```

SEC-FILEOP-open-no-close

Synopsis

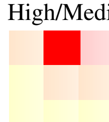
All file pointers obtained dynamically by means of Standard Library functions must be explicitly released.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

If file pointers are not explicitly released, a failure might occur caused by exhaustion of the resources. Release file pointers as soon as possible to reduce the risk of exhaustion. Make sure that files are closed on all execution paths in a function. This check is identical to MISRAC2012-Rule-22.1_b, RESOURCE-file-no-close-all, MISRAC2012-Dir-4.13_c

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);
}
```

SEC-FILEOP-path-traversal

Synopsis

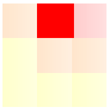
User input is used as a file path, or used to derive a file path.

Enabled by default

No

Severity/Certainty

High/Medium



Full description

User input is used either directly or in part to derive a file path. Unless this information is checked, an attacker could send a value that causes a file open to traverse out of the intended directory. As a result, files you wish to keep secure could be opened, modified, or deleted. An attacker could also create files in undesired locations. Values that come from user input should be checked, by string comparison or similar, before being used as a path to a file.

Coding standards

CWE 22

Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

CWE 23

Relative Path Traversal

CWE 36

Absolute Path Traversal

Code examples

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

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

int main(int argc, char *argv[]) {
  char path[100] = "/tmp/sandbox/";
  strncat(path, argv[1], 50);
  FILE *file = fopen(path, "r");
  if (!file) return -1;
  char c;
  while((c = fgetc(file)) != EOF) {
    printf("%c", c);
  }
  fclose (file);
  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(int argc, char *argv[]) {
  char path[100] = "/tmp/sandbox/plain.txt";
  FILE *file = fopen(path, "r");
  if (!file) return -1;
  char c;
  while((c = fgetc(file)) != EOF) {
    printf("%c", c);
  }
  fclose (file);
  return 0;
}
```

SEC-FILEOP-use-after-close

Synopsis

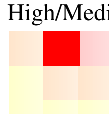
A file resource is used after it has been closed.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

A file resource is referred to after it has been closed. Once a file has been closed, the reference to that file is invalidated. Any use of this reference is undefined and might result in an application crash. A file pointer should not be used after the file it points to is closed. To use the file pointer again, you must open a new file with that pointer.

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);
}
```

SEC-INJECTION-sql

Synopsis

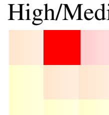
User input is improperly used in an SQL statement

Enabled by default

No

Severity/Certainty

High/Medium



Full description

An SQL statement is constructed either completely or partially from user input. When user input is used in an SQL statement, that statement should be parameterized and the user input be passed as a parameter. By using user input directly in an SQL statement (through string concatenation or similar) you leave the statement open to attack. An attacker could provide input to execute arbitrary commands on your database. These commands could expose information in the database, overwrite existing data, or delete elements from the database. This check supports the following C/C++ libraries for SQL: * MySQL C API * MySQL Connector/C++ * libpq (PostgreSQL) * libpq++ (PostgreSQL) * libpqxx (PostgreSQL) * sqlite3 * Microsoft ODBC * OLE DB User input should be sanitized using an SQL escaping function.

Coding standards

CWE 89

Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Code examples

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

```
#include <string.h>

void example(void * conn) {
  char *name;
  char *sql;
  name = gets(name);
  strcpy(sql, "SELECT age FROM people WHERE name = \"");
  strcat(sql, name);
  strcat(sql, "\"");
  sqlite3_exec(conn, sql);
}
```

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

```
#include <string.h>

void example(void * conn, void * stmt) {
  char *name;
  name = gets(name);
  sqlite3_bind_text(stmt, "A", name);
  sqlite3_exec(conn, "SELECT age FROM people WHERE name = $A");
}
```

SEC-INJECTION-xpath

Synopsis

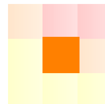
User input is improperly used as an XPath expression

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

An XPath expression is constructed either entirely or partially from user input. User input used in XPath expressions must be sanitized before used. An attacker could provide input to expose the structure of the XML document, or acccess fields they normally do not have access to. Unlike databases there is no level access control, so an attacker can access the entire document. This check supports the following C/C++ libraries for XPath: * libxml2 * Xerces * MSXML * libxml++ * TinyXPath * libroxml * pugixml User input should be checked through string comparison or similar before being used in an XPath query.

Coding standards

CWE 91

XML Injection (aka Blind XPath Injection)

Code examples

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

```
#include <string.h>

void example(void * xml) {
  char *name;
  char *xpath;
  name = gets(name);
  strcpy(xpath, "children::*[@name = '");
  strcat(xpath, name);
  strcat(xpath, "'");
  xmlXPathEval(xml, xpath);
}
```

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

```
#include <string.h>

void example(void * xml, char *name) {
  char *xpath;
  strcpy(xpath, "children::*[@name = '");
  strcat(xpath, name);
  strcat(xpath, "'");
  xmlXPathEval(xml, xpath);
}
```

SEC-LOOP-tainted-bound

Synopsis

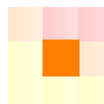
A user-controlled value is used as part of a loop condidition.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

A user-controlled value is used as part of a loop condidition. Unless the bounds of the value used in the condition is checked properly, an attacker might control the number of times a loop executes. This might cause integer overflows or possibly be used in denial of service attacks.   User input used in a loop condition must have its upper and lower bounds checked before used.

Coding standards

CWE 606

Unchecked Input for Loop Condition

Code examples

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

```
void example(void) {
  int a;
  int i = 0;
  scanf("%d", &a);
  while (i < a) {
    i++;
  }
}
```

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

```
void example(void) {
  int a;
  int i = 0;
  scanf("%d", &a);
  if (a > 0 && a < 10) {
    while (i < a) {
      i++;
    }
  }
}
```

SEC-NULL-assignment-fun-pos

Synopsis

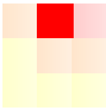
A pointer that might have been assigned the value NULL is dereferenced.

Enabled by default

No

Severity/Certainty

High/Medium



Full description

A pointer that might have been assigned the value NULL, either directly or by a function call that can return NULL, is dereferenced, either directly or by being passed to a function which might dereference it without checking its value. This might cause an application crash. A pointer that might be NULL should be checked before it is 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:

```
#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);
  }
}
```

SEC-NULL-assignment

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. The assignment might be intentional to indicate that the pointer is no longer used, but it is an error to subsequently dereference it, and it might cause an application crash. The pointer should be checked for NULL before it is dereferenced. If the dereference is unintentional, you might want to either assign a value to the pointer or remove the dereference.

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;
}
```

SEC-NULL-cmp-aft

Synopsis

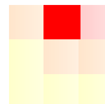
A pointer is dereferenced, then compared with NULL.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

Checks whether a dereferenced pointer are subsequently compared with NULL. Dereferencing a pointer implicitly asserts that it is not NULL. Comparing it with NULL after this may suggests that it may have been NULL at the point of dereference. The pointer should be checked to be non-NULL before being derefenced.

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:

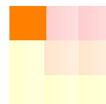
```
#include <stdlib.h>

void example(int *p) {
  if (p == NULL) {
    return;
  }
  *p = 4;
}
```

SEC-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 be caused by an accidental use of the wrong comparison operator, for example == instead of !=, or by accidentally swapping the then- and else-clauses of an if-statement. If the function does dereference the pointer, the application will crash. If it does not, the argument is not needed. Check comparison operators to make sure they test the correct condition, and make sure that branches have not been accidentally swapped.

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 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);
}
```

SEC-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 be caused by an accidental use of the wrong comparison operator, for example == instead of !=, or by accidentally swapping the then- and else- clauses of an if-statement. If the condition is evaluated and found to be true, the application will crash. Check comparison operators to make sure they test the correct condition, and make sure that branches have not been accidentally swapped.

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;
}
```

SEC-NULL-literal-pos

Synopsis

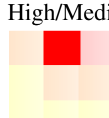
A literal pointer expression (e.g. NULL) is dereferenced by a function call.

Enabled by default

No

Severity/Certainty

High/Medium



Full description

A literal pointer expression (for example, NULL) is passed as an argument to a function that might dereference it. Pointer values are generally only useful if acquired at runtime; thus dereferencing a literal address will usually be an accident, resulting in corrupted memory or an application crash. Make sure that the function being called checks the argument it is given with NULL, before it dereferences it.

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);
}
```

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);
}
```

SEC-STRING-format-string

Synopsis

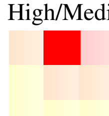
User input is used as a format string.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

User input is used as a format string. An attacker might supply an input string that contains format tokens. Such a string can be used to read and write to arbitrary memory locations, making the attacker able to execute code, crash the application, or access sensitive information stored in memory. User input should be tested, using string comparison or similar, before being used as a format string.

Coding standards

CERT FIO30-C

Exclude user input from format strings

CWE 134

Uncontrolled Format String

Code examples

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

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

int main(char* argc, char** argv) {
  char mystring[100];
  fgets(mystring, 100, stdin);
  char buf[100];
  snprintf(buf, sizeof buf, mystring);
  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(char* argc, char** argv) {
  char mystring[100];
  fgets(mystring, 100, stdin);
  char buf[100];
  snprintf(buf, sizeof buf, "%s", mystring);
  return 0;
}
```

SEC-STRING-hard-coded-credentials

Synopsis

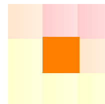
The application hard codes a username or password to connect to an external component.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

The application uses a hard-coded username or password to connect to an external resource, such as a database. An attacker might extract the password from the application binary through an exploit. Or, if the application is indended for client-side use, an attacker could extract the credentials from the binary itself. Credentials should be read into the application using a strongly-protected encrypted configuration file or database. This check supports the following C/C++ SQL libraries: * MySQL C API * MySQL Connector/C++ * libpq (PostgreSQL) * libpq++ (PostgreSQL) * libpqxx (PostgreSQL) * Microsoft ODBC * OLE DB and, also supports Windows Login functions

Coding standards

CWE 798

Use of Hard-coded Credentials

Code examples

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

```
void example(void *conn) {
  char *b;
  char *a = "top_secret_password";
  mysql_real_connect(conn, "localhost", b, a, "FOO", 2000);
}
```

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

```
#include <stdio.h>

void example(void *conn, FILE *f) {
  char *b;
  char *a;
  fscanf(f, "%s;%s", a, b);
  mysql_real_connect(conn, "localhost", b, a, "FOO", 2000);
}
```

MISRAC2004-1.1

Synopsis

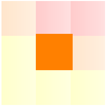
Code was found that does not conform to the ISO/IEC 9899:1990 standard.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) All code shall conform to ISO 9899 standard, with no extensions permitted.

Coding standards

MISRA C:2004 1.1

(Required) All code shall conform to ISO 9899 standard, with no extensions permitted.

Code examples

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

```
struct { int i; }; /* Does not declare anything */
```

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

```
struct named { int i; };
```

MISRAC2004-1.2_a

Synopsis

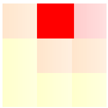
There are read accesses from local buffers that are not preceded by write accesses.

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, which might cause the application to behave erroneously or crash. This check is identical to SPC-uninit-arr-all, MISRAC2012-Rule-9.1_b

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 a[20];
  int b = a[1];
}
```

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

```
extern void f(int*);
void example() {
  int a[20];
  f(a);
  int b = a[1];
}
```

MISRAC2004-1.2_b

Synopsis

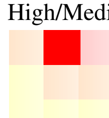
On all execution paths, one or more fields are read from a struct before they are initialized.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

(Required) No reliance shall be placed on undefined or unspecified behavior. Using uninitialized values might cause unexpected results or unpredictable behavior, particularly in the case of pointer fields. This check is identical to SPC-uninit-struct, MISRAC2012-Rule-9.1_c

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. This check is identical to ATH-div-0, MISRAC2012-Rule-1.3_a

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
}
```

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 was found that is assigned the value `0`, and 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. This check is identical to ATH-div-0-assign, MISRAC2012-Rule-1.3_b

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;
}
```

MISRAC2004-1.2_e

Synopsis

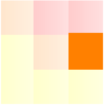
A variable is used as a divisor after a successful comparison with `0`.

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

(Required) No reliance shall be placed on undefined or unspecified behavior. This check is identical to ATH-div-0-cmp-aft, MISRAC2012-Rule-1.3_c, SEC-DIV-0-compare-after

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 is identical to ATH-div-0-cmp-bef, MISRAC2012-Rule-1.3_d, SEC-DIV-0-compare-before

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

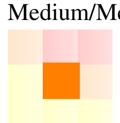
A value that is determined using interval analysis to be `0` is used as a divisor.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) No reliance shall be placed on undefined or unspecified behavior. This check is identical to ATH-div-0-interval, MISRAC2012-Rule-1.3_e

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 might 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. This check is identical to ATH-div-0-pos, MISRAC2012-Rule-1.3_f

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
}
```

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

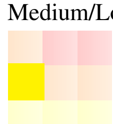
A global variable is not checked against `0` before it is used as a divisor.

Enabled by default

Yes

Severity/Certainty

Medium/Low



Full description

(Required) No reliance shall be placed on undefined or unspecified behavior. This check is identical to ATH-div-0-unchk-global, MISRAC2012-Rule-1.3_g

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

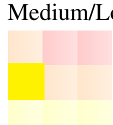
A local variable is not checked against `0` before it is used as a divisor.

Enabled by default

Yes

Severity/Certainty

Medium/Low



Full description

(Required) No reliance shall be placed on undefined or unspecified behavior. This check is identical to ATH-div-0-unchk-local, MISRAC2012-Rule-1.3_h

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

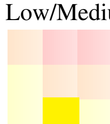
Inline assembler statements were found that are not encapsulated in functions.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) Assembler language shall be encapsulated and isolated. This check is identical to MISRAC++2008-7-4-3, MISRAC2012-Dir-4.3

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 example(int x)
{
  int r;
  asm("");
  return r + 1;
}
```

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

```
int example(int x)
{
    asm("");
    return x;
}
```

MISRAC2004-2.2

Synopsis
Uses of // comments were found.

Enabled by default
Yes

Severity/Certainty
Low/High

Full description
(Required) Source code shall only use /* ... */ style comments.

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
The character sequence `/*` was found inside comments.

Enabled by default
Yes

Severity/Certainty
Low/High

Full description
(Required) The character sequence `/*` shall not be used within a comment. This check is identical to COMMENT-nested, MISRAC++2008-2-7-1

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

Code sections in comments were found, where the comment ends in `;`, `{`, or `}` characters.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

(Advisory) Sections of code should not be commented out. This check is identical to MISRAC2012-Dir-4.4

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:

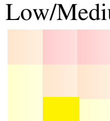
```
void example(void) {
#if 0
   int i;
#endif
}
```

MISRAC2004-5.2

Synopsis
An identifier name was found that is not distinct in the first 31 characters from other names in an outer scope.

Enabled by default
Yes

Severity/Certainty
Low/Medium



Full description
(Required) Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and thus hide that identifier. This check is identical to MISRAC2012-Rule-5.3_c89

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:

```
/*          1234567890123456789012345678901********* */
extern int  n01_param_hides_var__________31x;
extern int  n02_var_hides_var____________31x;
void        n03_var_hides_function_______31x (void) {}

union       n04_var_hides_union_tag______31x {
  int v1;
  unsigned int v2;
};
enum        n05_var_hides_enum_tag_______31x {
            n06_var_hides_enum_const_____31x,
      n07_tag_hides_enum_const_____31x
};
#define     n08_var_hides_macro_name_____31x 123
extern int  n09_label_hides_var__________31x;
extern int  n10_type_hides_var___________31x;

void f1(int n01_param_hides_var__________31y) {
  int       n02_var_hides_var____________31y;
  int       n03_var_hides_function_______31y;
  int       n04_var_hides_union_tag______31y;
  int       n05_var_hides_enum_tag_______31y;
  int       n06_var_hides_enum_const_____31y;
  struct    n07_tag_hides_enum_const_____31y {
    int ff2;
  };
  int       n08_var_hides_macro_name_____31y;
/*
 1234567890123456789012345678901********* */
 n09_label_hides_var__________31y:

  switch(f2()) {
  case 1: {
    typedef int n10_type_hides_var___________31y;
    do {
      /*     1234567890123456789012345678901********* */
      struct n11_var_hides_struct_tag_____31x {
  int ff1;
      };
      if(f3()) {
  int  n11_var_hides_struct_tag_____31y = 1;
      }
    } while(f2());
  }
  }
}
```

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

```
void f1 (void) {
/*           1234567890123456789012345678901********* */
  extern int n01_var_in_same_scope________31x;
  static int n01_var_in_same_scope________31y;

  switch(fn()) {
  case 1:
    {
      int    n02_var_in_different_scope___31a;
    }
    break;
  case 2:
    {
      int    n02_var_in_different_scope___31b;
    }
    break;
  }
  {
      int    n02_var_in_different_scope___31c;
  }
  {
      int    n02_var_in_different_scope___31d;
  }
}
```

MISRAC2004-5.3

Synopsis
A typedef declaration was found with a name already used for a previously declared typedef.

Enabled by default
Yes

Severity/Certainty
Low/Medium

Full description
(Required) A typedef name shall be a unique identifier. This check is identical to MISRAC++2008-2-10-3, MISRAC2012-Rule-5.6

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;

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 was found 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. This check is identical to MISRAC++2008-2-10-4, MISRAC2012-Rule-5.7

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

An identifier is used that might 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. This check is identical to MISRAC++2008-2-10-5

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;
}
```

MISRAC2004-5.6

Synopsis

Identifier reuse in different namespaces

Enabled by default

No

Severity/Certainty

Low/Low

Full description

(Advisory) No identifier in one namespace should have the same spelling as an identifier in another namespace, with the exception of structure member and union member names.

Coding standards

MISRA C:2004 5.6

> (Advisory) No identifier in one namespace should have the same spelling as an identifier in another namespace, with the exception of structure member and union member names.

Code examples

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

```
struct n01_tag_vs_var {
  int n02_field_vs_var;
  int n03_field_vs_func;
} n01_tag_vs_var;

int n04_var_vs_label;

int n02_field_vs_var;

void n03_field_vs_func(void) {
 n04_var_vs_label:
}
```

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

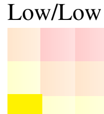
```
struct s {
  int n01_field_vs_field;
};

union u {
  int n01_field_vs_field;
  int u2;
};
```

MISRAC2004-5.7

Synopsis
An identifier in a variable, enumeration, struct, #define, or union definition is reused.

Enabled by default
No

Severity/Certainty
Low/Low



Full description
(Advisory) No identifier name should be reused.

Coding standards
MISRA C:2004 5.7

(Advisory) No identifier name should be reused.

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

```
void example(void) {
  struct {
    int x;
  } name1;
  struct {
    int x; // x is reused here
  } name2;
}
```

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

```
void example(void) {
  struct {
    int x;
  } name1;
  struct {
    int y;
  } name2;
}
```

MISRAC2004-6.1

Synopsis

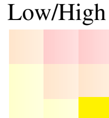
Arithmetic is performed on objects of type plain char, without an explicit signed or unsigned qualifier.

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. This check is identical to MISRAC++2008-4-5-3

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:

```
typedef signed char INT8;
typedef unsigned char UINT8;

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:

```
typedef signed char INT8;
typedef unsigned char UINT8;

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);
}
```

MISRAC2004-6.2

Synopsis

A signed or unsigned char is used on character data.

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

(Required) signed and unsigned char type shall be used only for the storage and use of numeric values.

Coding standards

CERT INT07-C

Use only explicitly signed or unsigned char type for numeric values

MISRA C:2004 6.2

(Required) signed and unsigned char type shall be used only for the storage and use of numeric values.

Code examples

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

```
void example(void) {
  unsigned char c = 'c';
}
```

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

```
void example(void) {
  char c = 'c';
}
```

MISRAC2004-6.3

Synopsis

One or more of the basic types char, int, short, long, double, and float are used without a 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. This check is identical to MISRAC++2008-3-9-2, MISRAC2012-Dir-4.6_a

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 char SCHAR;
typedef int INT;
typedef float FLOAT;

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 char SCHAR;
typedef int INT;
typedef float FLOAT;

INT func(FLOAT f, INT *pi)
{
  INT x;
  INT (*fp)(const SCHAR *);
}
```

MISRAC2004-6.4

Synopsis

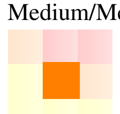
Bitfields of plain int type were found.

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. This check is identical to MISRAC2012-Rule-6.1

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;
};
```

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

```
struct good {
  unsigned int x:3;
};
```

MISRAC2004-6.5

Synopsis

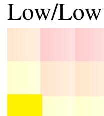
Signed bitfields consisting of a single bit (excluding anonymous fields) were found.

Enabled by default

Yes

Severity/Certainty

Low/Low



Full description

(Required) Bitfields of signed type shall be at least 2 bits long. This check is identical to STRUCT-signed-bit, MISRAC++2008-9-6-4, MISRAC2012-Rule-6.2

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

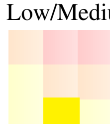
Uses of octal integer constants were found.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) Octal constants shall not be used. Zero is okay This check is identical to MISRAC++2008-2-13-2, MISRAC2012-Rule-7.1

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

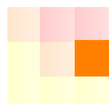
Functions were found that are used despite not having a valid prototype.

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. This check is identical to FUNC-implicit-decl, MISRAC2012-Rule-17.3

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

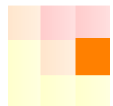
An implicit `int` was found in a declaration.

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. This check is identical to DECL-implicit-int, MISRAC2012-Rule-8.1

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;
}
```

MISRAC2004-8.3

Synopsis

A declaration and definition for a function were found that use different type qualifiers.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.

Coding standards

MISRA C:2004 8.3

(Required) For each function parameter, the type given in the declaration and definition shall be identical and the return types shall also be identical.

Code examples

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

```
/* file2.c
int foo(int i);
*/
void foo(int i) {}
```

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

```
/* file2.c
void foo(int i);
 */
void foo(int i) {}
```

MISRAC2004-8.5_a

Synopsis

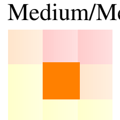
A global variable is declared in a header file.

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

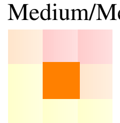
One or more non-inlined functions are defined in header files.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) There shall be no definitions of objects or functions in a header file. This check is identical to MISRAC++2008-3-1-1

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();
}
```

MISRAC2004-8.6

Synopsis
A function declaration was found at block scope.

Enabled by default
Yes

Severity/Certainty
Low/High

Full description
(Required) Functions shall be declared at file scope.

Coding standards
MISRA C:2004 8.6

(Required) Functions shall be declared at file scope.

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

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

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

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

MISRAC2004-8.7

Synopsis

A global object was found that is only referenced from a single function.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Objects shall be defined at block scope if they are only accessed from within a single function.

Coding standards

MISRA C:2004 8.7

> (Required) Objects shall be defined at block scope if they are only accessed from within a single function.

MISRA C:2012 Rule-8.9

> (Advisory) An object should be defined at block scope if its identifier only appears in a single function

Code examples

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

```
static int i = 10;
int example(void) {
  return i;
}
void main() {
  printf("example() = %d\n", example());
}
```

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

```
int example(void) {
  int i = 10;
  return i;
}
void main() {
  printf("example() = %d\n", example());
}
```

MISRAC2004-8.8_a

Synopsis

Multiple declarations of the same external object or function were found.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) An external object or function shall be declared once in one and only one file. This check is identical to MISRAC2012-Rule-8.5_a

Coding standards

MISRA C:2004 8.8

(Required) An external object or function shall be declared in one and only one file.

MISRA C:2012 Rule-8.5

(Required) An external object or function shall be declared once in one and only one file

Code examples

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

```
extern int x;
extern int x;
int x = 1;
```

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

```
extern int x;
int x = 1;
```

MISRAC2004-8.8_b

Synopsis

Multiple declarations of the same external object or function were found.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) An external object or function shall be declared once in one and only one file. This check is identical to MISRAC2012-Rule-8.5_b

Coding standards

MISRA C:2004 8.8

> (Required) An external object or function shall be declared in one and only one file.

MISRA C:2012 Rule-8.5

> (Required) An external object or function shall be declared once in one and only one file

Code examples

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

```
/* file2.c
  extern int foo(int m);
 */
extern int foo(int m);
```

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

```
/* file1.c
   extern int foo( int m );
*/
int foo(int m) {
  return m;
}
```

MISRAC2004-8.9

Synopsis

Multiple definitions or no definition were found for an external object or function.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) An identifier with external linkage shall have exactly one external definition. Note: This check is not part of C-STAT but detected by the IAR linker. This check is identical to MISRAC2012-Rule-8.6

Coding standards

MISRA C:2004 8.9

(Required) An identifier with external linkage shall have exactly one external definition.

MISRA C:2012 Rule-8.6

(Required) An identifier with external linkage shall have exactly one external definition

Code examples

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

```
void example(void) {}
```

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

```
void example(void) {}
```

MISRAC2004-8.10

Synopsis

An externally linked object or function was found referenced in only one translation unit.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required. This check is identical to MISRAC2012-Rule-8.7

Coding standards

MISRA C:2004 8.10

(Required) All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.

MISRA C:2012 Rule-8.7

(Advisory) Functions and objects should not be defined with external linkage if they are referenced in only one translation unit

Code examples

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

```
/* file1.c
static void example (void) {
  // dummy function
}
*/

/* extern linkage */
extern int x;

/* static linkage */
static void foo(void) {
  /* only referenced here */
  x = 1;
}
```

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

```
/* static linkage */
static int x;

/* static linkage */
static void foo(void) {
  /* no linkage */
  int y = (x++);
  if(y < 10)
    foo();
}
```

MISRAC2004-8.12

Synopsis

External arrays are declared without their size being 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. This check is identical to MISRAC++2008-3-1-3, MISRAC2012-Rule-8.11

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

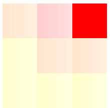
A variable is read before it is assigned a value, on all execution paths.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

(Required) All automatic variables shall have been assigned a value before being used. This check is identical to SPC-uninit-var-all, MISRAC++2008-8-5-1_a, MISRAC2012-Rule-9.1_e, MISRAC2012-Rule-1.3_j

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

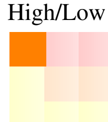
On some execution paths, a variable is read before it is assigned a value.

Enabled by default

Yes

Severity/Certainty

High/Low



Full description

(Required) All automatic variables shall have been assigned a value before being used. This check is identical to SPC-uninit-var-some, MISRAC++2008-8-5-1_b, MISRAC2012-Rule-9.1_f, MISRAC2012-Rule-1.3_k

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

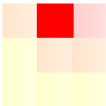
An uninitialized or NULL pointer that is dereferenced was found.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

(Required) All automatic variables shall have been assigned a value before being used. This check is identical to PTR-uninit, MISRAC++2008-8-5-1_c

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

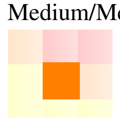
A non-zero array initialization was found that does not exactly match the structure of the array declaration.

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. This check is identical to MISRAC++2008-8-5-2

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][2] = { { 1, 2 }, { 4, 5 } };
}
```

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 } };
}
```

MISRAC2004-9.3

Synopsis

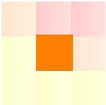
Partially initialized enum.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) In an enumerator list, the `=' construct shall not be used to explicitily initialise members other than the first, unless all items are explicitily initialized.

Coding standards

MISRA C:2004 9.3

> (Required) In an enumerator list, the "=" construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Code examples

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

```
enum E {
  A = 1,
  B = 2,
  C
};
```

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

```
enum E {
  A = 1,
  B,
  C
};
```

MISRAC2004-10.1_a

Synopsis

An expression of integer type was found that is implicitly converted to a narrower or differently signed 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 was found that 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 was found that 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 was found that 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 was found that 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:

```
void example(void) {
  double pc[10];
  float x = pc[5]; // architecture dependent
}
```

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

```
void example(void) {
  unsigned char c;
  float x = c;
}
```

MISRAC2004-10.2_b

Synopsis

An expression of floating type was found that 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:

```
void example(void) {
  float pc[10];
  double x = pc[5] + 5; // architecture dependent
}
```

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 was found that 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:

```
void function(double argument);

void example(void) {
  float x = 4;
  function(x); // architecture dependent
}
```

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 was found that 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:

```
double example(void) {
  float x = 4;
  return x; // architecture dependent
}
```

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

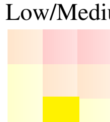
```
double example(void) {
  return 4.0;
}
```

MISRAC2004-10.3

Synopsis
A complex expression of integer type was found that is cast to a wider or differently signed 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 s16a = 3;
  int s16b = 3;

  // arithmetic makes it a complex expression
  long long x = (long long)(s16a + s16b);
}
```

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

```
void example(void) {
   int array[10];

   // A non complex expression is considered safe
   long x = (long)(array[5]);
}
```

MISRAC2004-10.4

Synopsis

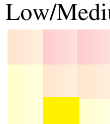
A complex expression of floating type was found that is cast to a wider or different underlying type.

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];
  // architecture dependant
  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]);
}
```

MISRAC2004-10.5

Synopsis

Detected a 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. This check is identical to MISRAC++2008-5-0-10

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;
}
```

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 = ((uint8_t)(~port)) >> 4;
  result_16 = ((uint16_t)(~(uint16_t)port)) >> 4;
}
```

MISRAC2004-10.6

Synopsis

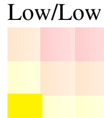
Constants of unsigned type were found that do not have a `U` suffix.

Enabled by default

Yes

Severity/Certainty

Low/Low



Full description

(Required) A U suffix shall be applied to all constants of unsigned type. This check is identical to MISRAC++2008-2-13-3, MISRAC2012-Rule-7.2

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

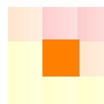
Conversions were found between a pointer to a function and a type other than an integral type.

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 between a pointer type and an integral type was found.

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. This check is identical to MISRAC++2008-5-2-9, MISRAC2012-Rule-11.4

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 was found that 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. This check is identical to MISRAC++2008-5-2-7

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

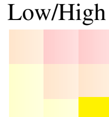
Casts were found that that remove any const or volatile qualification.

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 check is identical to MISRAC++2008-5-2-5, MISRAC2012-Rule-11.8

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

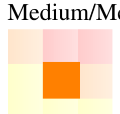
Expressions were found without parentheses, making the operator precedence implicit instead of explicit.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

(Advisory) Limited dependence should be placed on the C operator precedence rules in expressions. This check is identical to MISRAC++2008-5-0-2

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

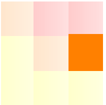
Expressions were found that depend on the order of evaluation.

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. This check is identical to SPC-order, MISRAC++2008-5-0-1_a, MISRAC2012-Rule-13.2_a, MISRAC2012-Rule-1.3_i

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

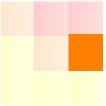
More than one read access with volatile-qualified type was found within one sequence point.

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. This check is identical to SPC-volatile-reads, MISRAC++2008-5-0-1_b, MISRAC2012-Rule-13.2_b

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:

```
void example(void) {
  int x;
  volatile int v;
  x = v + v;
}
```

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

```
int main(void) {
  volatile int i = 0;
  int x = i;
  i++;
  x = x * i;  //OK - statement is broken up
  return 0;
}
```

MISRAC2004-12.2_c

Synopsis

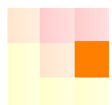
More than one modification access with volatile-qualified type was found within one sequence point.

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. This check is identical to SPC-volatile-writes, MISRAC++2008-5-0-1_c, MISRAC2012-Rule-13.2_c

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:

```
void example(void) {
  int x;
  volatile int 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

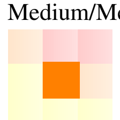
Sizeof expressions were found that contain side effects.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The sizeof operator shall not be used on expressions that contain side effects. The sizeof operator was found used on expressions that contain side effects. This might make it look as if the expression will be evaluated, but because sizeof only operates on the type of the expression, the expression itself is not evaluated. This check is identical to SIZEOF-side-effect, MISRAC++2008-5-3-4

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

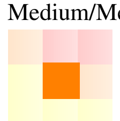
Right-hand operands of && or || were found that contain side effects.

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. This check is identical to MISRAC++2008-5-14-1, MISRAC2012-Rule-13.5

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;
}
```

MISRAC2004-12.5

Synopsis

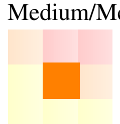
The operands of a logical && or || is not an identifier, a constant, a parenthesized expression or a sequence of the same logical operator.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The operands of a logical && or || shall be primary-expressions.

Coding standards

MISRA C:2004 12.5

(Required) The operands of a logical && or || shall be primary expressions.

Code examples

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

```
int main(void) {
  int a,b;
  if (a > 0 && !b);
}
```

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

```
int main(void) {
  int a,b;
  if ((a > 0) && (!b));
}
```

MISRAC2004-12.6_a

Synopsis

Operands of logical operators (&&, ||, and !) were found 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. This check is identical to MISRAC++2008-5-3-1

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 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:

```
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;

}
```

MISRAC2004-12.6_b

Synopsis

Uses of arithmetic operators on Boolean operands were found.

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 ?:). This check is identical to MISRAC++2008-4-5-1

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;
}
```

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

```
void func()
{
  bool x;
  bool y;
  y = x && y;
}
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;

}
```

MISRAC2004-12.7

Synopsis Applications of bitwise operators to signed operands were found.

Enabled by default Yes

Severity/Certainty Low/Medium

Full description (Required) Bitwise operators shall not be applied to operands whose underlying type is signed. This check is identical to MISRAC++2008-5-0-21

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

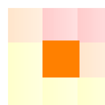
Shifts were found where the right-hand operand might be negative, or too large.

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. This check is identical to ATH-shift-bounds, MISRAC++2008-5-8-1, MISRAC2012-Rule-12.2

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 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;
}
```

MISRAC2004-12.9

Synopsis

Uses of unary minus on unsigned expressions were found.

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. This check is identical to MISRAC2012-Rule-10.1_R8, MISRAC++2008-5-3-2_a

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 were found.

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

(Required) The comma operator shall not be used. This check is identical to MISRAC++2008-5-18-1, MISRAC2012-Rule-12.3

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

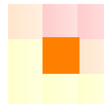
Found a constant unsigned integer expression that overflows.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

(Advisory) Evaluation of constant unsigned integer expressions should not lead to wrap-around. This check is identical to EXPR-const-overflow, MISRAC++2008-5-19-1

Coding standards

CWE 190

Integer Overflow or Wraparound

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

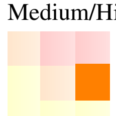
Found a read access to a field of a union following a write access to a different field, which effectively re-interprets the bit pattern with a different type.

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, use an explicit cast. This check is identical to UNION-type-punning

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;
}
```

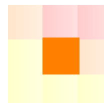
MISRAC2004-12.12_b

Synopsis

An expression was found that 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. This check is identical to MISRAC++2008-3-9-3

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 the increment (++) and decrement (--) operators werew found 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. This check is identical to MISRAC++2008-5-2-10, MISRAC2012-Rule-13.3

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 were found 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 were found 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. This check is identical to MISRAC++2008-5-0-13_a, MISRAC2012-Rule-14.4_a

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:

```
typedef int int32_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

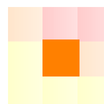
Non-boolean termination conditions were found in `for` loops.

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. This check is identical to MISRAC++2008-5-0-13_b, MISRAC2012-Rule-14.4_b

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 were found 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. This check is identical to MISRAC++2008-5-0-13_c, MISRAC2012-Rule-14.4_c

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 were found 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. This check is identical to MISRAC++2008-5-0-13_d, MISRAC2012-Rule-14.4_d

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 were found.

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. This check is identical to MISRAC++2008-5-0-14

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(bool b) {
  int x;
  x = b ? 1 : 2;   //OK - b is a bool
}
```

MISRAC2004-13.3

Synopsis

Floating-point comparisons using == or != were found.

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

(Required) Floating-point expressions shall not be tested for equality or inequality. This check is identical to ATH-cmp-float, MISRAC++2008-6-2-2

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 were found 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. This check is identical to MISRAC++2008-6-5-1_a, MISRAC2012-Rule-14.1_a

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

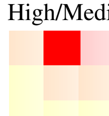
A `for` loop counter variable is not initialized in the `for` loop.

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.

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 was found that 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. This check is identical to MISRAC++2008-6-5-3, MISRAC2012-Rule-14.2

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 >= was found that always evaluates to true.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Boolean operations whose results are invariant shall not be permitted. This check is identical to RED-cmp-always

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 >= was found that always evaluates to false.

Enabled by default — Yes

Severity/Certainty — Low/Medium

Full description — (Required) Boolean operations whose results are invariant shall not be permitted. This check is identical to RED-cmp-never

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

A part of the application is not executed on any of the execution paths.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) There shall be no unreachable code. This check is identical to RED-dead, MISRAC++2008-0-1-1, MISRAC++2008-0-1-9, MISRAC2012-Rule-2.1_b

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

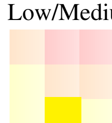
A statement was found that potentially contains no side effects.

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. This check is identical to RED-no-effect, MISRAC2012-Rule-2.2_a

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>

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 There are 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. This check is identical to EXP-stray-semicolon, MISRAC++2008-6-2-3

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 the goto statement were found.

Enabled by default
Yes

Severity/Certainty

Low/Medium

Full description

(Required) The goto statement shall not be used. This check is identical to MISRAC2012-Rule-15.1

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 the continue statement were found.

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 termination points were found in a 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:

```
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;
    }
  }
}
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
    }
  }
}
```

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

```
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
    {
    }
  }
}
void example(void)
{
   int i = 0;
   for (i = 0; i < 10 && i != 9; i++) {
     if (i == 9) {
       break;
     }
   }
}
```

MISRAC2004-14.7

Synopsis

More than one point of exit was found in a function, or an exit point before 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. This check is identical to MISRAC++2008-6-6-5, MISRAC2012-Rule-15.5

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

There are missing braces in one or more 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. This check is identical to MISRAC++2008-6-3-1_a, MISRAC2012-Rule-15.6_a

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

There are missing braces in one or more `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. This check is identical to MISRAC++2008-6-3-1_b, MISRAC2012-Rule-15.6_b

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

There are missing braces in one or more 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. This check is identical to MISRAC++2008-6-3-1_c, MISRAC2012-Rule-15.6_d

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

There are missing braces in one or more 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. This check is identical to MISRAC++2008-6-3-1_d, MISRAC2012-Rule-15.6_e

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

There are missing braces in one or more `if`, `else`, or `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. This check is identical to MISRAC++2008-6-4-1, MISRAC2012-Rule-15.6_c

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

One or more `if ... else if` constructs were found 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. This check is identical to MISRAC++2008-6-4-2, MISRAC2012-Rule-15.7

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 were found 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. This check is identical to MISRAC++2008-6-4-3, MISRAC2012-Rule-16.1

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 were found 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. This check is identical to MISRAC++2008-6-4-4, MISRAC2012-Rule-16.2

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

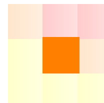
Non-empty switch cases were found that are not terminated by a break statement.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) An unconditional break statement shall terminate every non-empty switch clause. This check is identical to MISRAC++2008-6-4-5, MISRAC2012-Rule-16.3

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) {

  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;
  }

}
```

MISRAC2004-15.3

Synopsis

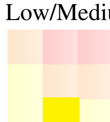
Switch statements were found without a default clause, or with a default clause that is not the final clause.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) The final clause of a switch statement shall be the default clause. This check is identical to MISRAC++2008-6-4-6

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 was found that represents 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. This check is identical to MISRAC++2008-6-4-7, MISRAC2012-Rule-16.7

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 without case clauses were found.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Every switch statement shall have at least one case clause. This check is identical to MISRAC++2008-6-4-8

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

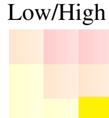
Functions that are defined using ellipsis (...) notation were found.

Enabled by default

Yes

Severity/Certainty

Low/High



Full description

(Required) Functions shall not be defined with a variable number of arguments. This check is identical to MISRAC++2008-8-4-1

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 were found that call themselves directly.

Enabled by default — Yes

Severity/Certainty

Low/Medium

Full description

(Required) Functions shall not call themselves, either directly or indirectly. This check is identical to MISRAC++2008-7-5-4_a, MISRAC2012-Rule-17.2_a

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 were found that call themselves indirectly.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Functions shall not call themselves, either directly or indirectly. This check is identical to MISRAC++2008-7-5-4_b, MISRAC2012-Rule-17.2_b

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 were found that do not give all parameters a name.

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. This check is identical to MISRAC2012-Rule-8.2_b

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", '!');
}
```

MISRAC2004-16.4

Synopsis

The parameter names between the function declaration and definition does not match.

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

(Required) The identifiers used in the declaration and definition of a function shall be identical.

Coding standards

MISRA C:2004 16.4

> (Required) The identifiers used in the declaration and definition of a function shall be identical.

Code examples

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

```
/*
file2.c:
int foo(int b, int a);
 */
int foo(int a, int b)
{
  return a + b;
}
```

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

```
/*
file2.c:
int foo(int a, int b);
 */
int foo(int a, int b)
{
  return a + b;
}
```

MISRAC2004-16.5

Synopsis

Functions were found that are 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. This check is identical to FUNC-unprototyped-all, MISRAC2012-Rule-8.2_a

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 was found that 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. This check is identical to MISRAC++2008-7-1-2

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

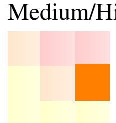
For some execution paths, no return statement is executed in a function with a non-void return type.

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. This check is identical to SPC-return, MISRAC++2008-8-4-3, MISRAC2012-Rule-17.4

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

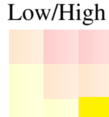
One or more function addresses are taken without an explicit &.

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. This check is identical to MISRAC++2008-8-4-4

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

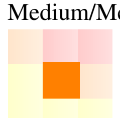
A return value for a library function that might return an error value is not used.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) If a function returns error information, then that error information shall be tested. This check is identical to LIB-return-error, MISRAC++2008-0-3-2

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

A direct access to a field of a struct was found, that uses 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. This check is identical to PTR-arith-field

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

Detected 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. This check is identical to PTR-arith-stack, MISRAC++2008-5-0-16_a

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

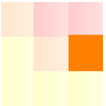
Detected invalid pointer arithmetic with an automatic variable that is neither an array nor a pointer.

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 is identical to PTR-arith-var, MISRAC++2008-5-0-16_b

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;
}
```

MISRAC2004-17.2

Synopsis

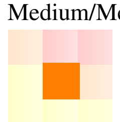
A subtraction was found between pointers that address elements of different arrays.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Pointer subtraction shall only be applied to pointers that address elements of the same array. Note: This rule will only accept arrays of the form '<type> <name>[<size>]'. This check is identical to MISRAC2012-Rule-18.2

Coding standards

MISRA C:2004 17.2

(Required) Pointer subtraction shall only be applied to pointers that address elements of the same array.

MISRA C:2012 Rule-18.2

(Required) Subtraction between pointers shall only be applied to pointers that address elements of the same array

Code examples

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

```
#include <stddef.h>

void example(void) {
  int a[20];
  int b[20];
  int *p1 = &a[5];
  int *p2 = &b[2];
  ptrdiff_t diff;
  diff = p2 - p1;
}
```

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

```
#include <stddef.h>

void example(void) {
  int arr[10];
  int *p1 = &arr[5];
  int *p2 = &arr[5];
  ptrdiff_t diff;
  diff = p2 - p1;
}
```

MISRAC2004-17.3

Synopsis

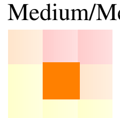
A relational operator was found applied to an object of pointer type that does not point into the same object.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) >, >=, < and <= shall not be applied to pointer types except where they point to the same array. This check is identical to MISRAC2012-Rule-18.3

Coding standards

MISRA C:2004 17.3

> (Required) >, >=, <, <= shall not be applied to pointer types except where they point to the same array.

MISRA C:2012 Rule-18.3

(Required) The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object

Code examples

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

```
void example(void) {
  int a[10];
  int b[10];
  int *p1 = &a[1];
  if (p1 < b) {

  }
}
```

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];
  int *p1 = &a[1];
  if (p1 < a) {

  }
}
```

MISRAC2004-17.4_a

Synopsis

Pointer arithmetic that is not array indexing was detected.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Array indexing shall be the only allowed form of pointer arithmetic. This check is identical to MISRAC++2008-5-0-15_a

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 was detected applied to an object defined as a pointer type.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Array indexing shall be the only allowed form of pointer arithmetic. This check is identical to MISRAC++2008-5-0-15_b

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 char UINT8;
typedef unsigned int UINT;

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 char UINT8;
typedef unsigned int UINT;

void example(void) {
  UINT8 p[10];
  UINT  i;
  for (i = 0; i < 10; i++) {
    p[i] = 0;
  }
}
```

MISRAC2004-17.5

Synopsis

One or more declarations of objects were found that contain 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. This check is identical to MISRAC++2008-5-0-19, MISRAC2012-Rule-18.5

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

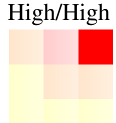
Detected the return of a stack address.

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. This check is identical to MEM-stack, MISRAC++2008-7-5-1_b, MISRAC2012-Rule-18.6_a

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 *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

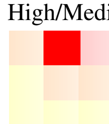
Detected a stack address stored in a global pointer.

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 check is identical to MEM-stack-global, MISRAC++2008-7-5-2_a, MISRAC2012-Rule-18.6_b

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

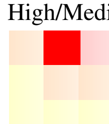
Detected a stack address stored in the field of a global struct.

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 check is identical to MEM-stack-global-field, MISRAC++2008-7-5-2_b, MISRAC2012-Rule-18.6_c

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

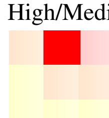
Detected a stack address stored outside a function via a parameter.

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 check is identical to MEM-stack-param, MISRAC++2008-7-5-2_c, MISRAC2012-Rule-18.6_d, MISRAC2012-Rule-1.3_s

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 were found 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

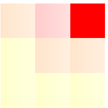
Assignments from one field of a union to another were found.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

(Required) An object shall not be assigned to an overlapping object. This check is identical to UNION-overlap-assign, MISRAC++2008-0-2-1, MISRAC2012-Rule-19.1

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:

```
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;
}
```

MISRAC2004-18.4

Synopsis
Unions were detected.

Enabled by default
Yes

Severity/Certainty
Low/Medium

Full description
(Required) Unions shall not be used. This check is identical to MISRAC++2008-9-5-1, MISRAC2012-Rule-19.2

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;
}
```

MISRAC2004-19.1

Synopsis

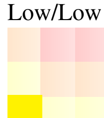
#include directives were found that are not first in the source file.

Enabled by default

No

Severity/Certainty

Low/Low



Full description

(Advisory) #include statements in a file should only be preceded by other preprocessor directives or comments. This check is identical to MISRAC2012-Rule-20.1

Coding standards

MISRA C:2004 19.1

> (Advisory) #include statements in a file should only be preceded by other preprocessor directives or comments.

MISRA C:2012 Rule-20.1

> (Advisory) #include directives should only be preceded by preprocessor directives or comments

Code examples

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

```
void example(void) {}
int x;
#include <cstdio>
void example(void) {}
```

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

```
#include <cstdio>
void example(void) {}
void example(void) {}
```

MISRAC2004-19.2

Synopsis

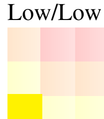
There are illegal characters in header file names.

Enabled by default

No

Severity/Certainty

Low/Low



Full description

(Advisory) Non-standard characters should not occur in header file names in #include directives. This check is identical to MISRAC2012-Rule-20.2

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) {}
```

MISRAC2004-19.4

Synopsis

A macro definition was found that is not permitted.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) C macros shall only expand to a braced initialiser, a constant, a string literal, a parenthesised expression, a type qualifier, a storage class specifier, or a do-while-zero construct.

Coding standards

MISRA C:2004 19.4

> (Required) C macros shall only expand to a braced initializer, a constant, a string literal, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.

Code examples

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

```
#define PLUS_TWO(X) (X) + 2
```

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

```
#define PLUS_TWO(X) ((X) + 2)
```

MISRAC2004-19.5

Synopsis

A #define or #undef was found inside a block.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Macros shall not be #define'd or #undef'd within a block.

Coding standards

MISRA C:2004 19.5

> (Required) Macros shall not be #define'd or #undef'd within a block.

Code examples

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

```
int example() {
#define ONE 1
  return 0;
}
```

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

```
#define ONE 1
int example() {
  return 0;
}
```

MISRAC2004-19.6

Synopsis

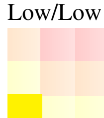
#undef directives were found.

Enabled by default

Yes

Severity/Certainty

Low/Low



Full description

(Required) #undef shall not be used. This check is identical to MISRAC++2008-16-0-3, MISRAC2012-Rule-20.5

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:

```
#define SYM
#undef SYM
```

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

```
#define SYM
```

MISRAC2004-19.7

Synopsis

Function-like macros were detected.

Enabled by default No

Severity/Certainty Low/Low

Full description (Advisory) A function should be used in preference to a function-like macro. This check is identical to MISRAC++2008-16-0-4, MISRAC2012-Dir-4.9

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:

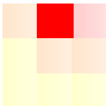
```
template <typename T>
inline T ABS(T x) { return x < 0 ? -x : x; }
```

MISRAC2004-19.10

Synopsis A macro parameter was not enclosed in parentheses or used as the operand of # or ##.

Enabled by default Yes

Severity/Certainty High/Medium



Full description (Required) In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.

Coding standards MISRA C:2004 19.10

(Required) In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.

Code examples

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

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

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

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

MISRAC2004-19.12

Synopsis

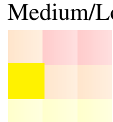
Multiple # or ## preprocessor operators were found in a macro definition.

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 check is identical to DEFINE-hash-multiple, MISRAC++2008-16-3-1

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:

```
#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 */
```

MISRAC2004-19.13

Synopsis

Uses were found of the # and ## operators.

Enabled by default — No

Severity/Certainty — Low/Low

Full description — (Advisory) The # and ## preprocessor operators should not be used. This check is identical to MISRAC++2008-16-3-2, MISRAC2012-Rule-20.10

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:

```
#define A(Y)#Y/* Non-compliant */
```

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

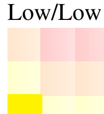
```
#define A(x)(x)/* Compliant */
```

MISRAC2004-19.15

Synopsis — Header files were found 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. This check is identical to MISRAC++2008-16-2-3, MISRAC2012-Dir-4.10

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

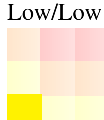
Detected a #define or #undef of a reserved identifier in the standard library.

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. This check is identical to MISRAC++2008-17-0-1, MISRAC2012-Rule-21.1

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 */
```

MISRAC2004-20.2

Synopsis

One or more library functions are being overridden.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The names of standard library macros, objects and functions shall not be reused. This check is identical to MISRAC++2008-17-0-3, MISRAC2012-Rule-21.2

Coding standards

MISRA C:2004 20.2

> (Required) The names of Standard Library macros, objects, and functions shall not be reused.

MISRA C:2012 Rule-21.2

> (Required) A reserved identifier or macro name shall not be declared

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:

```
void example(void) {}
extern "C" void strcpy(void);
void strcpy(void) {}
```

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

```
extern "C" void bar(void);
void foo(void) {}
void example(void) {}
```

MISRAC2004-20.3_a

Synopsis

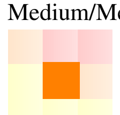
A parameter value (<=0) might cause a domain or range error.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The validity of values passed to library functions shall be checked (>0 case). This check is identical to MISRAC2012-Dir-4.11_a

Coding standards

MISRA C:2004 20.3

(Required) The validity of values passed to library functions shall be checked.

MISRA C:2012 Dir-4.11

(Required) The validity of values passed to library functions shall be checked

Code examples

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

```
#include <math.h>

void gtz(double d1, double d2) {
  double e;
  e = tgamma(-1.0);   /* const not in range */
  e = tgamma(d1);     /* var not checked */
  if(d1 > 0) {
  } else {
    e = tgamma(d1);   /* checked but in wrong branch */
  }
  if(d1 > 0) {
    d1 = d2;
    e = tgamma(d1);   /* checked but updated */
  }
}
```

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

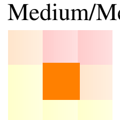
```
#include <math.h>

void example(double d) {
  double e;
  if(d > 0) {
    e = tgamma(d); /* checked before use */
  }
  if(0 < d) {
    e = tgamma(d); /* checked before use */
  }
  if(d <= 0) {
  } else {
    e = tgamma(d); /* checked before use */
  }
  if(0 >= d) {
  } else {
    e = tgamma(d); /* checked before use */
  }
  e = tgamma(1.0); /* constant > 0 */
}
```

MISRAC2004-20.3_b

Synopsis
A parameter value (<0) might cause a domain or range error.

Enabled by default
Yes

Severity/Certainty
Medium/Medium



Full description
(Required) The validity of values passed to library functions shall be checked (>=0 case). This check is identical to MISRAC2012-Dir-4.11_b

Coding standards
MISRA C:2004 20.3

> (Required) The validity of values passed to library functions shall be checked.

MISRA C:2012 Dir-4.11

> (Required) The validity of values passed to library functions shall be checked

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

```
#include <math.h>

void gez(double d1, double d2) {
  double e;
  e = sqrt(-2);     /* const not in range */
  e = sqrt(d1);    /* var not checked */
  if(d1 >= 0) {
  } else {
    e = sqrt(d1); /* checked but in wrong branch */
  }
  if(d1 >= 0) {
    d1 = d2;
    e = sqrt(d1);/* checked but updated */
  }
}
```

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

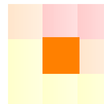
```
#include<math.h>

void gez(double d) {
  double e;
  if(d >= 0) {
    e = sqrt(d); /* checked before use */
  }
  if(0 <= d) {
    e = sqrt(d); /* checked before use */
  }
  if(d < 0) {
  } else {
    e = sqrt(d); /* checked before use */
  }
  if(0 > d) {
  } else {
    e = sqrt(d); /* checked before use */
  }
  e = sqrt(1.0); /* constant > 0 */
}
```

MISRAC2004-20.3_c

Synopsis	A parameter value (==0) might cause a domain or range error.
Enabled by default	Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The validity of values passed to library functions shall be checked (!=0 case). This check is identical to MISRAC2012-Dir-4.11_c

Coding standards

MISRA C:2004 20.3

(Required) The validity of values passed to library functions shall be checked.

MISRA C:2012 Dir-4.11

(Required) The validity of values passed to library functions shall be checked

Code examples

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

```
#include <math.h>

void nez(double d1, double d2) {
  double e;
  e = fmod(1, 0.0);       /* const not in range */
  e = fmod(1, d1);      /* var not checked */
  if(d1 != 0) {
  } else {
    e = fmod(1, d1);    /* checked but in wrong branch */
  }
  if(d1 != 0) {
    d1 = d2;
    e = fmod(1, d1);    /* checked but updated */
  }
}
```

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

```
#include <math.h>

void example(double d) {
  double e;
  if(d != 0) {
    e = logb(d); /* checked before use */
  }
  if(0 != d) {
    e = logb(d); /* checked before use */
  }
  if(d == 0) {
  } else {
    e = logb(d); /* checked before use */
  }
  if(0 == d) {
  } else {
    e = logb(d); /* checked before use */
  }
  e = logb(1.0); /* constant != 0 */
}
```

MISRAC2004-20.3_d

Synopsis

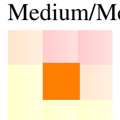
A parameter value (>1) might cause domain or range error.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The validity of values passed to library functions shall be checked (<=1 case). This check is identical to MISRAC2012-Dir-4.11_d

Coding standards

MISRA C:2004 20.3

(Required) The validity of values passed to library functions shall be checked.

MISRA C:2012 Dir-4.11

(Required) The validity of values passed to library functions shall be checked

Code examples

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

```
#include <math.h>

void le1(double d1, double d2) {
  double e;
  e = acos(2);       /* const not in range */
  e = acos(d1);      /* var not checked */
  if(d1 <= 1) {
  } else {
    e = acos(d1);    /* checked but in wrong branch */
  }
  if(d1 <= 1) {
    d1 = d2;
    e = acos(d1);    /* checked but updated */
  }
}
```

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

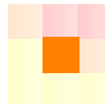
```
#include<math.h>

void example(double d) {
  double e;
  if(d <= 1) {
    e = acos(d); /* checked before use */
  }
  if(1 >= d) {
    e = acos(d); /* checked before use */
  }
  if(d > 1) {
  } else {
    e = acos(d); /* checked before use */
  }
  if(1 < d) {
  } else {
    e = acos(d); /* checked before use */
  }
  e = acos(0.5); /* constant <= 1 */
}
```

MISRAC2004-20.3_e

Synopsis	A parameter value (>=1) might cause domain or range error.
Enabled by default	Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The validity of values passed to library functions shall be checked (<1 case). This check is identical to MISRAC2012-Dir-4.11_e

Coding standards

MISRA C:2004 20.3

(Required) The validity of values passed to library functions shall be checked.

MISRA C:2012 Dir-4.11

(Required) The validity of values passed to library functions shall be checked

Code examples

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

```
#include <math.h>

void lt1(double d1, double d2) {
  double e;
  e = atanh(2.0);       /* const not in range */
  e = atanh(d1);      /* var not checked */
  if(d1 < 1) {
  } else {
    e = atanh(d1);    /* checked but in wrong branch */
  }
  if(d1 < 1) {
    d1 = d2;
    e = atanh(d1);    /* checked but updated */
  }
}
```

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

```
#include<math.h>

void example(double d) {
  double e;
  if(d < 1) {
    e = atanh(d); /* checked before use */
  }
  if(0 > d) {
    e = atanh(d); /* checked before use */
  }
  if(d >= 1) {
  } else {
    e = atanh(d); /* checked before use */
  }
  if(1 <= d) {
  } else {
    e = atanh(d); /* checked before use */
  }
  e = atanh(0.5); /* constant < 1 */
}
```

MISRAC2004-20.3_f

Synopsis

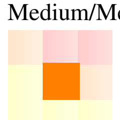
A parameter value (<-1) might cause a domain or range error.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The validity of values passed to library functions shall be checked (>=-1 case). This check is identical to MISRAC2012-Dir-4.11_f

Coding standards

MISRA C:2004 20.3

(Required) The validity of values passed to library functions shall be checked.

MISRA C:2012 Dir-4.11

(Required) The validity of values passed to library functions shall be checked

Code examples

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

```
#include <math.h>

void gen1(double d1, double d2) {
  double e;
  e = acos(-2.0);       /* const not in range */
  e = acos(d1);     /* var not checked */
  if(d1 >= -1) {
  } else {
    e = acos(d1);   /* checked but in wrong branch */
  }
  if(d1 >= -1) {
    d1 = d2;
    e = acos(d1);   /* checked but updated */
  }
}
```

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

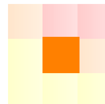
```
#include <math.h>

void example(double d) {
  double e;
  if(d >= -1) {
    e = acos(d); /* checked before use */
  }
  if(-1 <= d) {
    e = acos(d); /* checked before use */
  }
  if(d < -1) {
  } else {
    e = acos(d); /* checked before use */
  }
  if(-1 > d) {
  } else {
    e = acos(d); /* checked before use */
  }
  e = acos(-0.5); /* constant >= -1 */
}
```

MISRAC2004-20.3_g

Synopsis	A parameter value (<=-1) might cause a domain or range error.
Enabled by default	Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The validity of values passed to library functions shall be checked (>-1 case). This check is identical to MISRAC2012-Dir-4.11_g

Coding standards

MISRA C:2004 20.3

(Required) The validity of values passed to library functions shall be checked.

MISRA C:2012 Dir-4.11

(Required) The validity of values passed to library functions shall be checked

Code examples

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

```
#include <math.h>

void gtn1(double d1, double d2) {
  double e;
  e = atanh(-1.5);      /* const not in range */
  e = atanh(d1);     /* var not checked */
  if(d1 > -1) {
  } else {
    e = atanh(d1);   /* checked but in wrong branch */
  }
  if(d1 > -1) {
    d1 = d2;
    e = atanh(d1);   /* checked but updated */
  }
}
```

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

```
#include <math.h>

void example(double d) {
  double e;
  if(d > -1) {
    e = atanh(d); /* checked before use */
  }
  if(-1 < d) {
    e = atanh(d); /* checked before use */
  }
  if(d <= -1) {
  } else {
    e = atanh(d); /* checked before use */
  }
  if(-1 >= d) {
  } else {
    e = atanh(d); /* checked before use */
  }
  e = atanh(-0.5); /* constant > -1 */
}
```

MISRAC2004-20.3_h

Synopsis

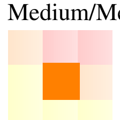
A parameter value (>255) might cause a domain or range error.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The validity of values passed to library functions shall be checked (<=255 case). This check is identical to MISRAC2012-Dir-4.11_h

Coding standards

MISRA C:2004 20.3

(Required) The validity of values passed to library functions shall be checked.

MISRA C:2012 Dir-4.11

(Required) The validity of values passed to library functions shall be checked

Code examples

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

```
extern int isalpha(int c);

void leff(int d1, int d2) {
  int e;
  e = isalpha(2512);   /* const not in range */
  e = isalpha(d1);     /* var not checked */
  if(d1 <= 0xFF) {
  } else {
    e = isalpha(d1);   /* checked but in wrong branch */
  }
  if(d1 <= 255) {
    d1 = d2;
    e = isalpha(d1);   /* checked but updated */
  }
}
```

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

```
extern int isalpha(int c);

void example(int d) {
  int e;
  if(d <= 255) {
    e = isalpha(d); /* checked before use */
  }
  if(0xFF >= d) {
    e = isalpha(d); /* checked before use */
  }
  if(d > 0xFF) {
  } else {
    e = isalpha(d); /* checked before use */
  }
  if(255 < d) {
  } else {
    e = isalpha(d); /* checked before use */
  }
  e = isalpha('c'); /* constant <= 0xFF */
}
```

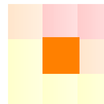
MISRAC2004-20.3_i

Synopsis A parameter value (min) might cause a domain or range error.

Enabled by default Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The validity of values passed to library functions shall be checked (min value case). This check is identical to MISRAC2012-Dir-4.11_i

Coding standards

MISRA C:2004 20.3

(Required) The validity of values passed to library functions shall be checked.

MISRA C:2012 Dir-4.11

(Required) The validity of values passed to library functions shall be checked

Code examples

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

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

void minint(int d1, int d2) {
  int e;
  e = abs(INT_MIN);    /* const not in range */
  e = abs(d1);      /* var not checked */
  if(d1 > INT_MIN) {
  } else {
    e = abs(d1);    /* checked but in wrong branch */
  }
  if(d1 > INT_MIN) {
    d1 = d2;
    e = abs(d1);    /* checked but updated */
  }
}
```

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

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

void example(int d) {
  int e;
  if(d > INT_MIN) {
    e = abs(d); /* checked before use */
  }
  if(INT_MIN < d) {
    e = abs(d); /* checked before use */
  }
  if(d <= INT_MIN) {
  } else {
    e = abs(d); /* checked before use */
  }
  if(INT_MIN >= d) {
  } else {
    e = abs(d); /* checked before use */
  }
  e = abs(INT_MIN+1); /* constant not INT_MIN */
}
```

MISRAC2004-20.4

Synopsis

Detected use of malloc, calloc, realloc, or free.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Dynamic heap memory allocation shall not be used. This check is identical to MISRAC++2008-18-4-1, MISRAC2012-Rule-21.3

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

Detected use of the error indicator errno.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The error indicator errno shall not be used. This check is identical to MISRAC++2008-19-3-1

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 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
Detected use of the built-in function offsetof.

Enabled by default
Yes

Severity/Certainty
Low/Medium

Full description
(Required) The macro offsetof in the stddef.h library shall not be used. This check is identical to MISRAC++2008-18-2-1

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>

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
Detected use 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. This check is identical to MISRAC++2008-17-0-5, MISRAC2012-Rule-21.4

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

Use of signal.h was detected.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The signal handling facilities of signal.h shall not be used. This check is identical to MISRAC++2008-18-7-1, MISRAC2012-Rule-21.5

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

Use of stdio.h was detected.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The input/output library stdio.h shall not be used in production code. This check is identical to MISRAC++2008-27-0-1, MISRAC2012-Rule-21.6

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

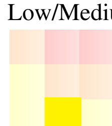
Use of the functions atof, atoi, atol, or atoll was detected.

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. This check is identical to MISRAC++2008-18-0-2, MISRAC2012-Rule-21.7

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

Use of the functions abort, exit, getenv, or system was detected.

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. This check is identical to MISRAC++2008-18-0-3, MISRAC2012-Rule-21.8

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

Use of the time.h functions was detected: asctime, clock, ctime, difftime, gmtime, localtime, mktime, strftime, or time.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The time handling functions of time.h shall not be used. This check is identical to MISRAC++2008-18-0-4, MISRAC2012-Rule-21.10

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 assembler statements were found that are not encapsulated in functions.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Assembly language shall be encapsulated and isolated This check is identical to MISRAC2004-2.1, MISRAC++2008-7-4-3

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 example(int x)
{
  int r;
  asm("");
  return r + 1;
}
```

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

```
int example(int x)
{
  asm("");
  return x;
}
```

MISRAC2012-Dir-4.4

Synopsis

Code sections in comments were found where the comment ends with a ';', '{', or '}' character.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

(Advisory) Sections of code should not be "commented out" Code sections in comments were found where the comment ends with a ';', '{', or '}' character. This check is identical to MISRAC2004-2.4

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-4.5

Synopsis

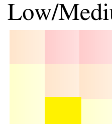
Identifiers in the same namespace, with overlapping visibility, should be typographically unambiguous.

Enabled by default

No

Severity/Certainty

Low/Medium



Full description

(Advisory) Identifiers in the same namespace, with overlapping visibility, should be typographically unambiguous.

Coding standards

MISRA C:2012 Dir-4.5

> (Advisory) Identifiers in the same name space with overlapping visibility should be typographically unambiguous

Code examples

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

```
void example(void) {
  int foo;
  int f00;
}
```

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

```
void example(void) {
  int foo;
  int bar;
}
```

MISRAC2012-Dir-4.6_a

Synopsis

The basic types char, int, short, long, double, and float are used without a 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 The basic types char, int, short, long, double, and float are used without a typedef. Best practice is to use typedefs for portability. This check is identical to MISRAC2004-6.3, MISRAC++2008-3-9-2

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 char SCHAR;
typedef int INT;
typedef float FLOAT;

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 char SCHAR;
typedef int INT;
typedef float FLOAT;

INT func(FLOAT f, INT *pi)
{
  INT x;
  INT (*fp)(const SCHAR *);
}
```

MISRAC2012-Dir-4.6_b

Synopsis

Typedefs of basic types were found with names that do not indicate the size or signedness.

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

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:

```
/* MISRA C 2012 Directive 4.6 Example */

/* Non-compliant - no sign or size specified
*/
typedef int speed_t;
```

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

```
/* MISRA C 2012 Directive 4.6 Example */

/* Compliant    - int used to define specific-length type
*/
typedef int SINT_16;
```

MISRAC2012-Dir-4.7_a

Synopsis

Returned error information should be tested.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

(Required) If a function returns error information, then that error information shall be tested.

Coding standards

CWE 252

Unchecked Return Value

MISRA C:2012 Dir-4.7

(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(5);
}
```

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

```
void example() {
  int p = malloc(5);
}
```

MISRAC2012-Dir-4.7_b

Synopsis

Returned error information should be tested.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

(Required) If a function returns error information, then that error information shall be tested.

Coding standards

CWE 252

Unchecked Return Value

MISRA C:2012 Dir-4.7

(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) {
  int ec = malloc(5);
  ec = 2;
}
```

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

```
void example(void) {
  int ec = malloc(5);
  if (ec)
  {
    // ...
  }
  ec = 2;
}
```

MISRAC2012-Dir-4.7_c

Synopsis

Returned error information should be tested.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

(Required) If a function returns error information, then that error information shall be tested.

Coding standards

CWE 252

Unchecked Return Value

MISRA C:2012 Dir-4.7

(Required) If a function returns error information, then that error information shall be tested

Code examples

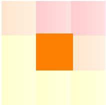
The following code example fails the check and will give a warning:

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

void no_test() {
  FILE * f;
  fpos_t * p;
  int x = fgetpos(f, p);
}

void test_after_overwritten() {
  FILE * f;
  fpos_t * p;
  int x = fgetpos(f, p);

  int y = fgetpos(f, p);

  switch(errno) {
  case 1:
    /* ... */
    break;
  }
}
```

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

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

void test() {
  FILE * f;
  fpos_t * p;
  int x = fgetpos(f, p);

  switch(errno) {
  case 1:
    /* ... */
    break;
  }
}

void test_again() {
  FILE * f;
  fpos_t * p;
  int x = fgetpos(f, p);

  switch(errno) {
  case 1:
    /* ... */
    break;
  }

  x = fgetpos(f, p);

  switch(errno) {
  case 1:
    /* ... */
    break;
  }
}
```

MISRAC2012-Dir-4.8

Synopsis	The implementation of a structure is unnecessarily exposed to a translation unit.
Enabled by default	No
Severity/Certainty	Medium/Medium 

Full description

(Advisory) If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden.

Coding standards

MISRA C:2012 Dir-4.8

> (Advisory) If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden

Code examples

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

```
#include "transparent_struct.h"
/*
transparent_struct.h:
struct t_struct {
  int field;
};
*/

#include "transparent_struct_getset.h"
/*
transparent_struct_getset.h:
struct t_struct * get();
void set(struct t_struct *);
*/

void example() {
  struct t_struct * value = get();
  // struct t_struct * is not derefenced
  set(value);
}
```

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

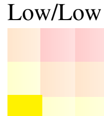
```
#include "opaque_struct.h"
/*
opaque_struct.h:
typedef struct o_struct * structure;
*/

#include "opaque_struct_getset.h"
/*
opaque_struct_getset.h:
structure get();
void set_field(structure, int);
void set(structure);
*/

void example() {
  structure value = get();
  // structure is not derefenced explicitly
  set_field(value, 10);
  set(value);
}
```

MISRAC2012-Dir-4.9

Synopsis
Function-like macros were detected.

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 This check is identical to MISRAC2004-19.7, MISRAC++2008-16-0-4

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

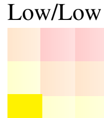
Header files were found without #include guards.

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 This check is identical to MISRAC2004-19.15, MISRAC++2008-16-2-3

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-Dir-4.11_a

Synopsis

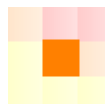
A parameter value (<=0) might cause a domain or range error.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

(Required) The validity of values passed to library functions shall be checked (>0 case). This check is identical to MISRAC2004-20.3_a

Coding standards

MISRA C:2004 20.3

(Required) The validity of values passed to library functions shall be checked.

MISRA C:2012 Dir-4.11

(Required) The validity of values passed to library functions shall be checked

Code examples

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

```
#include <math.h>

void gtz(double d1, double d2) {
  double e;
  e = tgamma(-1.0);    /* const not in range */
  e = tgamma(d1);      /* var not checked */
  if(d1 > 0) {
  } else {
    e = tgamma(d1);    /* checked but in wrong branch */
  }
  if(d1 > 0) {
    d1 = d2;
    e = tgamma(d1);    /* checked but updated */
  }
}
```

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

```
#include <math.h>

void example(double d) {
  double e;
  if(d > 0) {
    e = tgamma(d); /* checked before use */
  }
  if(0 < d) {
    e = tgamma(d); /* checked before use */
  }
  if(d <= 0) {
  } else {
    e = tgamma(d); /* checked before use */
  }
  if(0 >= d) {
  } else {
    e = tgamma(d); /* checked before use */
  }
  e = tgamma(1.0); /* constant > 0 */
}
```

MISRAC2012-Dir-4.11_b

Synopsis

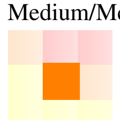
A parameter value (<0) might cause a domain or range error.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

(Required) The validity of values passed to library functions shall be checked (>=0 case). This check is identical to MISRAC2004-20.3_b

Coding standards

MISRA C:2004 20.3

(Required) The validity of values passed to library functions shall be checked.

MISRA C:2012 Dir-4.11

(Required) The validity of values passed to library functions shall be checked

Code examples

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

```
#include <math.h>

void gez(double d1, double d2) {
  double e;
  e = sqrt(-2);     /* const not in range */
  e = sqrt(d1);    /* var not checked */
  if(d1 >= 0) {
  } else {
    e = sqrt(d1); /* checked but in wrong branch */
  }
  if(d1 >= 0) {
    d1 = d2;
    e = sqrt(d1);/* checked but updated */
  }
}
```

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

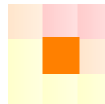
```
#include<math.h>

void gez(double d) {
  double e;
  if(d >= 0) {
    e = sqrt(d); /* checked before use */
  }
  if(0 <= d) {
    e = sqrt(d); /* checked before use */
  }
  if(d < 0) {
  } else {
    e = sqrt(d); /* checked before use */
  }
  if(0 > d) {
  } else {
    e = sqrt(d); /* checked before use */
  }
  e = sqrt(1.0); /* constant > 0 */
}
```

MISRAC2012-Dir-4.11_c

Synopsis	A parameter value (==0) might cause a domain or range error.
Enabled by default	No

Severity/Certainty

Medium/Medium



Full description

(Required) The validity of values passed to library functions shall be checked (!=0 case). This check is identical to MISRAC2004-20.3_c

Coding standards

MISRA C:2004 20.3

> (Required) The validity of values passed to library functions shall be checked.

MISRA C:2012 Dir-4.11

> (Required) The validity of values passed to library functions shall be checked

Code examples

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

```
#include <math.h>

void nez(double d1, double d2) {
  double e;
  e = fmod(1, 0.0);      /* const not in range */
  e = fmod(1, d1);     /* var not checked */
  if(d1 != 0) {
  } else {
    e = fmod(1, d1);   /* checked but in wrong branch */
  }
  if(d1 != 0) {
    d1 = d2;
    e = fmod(1, d1);   /* checked but updated */
  }
}
```

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

```
#include <math.h>

void example(double d) {
  double e;
  if(d != 0) {
    e = logb(d); /* checked before use */
  }
  if(0 != d) {
    e = logb(d); /* checked before use */
  }
  if(d == 0) {
  } else {
    e = logb(d); /* checked before use */
  }
  if(0 == d) {
  } else {
    e = logb(d); /* checked before use */
  }
  e = logb(1.0); /* constant != 0 */
}
```

MISRAC2012-Dir-4.11_d

Synopsis

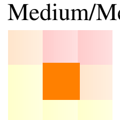
A parameter value (>1) might cause domain or range error.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

(Required) The validity of values passed to library functions shall be checked (<=1 case). This check is identical to MISRAC2004-20.3_d

Coding standards

MISRA C:2004 20.3

(Required) The validity of values passed to library functions shall be checked.

MISRA C:2012 Dir-4.11

(Required) The validity of values passed to library functions shall be checked

Code examples

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

```
#include <math.h>

void le1(double d1, double d2) {
  double e;
  e = acos(2);      /* const not in range */
  e = acos(d1);     /* var not checked */
  if(d1 <= 1) {
  } else {
    e = acos(d1);   /* checked but in wrong branch */
  }
  if(d1 <= 1) {
    d1 = d2;
    e = acos(d1);   /* checked but updated */
  }
}
```

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

```
#include<math.h>

void example(double d) {
  double e;
  if(d <= 1) {
    e = acos(d); /* checked before use */
  }
  if(1 >= d) {
    e = acos(d); /* checked before use */
  }
  if(d > 1) {
  } else {
    e = acos(d); /* checked before use */
  }
  if(1 < d) {
  } else {
    e = acos(d); /* checked before use */
  }
  e = acos(0.5); /* constant <= 1 */
}
```

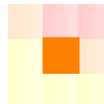
MISRAC2012-Dir-4.11_e

Synopsis: A parameter value (>=1) might cause domain or range error.

Enabled by default: No

Severity/Certainty

Medium/Medium



Full description

(Required) The validity of values passed to library functions shall be checked (<1 case). This check is identical to MISRAC2004-20.3_e

Coding standards

MISRA C:2004 20.3

(Required) The validity of values passed to library functions shall be checked.

MISRA C:2012 Dir-4.11

(Required) The validity of values passed to library functions shall be checked

Code examples

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

```
#include <math.h>

void lt1(double d1, double d2) {
  double e;
  e = atanh(2.0);      /* const not in range */
  e = atanh(d1);      /* var not checked */
  if(d1 < 1) {
  } else {
    e = atanh(d1);    /* checked but in wrong branch */
  }
  if(d1 < 1) {
    d1 = d2;
    e = atanh(d1);    /* checked but updated */
  }
}
```

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

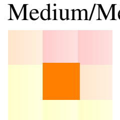
```
#include<math.h>

void example(double d) {
  double e;
  if(d < 1) {
    e = atanh(d); /* checked before use */
  }
  if(0 > d) {
    e = atanh(d); /* checked before use */
  }
  if(d >= 1) {
  } else {
    e = atanh(d); /* checked before use */
  }
  if(1 <= d) {
  } else {
    e = atanh(d); /* checked before use */
  }
  e = atanh(0.5); /* constant < 1 */
}
```

MISRAC2012-Dir-4.11_f

Synopsis
A parameter value (<-1) might cause a domain or range error.

Enabled by default
No

Severity/Certainty
Medium/Medium



Full description
(Required) The validity of values passed to library functions shall be checked (>=-1 case). This check is identical to MISRAC2004-20.3_f

Coding standards
MISRA C:2004 20.3

(Required) The validity of values passed to library functions shall be checked.

MISRA C:2012 Dir-4.11

(Required) The validity of values passed to library functions shall be checked

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

```
#include <math.h>

void gen1(double d1, double d2) {
  double e;
  e = acos(-2.0);       /* const not in range */
  e = acos(d1);     /* var not checked */
  if(d1 >= -1) {
  } else {
    e = acos(d1);   /* checked but in wrong branch */
  }
  if(d1 >= -1) {
    d1 = d2;
    e = acos(d1);   /* checked but updated */
  }
}
```

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

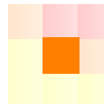
```
#include <math.h>

void example(double d) {
  double e;
  if(d >= -1) {
    e = acos(d); /* checked before use */
  }
  if(-1 <= d) {
    e = acos(d); /* checked before use */
  }
  if(d < -1) {
  } else {
    e = acos(d); /* checked before use */
  }
  if(-1 > d) {
  } else {
    e = acos(d); /* checked before use */
  }
  e = acos(-0.5); /* constant >= -1 */
}
```

MISRAC2012-Dir-4.11_g

Synopsis	A parameter value (<=-1) might cause a domain or range error.
Enabled by default	No

Severity/Certainty

Medium/Medium



Full description

(Required) The validity of values passed to library functions shall be checked (>-1 case). This check is identical to MISRAC2004-20.3_g

Coding standards

MISRA C:2004 20.3

> (Required) The validity of values passed to library functions shall be checked.

MISRA C:2012 Dir-4.11

> (Required) The validity of values passed to library functions shall be checked

Code examples

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

```
#include <math.h>

void gtn1(double d1, double d2) {
  double e;
  e = atanh(-1.5);      /* const not in range */
  e = atanh(d1);     /* var not checked */
  if(d1 > -1) {
  } else {
    e = atanh(d1);   /* checked but in wrong branch */
  }
  if(d1 > -1) {
    d1 = d2;
    e = atanh(d1);   /* checked but updated */
  }
}
```

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

```
#include <math.h>

void example(double d) {
  double e;
  if(d > -1) {
    e = atanh(d); /* checked before use */
  }
  if(-1 < d) {
    e = atanh(d); /* checked before use */
  }
  if(d <= -1) {
  } else {
    e = atanh(d); /* checked before use */
  }
  if(-1 >= d) {
  } else {
    e = atanh(d); /* checked before use */
  }
  e = atanh(-0.5); /* constant > -1 */
}
```

MISRAC2012-Dir-4.11_h

Synopsis

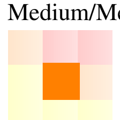
A parameter value (>255) might cause a domain or range error.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

(Required) The validity of values passed to library functions shall be checked (<=255 case). This check is identical to MISRAC2004-20.3_h

Coding standards

MISRA C:2004 20.3

(Required) The validity of values passed to library functions shall be checked.

MISRA C:2012 Dir-4.11

(Required) The validity of values passed to library functions shall be checked

Code examples

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

```
extern int isalpha(int c);

void leff(int d1, int d2) {
  int e;
  e = isalpha(2512);   /* const not in range */
  e = isalpha(d1);     /* var not checked */
  if(d1 <= 0xFF) {
  } else {
    e = isalpha(d1);   /* checked but in wrong branch */
  }
  if(d1 <= 255) {
    d1 = d2;
    e = isalpha(d1);   /* checked but updated */
  }
}
```

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

```
extern int isalpha(int c);

void example(int d) {
  int e;
  if(d <= 255) {
    e = isalpha(d); /* checked before use */
  }
  if(0xFF >= d) {
    e = isalpha(d); /* checked before use */
  }
  if(d > 0xFF) {
  } else {
    e = isalpha(d); /* checked before use */
  }
  if(255 < d) {
  } else {
    e = isalpha(d); /* checked before use */
  }
  e = isalpha('c'); /* constant <= 0xFF */
}
```

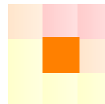
MISRAC2012-Dir-4.11_i

Synopsis A parameter value (min) might cause a domain or range error.

Enabled by default No

Severity/Certainty

Medium/Medium



Full description

(Required) The validity of values passed to library functions shall be checked (min value case). This check is identical to MISRAC2004-20.3_i

Coding standards

MISRA C:2004 20.3

> (Required) The validity of values passed to library functions shall be checked.

MISRA C:2012 Dir-4.11

> (Required) The validity of values passed to library functions shall be checked

Code examples

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

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

void minint(int d1, int d2) {
  int e;
  e = abs(INT_MIN);    /* const not in range */
  e = abs(d1);      /* var not checked */
  if(d1 > INT_MIN) {
  } else {
    e = abs(d1);    /* checked but in wrong branch */
  }
  if(d1 > INT_MIN) {
    d1 = d2;
    e = abs(d1);    /* checked but updated */
  }
}
```

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

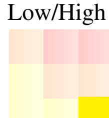
```
#include <math.h>
#include <limits.h>

void example(int d) {
  int e;
  if(d > INT_MIN) {
    e = abs(d); /* checked before use */
  }
  if(INT_MIN < d) {
    e = abs(d); /* checked before use */
  }
  if(d <= INT_MIN) {
  } else {
    e = abs(d); /* checked before use */
  }
  if(INT_MIN >= d) {
  } else {
    e = abs(d); /* checked before use */
  }
  e = abs(INT_MIN+1); /* constant not INT_MIN */
}
```

MISRAC2012-Dir-4.12

Synopsis: Dynamic memory allocation found.

Enabled by default: No

Severity/Certainty: Low/High



Full description: (Required) Dynamic memory allocation shall not be used.

Coding standards: MISRA C:2012 Dir-4.12

(Required) Dynamic 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) {
  int * x = malloc(sizeof(int));
}
void example(void) {
  int * x = new int[10];
}
```

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

```
void example(void) {
  int x[10];
  int * y = x;
}
```

MISRAC2012-Dir-4.13_b

Synopsis

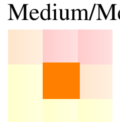
Incorrect deallocation causes memory leak.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Advisory) Functions which are designed to provide operations on a resource should be called in an approriate sequence. 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

MISRA C:2012 Dir-4.13

> (Advisory) Functions which are designed to provide operations on a resource should be called in an appropriate sequence

Code examples

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

```
#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:

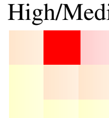
```
#include <stdlib.h>

int main(void) {
    int *ptr = (int*)malloc(sizeof(int));
    if (rand() < 5) {
        free(ptr);
    } else {
        free(ptr);
    }
    return 0;
}
```

MISRAC2012-Dir-4.13_c

Synopsis A file pointer is never closed.

Enabled by default Yes

Severity/Certainty High/Medium



Full description (Advisory) Functions which are designed to provide operations on a resource should be called in an approriate sequence. 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. This check is identical to MISRAC2012-Rule-22.1_b, SEC-FILEOP-open-no-close, RESOURCE-file-no-close-all

Coding standards

MISRA C:2012 Dir-4.13

(Advisory) Functions which are designed to provide operations on a resource should be called in an appropriate sequence

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);
}
```

MISRAC2012-Dir-4.13_d

Synopsis

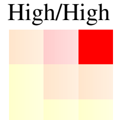
A pointer is used after it has been freed.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

(Advisory) Functions which are designed to provide operations on a resource should be called in an appropriate sequence. 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. This check is identical to SEC-BUFFER-use-after-free-all, MEM-use-free-all

Coding standards

MISRA C:2012 Dir-4.13

(Advisory) Functions which are designed to provide operations on a resource should be called in an appropriate sequence

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
}
```

MISRAC2012-Dir-4.13_e

Synopsis

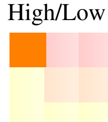
A pointer is used after it has been freed.

Enabled by default

Yes

Severity/Certainty

High/Low



Full description

(Advisory) Functions which are designed to provide operations on a resource should be called in an appropriate sequence. A pointer is used after it has been freed. This might cause data corruption or an application crash. This check is identical to SEC-BUFFER-use-after-free-some, MEM-use-free-some

Coding standards

MISRA C:2012 Dir-4.13

> (Advisory) Functions which are designed to provide operations on a resource should be called in an appropriate sequence

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++;
}
```

MISRAC2012-Dir-4.13_f

Synopsis

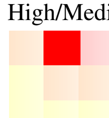
A file resource is used after it has been closed.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

(Advisory) Functions which are designed to provide operations on a resource should be called in an appropriate sequence. 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

MISRA C:2012 Dir-4.13

(Advisory) Functions which are designed to provide operations on a resource should be called in an appropriate sequence

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);
}
```

MISRAC2012-Dir-4.13_g

Synopsis

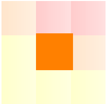
A pointer is freed without having been allocated.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Advisory) Functions which are designed to provide operations on a resource should be called in an appropriate sequence. A pointer is freed without having been allocated.

Coding standards

MISRA C:2012 Dir-4.13

(Advisory) Functions which are designed to provide operations on a resource should be called in an appropriate sequence

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);
}
```

MISRAC2012-Dir-4.13_h

Synopsis

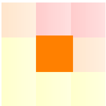
A struct field is deallocated without first having been allocated.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

(Advisory) Functions which are designed to provide operations on a resource should be called in an appropriate sequence. A struct field is deallocated without first having been allocated. This might cause a runtime error.

Coding standards

MISRA C:2012 Dir-4.13

(Advisory) Functions which are designed to provide operations on a resource should be called in an appropriate sequence

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);
}
```

MISRAC2012-Rule-1.3_a

Synopsis

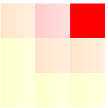
An expression resulting in `0` is used as a divisor.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

(Required) There shall be no occurrence of undefined or critical unspecified behavior. This check is identical to ATH-div-0, MISRAC2004-1.2_c

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(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_b

Synopsis

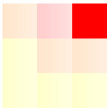
A variable was found that is assigned the value `0`, and then used as a divisor.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

(Required) There shall be no occurrence of undefined or critical unspecified behavior. This check is identical to ATH-div-0-assign, MISRAC2004-1.2_d

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(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;
}
```

MISRAC2012-Rule-1.3_c

Synopsis

A variable is used as a divisor after a successful comparison with `0`.

Enabled by default

Yes

Severity/Certainty

Medium/High

Full description

(Required) There shall be no occurrence of undefined or critical unspecified behavior. This check is identical to ATH-div-0-cmp-aft, MISRAC2004-1.2_e, SEC-DIV-0-compare-after

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:

```
#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 is identical to ATH-div-0-cmp-bef, MISRAC2004-1.2_f, SEC-DIV-0-compare-before

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

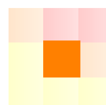
A value that is determined using interval analysis to be `0` is used as a divisor.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) There shall be no occurrence of undefined or critical unspecified behavior. This check is identical to ATH-div-0-interval, MISRAC2004-1.2_g

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(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

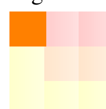
An expression that might be `0` is used as a divisor.

Enabled by default

Yes

Severity/Certainty

High/Low



Full description

(Required) There shall be no occurrence of undefined or critical unspecified behavior. This check is identical to ATH-div-0-pos, MISRAC2004-1.2_h

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(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 behavior. This check is identical to ATH-div-0-unchk-global, MISRAC2004-1.2_i

Coding standards

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 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 behavior. This check is identical to ATH-div-0-unchk-local, MISRAC2004-1.2_j

Coding standards 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 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;
  }
}
```

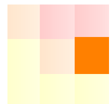
MISRAC2012-Rule-1.3_i

Synopsis Expressions found that 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. This check is identical to MISRAC2004-12.2_a, MISRAC++2008-5-0-1_a, MISRAC2012-Rule-13.2_a, SPC-order

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

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-1.3_j

Synopsis

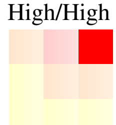
A variable is read before it is assigned a value.

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. This check is identical to MISRAC2004-9.1_a, MISRAC++2008-8-5-1_a, MISRAC2012-Rule-9.1_e, SPC-uninit-var-all

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:

```
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-1.3_k

Synopsis

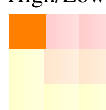
A variable is read before it is assigned a value.

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. This check is identical to MISRAC2004-9.1_b, MISRAC++2008-8-5-1_b, MISRAC2012-Rule-9.1_f, SPC-uninit-var-some

Coding standards

CWE 457

Use of Uninitialized Variable

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-1.3_m

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:

```
int foo(int x, int y){
  return x+y;
}

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:

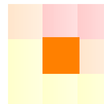
```
typedef int (*fptr)(int,int);

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;
}
```

MISRAC2012-Rule-1.3_n

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;
}
```

MISRAC2012-Rule-1.3_o

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. This check is identical to SEC-BUFFER-use-after-free-all, MEM-use-free-all

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
}
```

MISRAC2012-Rule-1.3_p

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. This check is identical to SEC-BUFFER-use-after-free-some, MEM-use-free-some

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++;
}
```

MISRAC2012-Rule-1.3_q

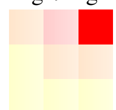
Synopsis

Might return an 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 stack frame 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

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

```
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

}
```

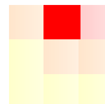
MISRAC2012-Rule-1.3_r

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

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-1.3_s

Synopsis

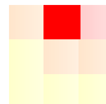
A stack address is stored outside a function via a parameter.

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. This check is identical to MISRAC2004-17.6_d, MISRAC++2008-7-5-2_c, MISRAC2012-Rule-18.6_d, MEM-stack-param

Coding standards

CERT DCL30-C

Declare objects with appropriate storage durations

CWE 466

Return of Pointer Value Outside of Expected Range

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-1.3_t

Synopsis

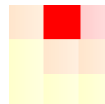
A call to `memcpy` or `memmove` causes the memory to overrun.

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

CWE 676

Use of Potentially Dangerous Function

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, sizeof(int) * (size + 1));
}
```

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

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

void func()
{
  int arr[10];
  int * ptr = (int *)malloc(sizeof(int) * 10);
  memcpy(ptr, arr, sizeof(int) * 10);
}
```

MISRAC2012-Rule-1.3_u

Synopsis

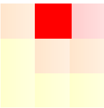
A call to `memset` causes a buffer overrun.

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

CWE 676

Use of Potentially Dangerous Function

CWE 122

Heap-based Buffer Overflow

CWE 121

Stack-based Buffer Overflow

CWE 119

Improper Restriction of Operations within the Bounds of a Memory Buffer

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 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);
}
```

MISRAC2012-Rule-1.3_v

Synopsis

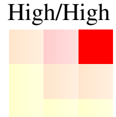
A call to `strcpy` causes a destination buffer overrun.

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

CWE 676

Use of Potentially Dangerous Function

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:

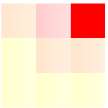
```
#include <string.h>
#include <stdlib.h>

void example(void)
{
  char *str1 = "Hello World!\n";
  char *str2 = (char *)malloc(14);
  strcpy(str2,str1);
}
```

MISRAC2012-Rule-1.3_w

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

CWE 122

Heap-based Buffer Overflow

CWE 676

Use of Potentially Dangerous Function

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);
}
```

MISRAC2012-Rule-2.1_a

Synopsis

A case statement within a switch statement cannot be reached.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) A project shall not contain unreachable code. This check is identical to RED-case-reach, MISRAC++2008-0-1-2_c

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

A part of the application is never executed.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) A project shall not contain unreachable code. This check is identical to RED-dead, MISRAC2004-14.1, MISRAC++2008-0-1-1, MISRAC++2008-0-1-9

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 potentially contains no side effects.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) There shall be no dead code. This check is identical to RED-no-effect, MISRAC2004-14.2

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>

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_b

Synopsis	A field in a struct is assigned a non-trivial value that is never used.
Enabled by default	Yes

Severity/Certainty

Low/Medium

Full description

(Required) There shall be no dead code. This check is identical to

Coding standards

CERT MSC13-C

Detect and remove unused values

CWE 563

Unused Variable

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:

```
typedef struct simpleStruct {
    int a;
} ss_t;

void example(void) {
    ss_t data;
    data.a = 0;
}
```

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

```
extern void foo(int num);

typedef struct simpleStruct {
    int a;
} ss_t;

void example(void) {
    ss_t data;
    data.a = 0;
    foo(data.a);
}
```

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. This check is identical to RED-unused-val, MISRAC++2008-0-1-6

Coding standards

CWE 563

Unused Variable

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;
}
```

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

```
int example(void) {
  int x;
  x = 20;
  return x;
}
```

MISRAC2012-Rule-2.3

Synopsis

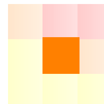
Unused type declaration.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

(Advisory) A project should not contain unused type declarations.

Coding standards

MISRA C:2012 Rule-2.3

(Advisory) A project should not contain unused type declarations

Code examples

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

```
typedef int unused;
```

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

```
typedef int used;
used name;
```

MISRAC2012-Rule-2.4

Synopsis

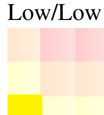
Unused tag declarations were found.

Enabled by default

No

Severity/Certainty

Low/Low



Full description

(Advisory) A project should not contain unused tag declarations.

Coding standards

MISRA C:2012 Rule-2.4

(Advisory) A project should not contain unused tag declarations

Code examples

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

```
struct abc {
  int x;
};

void foo(void) {
  /* not using abc */
}
```

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

```
struct abc {
  int x;
};

void foo(void) {
  struct abc m;
}
```

MISRAC2012-Rule-2.5

Synopsis An unused macro declaration was found.

Enabled by default No

Severity/Certainty Low/Medium

Full description (Advisory) A project should not contain unused macro declarations.

Coding standards MISRA C:2012 Rule-2.5

(Advisory) A project should not contain unused macro declarations

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

```
#define M(x) (x + 1)

void example(void) {
  /* not invoking M */
}
```

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

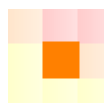
```
#define M(x) (x + 1)

void example(void) {
  /* invoking M */
  int x = M(1);
}
```

MISRAC2012-Rule-2.6

Synopsis
A function was found that contains an unused label declaration.

Enabled by default
No

Severity/Certainty
Medium/Medium



Full description
(Advisory) A function should not contain unused label declarations.

Coding standards
MISRA C:2012 Rule-2.6

(Advisory) A function should not contain unused label declarations

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

```
void example(void) {
unusedlabel:
}
```

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

```
void skip_funcion_call(void);

void example(void) {
  goto usedlabel;
  skip_funcion_call();
usedlabel:
}
```

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. This check is identical to RED-unused-param, MISRAC++2008-0-1-11

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:

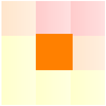
```
int example(int x) {
  return x + 20;
}
```

MISRAC2012-Rule-3.1

Synopsis The character sequences /* and // were found 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-3.2

Synopsis

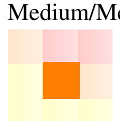
Line-splicing was found in // comments.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Line-splicing shall not be used in // comments.

Coding standards

MISRA C:2012 Rule-3.2

(Required) Line-splicing shall not be used in // comments

Code examples

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

```
// This comment \
has a line splice
```

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

```
// This comment
// has no line splice
```

MISRAC2012-Rule-5.1

Synopsis

An external identifier was found that is not unique for the first 31 characters, but still not identical to another identifier.

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:

```
/* file2.c
int ABC;
 */
int ABC;

void example (void) {
}
```

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

```
/* file2.c
int ABC;
 */
int a;

void example (void) {

}
```

MISRAC2012-Rule-5.2_c89

Synopsis

Identifier names were found that are not distinct in their first 31 characters from other names in the same scope.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Identifiers declared in the same scope and name space shall be distinct.

Coding standards

MISRA C:2012 Rule-5.2

> (Required) Identifiers declared in the same scope and name space shall be distinct

Code examples

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

```
/*         1234567890123456789012345678901********* */
extern int n01_var_hides_var____________31x;
static int n01_var_hides_var____________31y;

/*         1234567890123456789012345678901********* */
static int n02_function_hides_var_______31x;
void       n02_function_hides_var_______31y (void) {}

void foo(void) {
  int i;
  switch(f1()) {
  case 1: {
      do {
  for(i = 0; i < 10; i++) {
    if(f3()) {
      /*     1234567890123456789012345678901********* */
      int    n03_var_hides_var____________31x;
      int    n03_var_hides_var____________31y;
    }
  }
      } while(f2());
    }
  }
}

/*   1234567890123456789012345678901********* */
enum E {
     n04_var_hides_enum_const_____31x,
};

/*   1234567890123456789012345678901********* */
int  n04_var_hides_enum_const_____31y;

/*           1234567890123456789012345678901********* */
void bar(int n05_var_hides_parameter______31x) {
  int        n05_var_hides_parameter______31y;
}

/*      1234567890123456789012345678901********* */
#define n06_var_hides_macro_name_____31x 123
int     n06_var_hides_macro_name_____31y;

/*          1234567890123456789012345678901********* */
int         n07_type_hides_var___________31x;
typedef int n07_type_hides_var___________31y;

/*    1234567890123456789012345678901********* */
```

```
union U {
  int n08_field_hides_field________31x;
  int n08_field_hides_field________31y;
};

struct S {
  int n09_field_hides_field________31x;
  int n09_field_hides_field________31y;
};
```

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

```
/*           1234567890123456789012345678901********* */
extern int   n01_var_in_different_scope___31x;
void         n02_different_function_name__31x (void) {
  static int n01_var_in_different_scope___31y;

  switch(fn()) {
  case 1:
    {
      int    n01_var_in_different_scope___31a;
    }
    break;
  case 2:
    {
      int    n01_var_in_different_scope___31b;
    }
    break;
  }
  {
      int    n01_var_in_different_scope___31c;
  }
  {
      int    n01_var_in_different_scope___31d;
  }
}

/* exception for typedef of tag name*/
typedef struct s1 {
  int sf1;
} s1;

typedef union u1 {
  int uf1;
  int uf2;
} u1;

typedef enum e1 {
  ec1, ec2
} e1;

/* identifiers in different name spaces */
/*    1234567890123456789012345678901********* */
union n02_var_hides_union_tag______31x {
  int v1;
  unsigned int v2;
}     n02_var_hides_union_tag______31y;

/*   1234567890123456789012345678901********* */
```

```
enum n03_var_hides_enum_tag_______31x {
     n04_tag_hides_enum_const_____31x
};

/*   1234567890123456789012345678901********* */
int  n03_var_hides_enum_tag_______31y;

/*     1234567890123456789012345678901********* */
struct n04_tag_hides_enum_const_____31y {
  int ff2;
};

void foo() {
/*    1234567890123456789012345678901********* */
  int n05_label_hides_var__________31x;
  {
/*1234567890123456789012345678901********* */
  n05_label_hides_var__________31y:
    n05_label_hides_var__________31x = 1;
  }
}

void bar(void) {
  int i;
  switch(f1()) {
  case 1: {
      do {
   for(i = 0; i < 10; i++) {
     if(f3()) {
       /*     1234567890123456789012345678901********* */
       struct n06_var_hides_struct_tag_____31x {
         int f1;
       }      n06_var_hides_struct_tag_____31y;
     }
   }
      } while(f2());
     }
  }
}
```

MISRAC2012-Rule-5.2_c99

Synopsis
: Identifier names were found that are not distinct in their first 63 characters from other names in the same scope.

Enabled by default
: Yes

Severity/Certainty Low/Medium

Full description (Required) Identifiers declared in the same scope and name space shall be distinct.

Coding standards MISRA C:2012 Rule-5.2

(Required) Identifiers declared in the same scope and name space shall be distinct

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

```
/*         0        1         2         3         4         5
6     */
/*
123456789012345678901234567890123456789012345678901234567890123*
*/
extern int
n01_var_hides_var___________________________________________63x;
static int
n01_var_hides_var___________________________________________63y;

/*         0        1         2         3         4         5
6     */
/*
123456789012345678901234567890123456789012345678901234567890123*
*/
static int
n02_function_hides_var______________________________________63x;
void
n02_function_hides_var______________________________________63y
(void) {}

void foo(void) {
  int i;
  switch(f1()) {
  case 1: {
      do {
   for(i = 0; i < 10; i++) {
     if(f3()) {
/*               0        1         2         3         4
5         6     */
/*
123456789012345678901234567890123456789012345678901234567890123*
*/
      int
n03_var_hides_var___________________________________________63x;
      int
n03_var_hides_var___________________________________________63y;
     }
   }
      } while(f2());
    }
  }
}

/*   0        1         2         3         4         5         6
*/
/*
```

```
1234567890123456789012345678901234567890123456789012345678901234567890123*
*/
enum E {

n04_var_hides_enum_const_____________________________________63x
};

/*   0        1        2        3        4        5        6
*/
/*
1234567890123456789012345678901234567890123456789012345678901234567890123*
*/
int
n04_var_hides_enum_const_____________________________________63y;

/*          0       1        2        3        4        5
6     */
/*
1234567890123456789012345678901234567890123456789012345678901234567890123*
*/
void bar(int
n05_var_hides_parameter______________________________________63x)
{
  int
n05_var_hides_parameter______________________________________63y;
}

/*      0        1         2         3         4         5
6     */
/*
1234567890123456789012345678901234567890123456789012345678901234567890123*
*/
#define
n06_var_hides_macro_name_____________________________________63x
123
int
n06_var_hides_macro_name_____________________________________63y;

/*         0       1        2        3        4        5
6     */
/*
1234567890123456789012345678901234567890123456789012345678901234567890123*
*/
int
n07_type_hides_var___________________________________________63x;
typedef int
n07_type_hides_var___________________________________________63y;
```

```
/*     0         1         2         3         4         5
6     */
/*
123456789012345678901234567890123456789012345678901234567890123*
*/
union U {
  int
n08_field_hides_field________________________________________63x;
  int
n08_field_hides_field________________________________________63y;
};

struct S {
  int
n09_field_hides_field________________________________________63x;
  int
n09_field_hides_field________________________________________63y;
};
```

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

```
/*          0       1        2        3        4        5
6             */
/*
1234567890123456789012345678901234567890123456789012345678901234567890123**
******* */
extern int
n01_var_in_different_scope___________________________________63x;
void
n02_different_function_name__________________________________63x
(void) {
  static int
n01_var_in_different_scope___________________________________63y;

  switch(fn()) {
  case 1:
    {
      int
n01_var_in_different_scope___________________________________63a;
    }
    break;
  case 2:
    {
      int
n01_var_in_different_scope___________________________________63b;
    }
    break;
  }
  {
      int
n01_var_in_different_scope___________________________________63c;
  }
  {
      int
n01_var_in_different_scope___________________________________63d;
  }
}

/*         0       1        2        3        4        5
6     */
/*
1234567890123456789012345678901234567890123456789012345678901234567890123*
*/
void
n12_var_hides_function_different_scope_______________________63x
(void) {
static int
n12_var_hides_function_different_scope_______________________63y;
```

```
}

/* exception for typedef of tag name*/
typedef struct s1 {
  int sf1;
} s1;

typedef union u1 {
  int uf1;
  int uf2;
} u1;

typedef enum e1 {
  ec1, ec2
} e1;

/* identifiers in different name spaces */
void foo(void) {
  int i;
  switch(f1()) {
  case 1: {
      do {
   for(i = 0; i < 10; i++) {
     if(f3()) {
/*                0        1         2         3         4
5         6     */
/*
1234567890123456789012345678901234567890123456789012345678901 23*
*/
      struct
n03_var_hides_struct_tag_____________________________________63x
{
        int f1;
      }
n03_var_hides_struct_tag_____________________________________63y;
    }
   }
      } while(f2());
    }
  }
}

/*    0        1         2         3         4         5
6     */
/*
1234567890123456789012345678901234567890123456789012345678901 23*
*/
```

```
union
n04_var_hides_union_tag______________________________________63x
{
  int v1;
  unsigned int v2;
}
n04_var_hides_union_tag______________________________________63y;

/*   0        1         2         3         4         5         6
*/
/*
1234567890123456789012345678901234567890123456789012345678901234567890123*
*/
enum
n05_var_hides_enum_tag_______________________________________63x
{

n07_tag_hides_enum_const_____________________________________63x
};

/*   0        1         2         3         4         5         6
*/
/*
1234567890123456789012345678901234567890123456789012345678901234567890123*
*/
int
n05_var_hides_enum_tag_______________________________________63y;

struct

n07_tag_hides_enum_const_____________________________________63y
{
  int sf2;
};

void bar(void) {
/*    0        1         2         3         4         5
6     */
/*
1234567890123456789012345678901234567890123456789012345678901234567890123*
*/
  int
n09_label_hides_var__________________________________________63x;
  {
/*0        1         2         3         4         5         6
*/
/*1234567890123456789012345678901234567890123456789012345678901234567890123
```

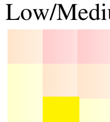
```
* */

n09_label_hides_var__________________________________________63y:

n09_label_hides_var__________________________________________63x
= 1;
  }
}
```

MISRAC2012-Rule-5.3_c89

Synopsis Identifier names were found that are not distinct in their first 31 characters from other names in an outer scope.

Enabled by default Yes

Severity/Certainty Low/Medium



Full description (Required) An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.

Coding standards 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:

```
/*          1234567890123456789012345678901********* */
extern int  n01_param_hides_var__________31x;
extern int  n02_var_hides_var____________31x;
void        n03_var_hides_function_______31x (void) {}

enum E {
            n04_var_hides_enum_const_____31x,
};
#define     n05_var_hides_macro_name_____31x 123
extern int  n06_type_hides_var___________31x;

void f1(int n01_param_hides_var__________31y) {
  int       n02_var_hides_var____________31y;
  int       n03_var_hides_function_______31y;
  int       n04_var_hides_enum_const_____31y;
  int       n05_var_hides_macro_name_____31y;

  switch(f2()) {
  case 1: {
    typedef int n06_type_hides_var___________31y;
    do {
      /*     1234567890123456789012345678901********* */
      int n07_var_hides_var_______________31x;
      if(f3()) {
   int  n07_var_hides_var_______________31y = 1;
      }
    } while(f2());
  }
  }
}
```

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

```
int f1 (void) {
/*           1234567890123456789012345678901********* */
  extern int n01_var_in_same_scope________31x;
  static int n01_var_in_same_scope________31y;

  switch(fn()) {
  case 1:
    {
      int    n02_var_in_different_scope___31a;
    }
    break;
  case 2:
    {
      int    n02_var_in_different_scope___31b;
    }
    break;
  }
  {
      int    n02_var_in_different_scope___31c;
  }
  {
      int    n02_var_in_different_scope___31d;
  }
  return 0;
}

/* identifiers in different name spaces */
/*          1234567890123456789012345678901********* */
union       n03_var_hides_union_tag______31x {
  int v1;
  unsigned int v2;
};
enum        n04_var_hides_enum_tag_______31x {
      n05_tag_hides_enum_const_____31x
};
extern int  n06_label_hides_var__________31x;

int f2(void) {
  int       n03_var_hides_union_tag______31y;
  int       n04_var_hides_enum_tag_______31y;
  struct    n05_tag_hides_enum_const_____31y {
    int ff2;
  };
/*
 1234567890123456789012345678901********* */
 n06_label_hides_var__________31y:
```

```
  switch(f2()) {
  case 0: {
    do {
      /*     1234567890123456789012345678901********* */
      struct n07_var_hides_struct_tag_____31x {
  int ff1;
      };
      if(f3()) {
  int  n07_var_hides_struct_tag_____31y = 1;
      }
    } while(f2());
  }
  }
  return 0;
}
```

MISRAC2012-Rule-5.3_c99

Synopsis

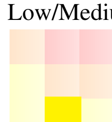
Identifier names were found that are not distinct in their first 63 characters from other names in an outer scope.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.

Coding standards

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:

```
/*         0        1         2         3         4         5
6     */
/*
123456789012345678901234567890123456789012345678901234567890123*
*/
extern int
n01_param_hides_var_________________________________________63x;
extern int
n02_var_hides_var___________________________________________63x;
void
n03_var_hides_function______________________________________63x
(void) {}

enum E {

n04_var_hides_enum_const____________________________________63x
};
#define
n05_var_hides_macro_name____________________________________63x
123
extern int
n06_type_hides_var__________________________________________63x;

void f1(int
n01_param_hides_var_________________________________________63y)
{
  int
n02_var_hides_var___________________________________________63y;
  int
n03_var_hides_function______________________________________63y;
  int
n04_var_hides_enum_const____________________________________63y;
  int
n05_var_hides_macro_name____________________________________63y;

  switch(f2()) {
  case 1: {
/*             0        1         2         3         4
5         6     */
/*
123456789012345678901234567890123456789012345678901234567890123*
*/
    typedef int
n06_type_hides_var__________________________________________63y;
    do {
      int
n07_var_var_________________________________________________63x;
```

```
      if(f3()) {
   int
n07_var_var_____________________________________________________63y
= 1;
      }
    } while(f2());
  }
  }
}
```

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

```
int f1 (void) {
/*         0         1         2         3         4         5
6     */
/*
123456789012345678901234567890123456789012345678901234567890123*
*/
  extern int
n01_var_in_same_scope_______________________________________63x;
  static int
n01_var_in_same_scope_______________________________________63y;

  switch(fn()) {
  case 1:
    {
      int
n02_var_in_different_scope__________________________________63a;
    }
    break;
  case 2:
    {
      int
n02_var_in_different_scope__________________________________63b;
    }
    break;
  }
  {
      int
n02_var_in_different_scope__________________________________63c;
  }
  {
      int
n02_var_in_different_scope__________________________________63d;
  }
  return 1;
}

/* identifiers in different name spaces */
/*         0         1         2         3         4         5
6     */
/*
123456789012345678901234567890123456789012345678901234567890123*
*/
union
n03_var_hides_union_tag_____________________________________63x
{
  int v1;
  unsigned int v2;
```

```
};
enum
n04_var_hides_enum_tag_______________________________________63x
{

n05_tag_hides_enum_const_____________________________________63x
};
extern int
n06_label_hides_var__________________________________________63x;

int f2(void) {
  int
n03_var_hides_union_tag______________________________________63y;
  int
n04_var_hides_enum_tag_______________________________________63y;
  struct
n05_tag_hides_enum_const_____________________________________63y
{
    int ff2;
  };
/*
 0         1         2         3         4         5         6
 123456789012345678901234567890123456789012345678901234567890123*
*/

n06_label_hides_var__________________________________________63y:

  switch(f2()) {
  case 1: {
/*              0        1         2         3         4
5         6     */
/*
123456789012345678901234567890123456789012345678901234567890123*
*/
    do {
      struct
n07_var_hides_struct_tag_____________________________________63x
{
  int ff1;
      };
      if(f3()) {
  int
n07_var_hides_struct_tag_____________________________________63y
= 1;
      }
    } while(f2());
  }
```

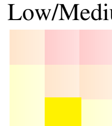
```
  }
  return 0;
}
```

MISRAC2012-Rule-5.4_c89

Synopsis
Macro names were found 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:

```
/*          1234567890123456789012345678901***   */
#define     n01_macro_hides_macro________31x 1
#define     n02_param_hides_macro________31x 1
#define     n03_macro_hides_param________31x 1

#define     n01_macro_hides_macro________31y 2
#define m1(n02_param_hides_macro________31y)
(n01_param_hides_macro________31y + 1)
#define     n03_macro_hides_param________31y 2

#define m2(n04_param_hides_param________31x,\
            n04_param_hides_param________31y) 1
```

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

```
#define m1(n01_param_of_other_macro) (n01_param_hides_macro + 1)
#define m2(n01_param_of_other_macro) (n01_param_hides_macro + 1)
```

MISRAC2012-Rule-5.4_c99

Synopsis

Macro names were found 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:

```
/*         0        1         2         3         4         5
6     */
/*
123456789012345678901234567890123456789012345678901234567890123*
*/
#define
n01_macro_hides_macro________________________________________63x
1
#define
n02_param_hides_macro________________________________________63x
1
#define
n03_macro_hides_param________________________________________63x
1

#define
n01_macro_hides_macro________________________________________63y
2
#define
m1(n02_param_hides_macro________________________________________6
3y) \

(n01_param_hides_macro________________________________________63y
+ 1)
#define
n03_macro_hides_param________________________________________63y
2

#define
m2(n04_param_hides_param________________________________________6
3x, \

n04_param_hides_param________________________________________63y)
1
```

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

```
#define m1(n01_param_of_other_macro) (n01_param_hides_macro + 1)
#define m2(n01_param_of_other_macro) (n01_param_hides_macro + 1)
```

MISRAC2012-Rule-5.5_c89

Synopsis Non-macro identifiers were found 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:

```
/*          1234567890123456789012345678901***  */
#define     n01_var_hides_macro__________31x 1
#define     n02_function_hides_macro_____31x 1
#define     n03_param_hides_macro________31x 1
#define     n04_type_hides_macro_________31x 1
#define     n05_tag_hides_macro__________31x 1
#define     n06_label_hides_macro________31x 1

int         n01_var_hides_macro__________31y;
void        n02_function_hides_macro_____31y(int
n03_param_hides_macro________31y){}
typedef int n04_type_hides_macro_________31y;
struct      n05_tag_hides_macro__________31y {
  int x;
};
void f1() {
n06_label_hides_macro________31y:
}
```

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

```
#define   n01_expanded_macro 1

void foo() {
  int x = n01_expanded_macro;
}
```

MISRAC2012-Rule-5.5_c99

Synopsis Non-macro identifiers were found 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:

```
/*         0         1         2         3         4         5
6    */
/*
123456789012345678901234567890123456789012345678901234567890123*
*/
#define
n01_var_hides_macro__________________________________________63x
1
#define
n02_function_hides_macro_____________________________________63x
1
#define
n03_param_hides_macro________________________________________63x
1
#define
n04_type_hides_macro_________________________________________63x
1
#define
n05_tag_hides_macro__________________________________________63x
1
#define
n06_label_hides_macro________________________________________63x
1

int
n01_var_hides_macro__________________________________________63y;
void
n02_function_hides_macro_____________________________________63y(
        int
n03_param_hides_macro________________________________________63y)
{}
typedef int
n04_type_hides_macro_________________________________________63y;
struct
n05_tag_hides_macro__________________________________________63y
{
  int x;
};
void f1() {
n06_label_hides_macro________________________________________63y:
}
```

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

```
#define   n01_expanded_macro 1

void foo() {
  int x = n01_expanded_macro;
}
```

MISRAC2012-Rule-5.6

Synopsis

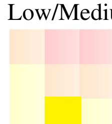
A typedef with this name has already been declared.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) A typedef name shall be a unique identifier. This check is identical to MISRAC2004-5.3, MISRAC++2008-2-10-3

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;

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 clashes with a previous declaration.

Enabled by default
Yes

Severity/Certainty
Low/Medium

Full description
(Required) A tag name shall be a unique identifier. This check is identical to MISRAC2004-5.4, MISRAC++2008-2-10-4

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
One or more external identifier names were found that are not 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 */
}

/* file2.c
#include <stdint.h>
static void foo ( void ) // Non-compliant - "foo" is not unique
                         // (it is already defined with external
                         // linkage in file1.c)
{
} */
```

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 */
}

/* file2.c
#include <stdint.h>
static void foo2 ( void )
{
  int32_t index; // Compliant - "index" has no linkage
} */
```

MISRAC2012-Rule-5.9

Synopsis	An internal identifier name was found that is not unique.
Enabled by default	No
Severity/Certainty	Low/Medium

Full description

(Advisory) Identifiers that define objects or functions with internal linkage should be unique.

Coding standards

MISRA C:2012 Rule-5.9

(Advisory) Identifiers that define objects or functions with internal linkage should be unique

Code examples

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

```
static int x;

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

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

```
static int x;
void example(void) {
  int y;
}
```

MISRAC2012-Rule-6.1

Synopsis

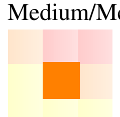
Bitfields of plain int type were found.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Bitfields shall only be declared with an appropriate type. This check is identical to MISRAC2004-6.4

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;
};
```

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

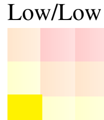
```
struct good {
  unsigned int x:3;
};
```

MISRAC2012-Rule-6.2

Synopsis
Signed single-bit bitfields (excluding anonymous fields) were found.

Enabled by default
Yes

Severity/Certainty
Low/Low



Full description
(Required) Single-bit named bitfields shall not be of a signed type. This check is identical to STRUCT-signed-bit, MISRAC2004-6.5, MISRAC++2008-9-6-4

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
Octal integer constants are used.

Enabled by default
Yes

Severity/Certainty
Low/Medium

Full description
(Required) Octal constants shall not be used. This check is identical to MISRAC2004-7.1, MISRAC++2008-2-13-2

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
There are unsigned integer constants without a `U` suffix.

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. This check is identical to MISRAC2004-10.6, MISRAC++2008-2-13-3

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

The lower case character `l` was found used as a suffix on numeric constants.

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:

```
void func()
{
  const int b = 0l;
}
```

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

```
void func()
{
  const int a = 0L;
}
```

MISRAC2012-Rule-7.4_a

Synopsis

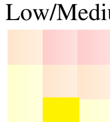
A string literal was found assigned to a variable that is not declared as constant.

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

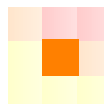
Part of a string literal was found that is modified via the array subscript operator [].

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

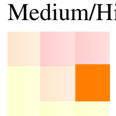
An object or function of the type `int` is declared or defined, but its type is not explicitly stated.

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description
(Required) Types shall be explicitly specified. This check is identical to DECL-implicit-int, MISRAC2004-8.2

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:

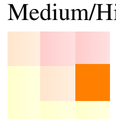
```
void func(void)
{
    int x;
}
```

MISRAC2012-Rule-8.2_a

Synopsis
There are 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. This check is identical to FUNC-unprototyped-all, MISRAC2004-16.5

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 were found with unnamed parameters.

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

(Required) Function types shall be in prototype form with named parameters. This check is identical to MISRAC2004-16.3

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-8.3_b

Synopsis

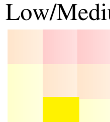
Multiple declarations of an object or function were found that use different names and type qualifiers.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) All declarations of an object or function shall use the same names and type qualifiers.

Coding standards

MISRA C:2004 8.3

> (Required) For each function parameter, the type given in the declaration and definition shall be identical and the return types shall also be identical.

MISRA C:2012 Rule-8.3

> (Required) All declarations of an object or function shall use the same names and type qualifiers

Code examples

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

```
/* file2.c:
const int x;
volatile int v;
*/
extern const unsigned int x;
```

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

```
/* file2.c
extern const int x;
 */
const int x;

int foo(const int param) {
  return (param + 1);
}
```

MISRAC2012-Rule-8.4

Synopsis

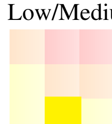
An extern definition is missing a compatible declaration.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) A compatible declaration shall be visible when an object or function with external linkage is defined.

Coding standards

MISRA C:2012 Rule-8.4

> (Required) A compatible declaration shall be visible when an object or function with external linkage is defined

Code examples

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

```
extern int x = 1;

char c = 'c';

void foo (void) {}
```

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

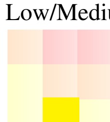
```
extern int x;

int x = 0;

extern void foo (void);

void foo (void) {}

static void bar1 (void){}

static void bar2 (void);

void bar2 (void) {}
```

MISRAC2012-Rule-8.5_a

Synopsis Multiple declarations of the same external object or function were found.

Enabled by default No

Severity/Certainty Low/Medium



Full description (Required) An external object or function shall be declared once in one and only one file. This check is identical to MISRAC2004-8.8_a

Coding standards MISRA C:2004 8.8

(Required) An external object or function shall be declared in one and only one file.

MISRA C:2012 Rule-8.5

(Required) An external object or function shall be declared once in one and only one file

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

```
#include"example.fail.h"

int x;
extern int x;
extern int x;

extern void fun(void);

void fun(void) {
}
```

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

```
#include"example.pass.h"

int x = 1;

void fun(void) {
}
```

MISRAC2012-Rule-8.5_b

Synopsis

Multiple declarations of the same external object or function were found.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) An external object or function shall be declared once in one and only one file. This check is identical to MISRAC2004-8.8_b

Coding standards

MISRA C:2004 8.8

> (Required) An external object or function shall be declared in one and only one file.

MISRA C:2012 Rule-8.5

> (Required) An external object or function shall be declared once in one and only one file

Code examples

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

```
/* file2.c
  extern int foo(int m);
 */
extern int foo(int m);
```

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

```
/* file1.c
   extern int foo( int m );
*/

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

MISRAC2012-Rule-8.6

Synopsis

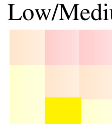
Multiple definitions or no definition were found for an external object or function.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) An identifier with external linkage shall have exactly one external definition. Note: This check is not part of C-STAT but detected by the IAR linker. This check is identical to MISRAC2004-8.9

Coding standards

MISRA C:2004 8.8

> (Required) An external object or function shall be declared in one and only one file.

MISRA C:2012 Rule-8.6

> (Required) An identifier with external linkage shall have exactly one external definition

Code examples

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

```
int foo(int v);
int example() {
  return foo(3);
}
```

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

```
extern int x;

extern void example(void);

int x = 1;

void example(void) {
}
```

MISRAC2012-Rule-8.7

Synopsis

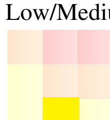
An externally linked object or function was found referenced in only one translation unit.

Enabled by default

No

Severity/Certainty

Low/Medium



Full description

(Advisory) Functions and objects should not be defined with external linkage if they are referenced in only one translation unit. This check is identical to MISRAC2004-8.10

Coding standards

MISRA C:2004 8.10

(Required) All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.

MISRA C:2012 Rule-8.7

(Advisory) Functions and objects should not be defined with external linkage if they are referenced in only one translation unit

Code examples

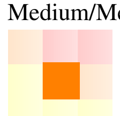
The following code example fails the check and will give a warning:

```
/* file1.c
static void example (void) {
  // dummy function
}
*/

/* extern linkage */
extern int x;

/* static linkage */
static void foo(void) {
  /* only referenced here */
  x = 1;
}
```

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

```
/* static linkage */
static int x;

/* static linkage */
static void foo(void) {
  /* no linkage */
  int y = (x++);
  if(y < 10)
    foo();
}
```

MISRAC2012-Rule-8.9_a

Synopsis	A global object was found that is only referenced from a single function.
Enabled by default	No
Severity/Certainty	Medium/Medium 
Full description	(Advisory) An object should be defined at block scope if its identifier only appears in a single function.
Coding standards	MISRA C:2012 Rule-8.9

(Advisory) An object should be defined at block scope if its identifier only appears in a single function

Code examples

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

```
static int i = 10; // this object is only used inside the example
function

int example(void) {
  return i;
}

void main() {
  printf("example() = %d\n", example());
}
```

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

```
int example(void) {
  int i = 10; // this object is only used inside the example
function
  return i;
}

void main() {
  printf("example() = %d\n", example());
}
```

MISRAC2012-Rule-8.9_b

Synopsis

A global object was found that is only referenced from a single function.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

(Advisory) An object should be defined at block scope if its identifier only appears in a single function.

Coding standards

MISRA C:2012 Rule-8.9

(Advisory) An object should be defined at block scope if its identifier only appears in a single function

Code examples

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

```
static int i = 10; // this object is only used inside the example
function

int example(void) {
  return i;
}

void main() {
  printf("example() = %d\n", example());
}
```

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

```
int example(void) {
  int i = 10; // this object is only used inside the example
function
  return i;
}

void main() {
  printf("example() = %d\n", example());
}
```

MISRAC2012-Rule-8.10

Synopsis

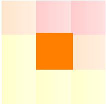
Inline functions were found that are not declared as static.

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

One or more external arrays are declared without their size being 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. This check is identical to MISRAC2004-8.12, MISRAC++2008-3-1-3

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-8.12

Synopsis

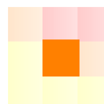
A duplicated implicit enumeration constant was found.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The value of an implicitly-specified enumeration constant shall be unique.

Coding standards

MISRA C:2012 Rule-8.12

> (Required) Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique

Code examples

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

```
/* skink equals to geko */
enum lizards { goanna = 1, parentie = 2, skink, geko = 3 };
```

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

```
enum lizards { goanna, parentie, skink = 3, geko = 3 };
```

MISRAC2012-Rule-8.13

Synopsis

A pointer was found that is not const-qualified.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

(Advisory) A pointer should be const-qualified whenever possible.

Coding standards

MISRA C:2012 Rule-8.13

> (Advisory) A pointer should point to a const-qualified type whenever possible

Code examples

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

```
int example(int *p) {
  return *p;
}
```

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

```
int example(const int *p) {
  return *p;
}
```

MISRAC2012-Rule-8.14

Synopsis

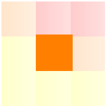
The `restrict` type qualifier was found used in function parameters.

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

A possible dereference of an uninitialized or `NULL` pointer was found.

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 check is identical to PTR-uninit-pos

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

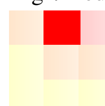
Read accesses from local buffers were found that are not preceded by writes.

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 check is identical to SPC-uninit-arr-all, MISRAC2004-1.2_a

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 a[20];
  int b = a[1];
}
```

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

```
extern void f(int*);
void example() {
  int a[20];
  f(a);
  int b = a[1];
}
```

MISRAC2012-Rule-9.1_c

Synopsis

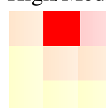
On all execution paths, there is a struct that has one or more fields read before they are initialized.

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. This check is identical to SPC-uninit-struct, MISRAC2004-1.2_b

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

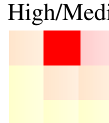
A field of a local struct is read before it is initialized.

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. This check is identical to SPC-uninit-struct-field

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

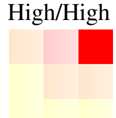
On all execution paths, there is a variable that is read before it is assigned a value.

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. This check is identical to SPC-uninit-var-all, MISRAC2004-9.1_a, MISRAC++2008-8-5-1_a, MISRAC2012-Rule-1.3_j

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

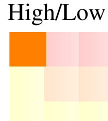
A variable was found that might read before it is assigned a value.

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. This check is identical to SPC-uninit-var-some, MISRAC2004-9.1_b, MISRAC++2008-8-5-1_b, MISRAC2012-Rule-1.3_k

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-9.2

Synopsis

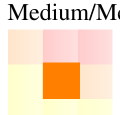
An initializer for an aggregate or union was found that is not enclosed in braces.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The initializer for an aggregate or union shall be enclosed in braces.

Coding standards

MISRA C:2012 Rule-9.2

(Required) The initializer for an aggregate or union shall be enclosed in braces

Code examples

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

```
void example(void) {
  int a[2][2] = { 1, 2, 3, 4 };
}
```

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

```
void example(void) {
  int a[2][2] = { { 1, 2 }, { 3, 4 } };
}
```

MISRAC2012-Rule-9.3

Synopsis

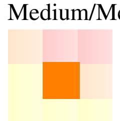
Arrays were found that are partially initialized.

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][3] = { { 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-9.4

Synopsis

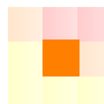
An object field was found that is initialized more than once. The last initialization will overwrite previous value(s).

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) An element of an object shall not be initialized more than once.

Coding standards

MISRA C:2012 Rule-9.4

(Required) An element of an object shall not be initialized more than once

Code examples

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

```
struct example {
  int x;
  int y;
};

struct example object = { .x = 100, .x = 200 };
// object = { .x = 100, .y = 0 };
```

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

```
struct example {
  int x;
  int y;
};

struct example object = { .x = 100, .y = 200 };
// object = { .x = 100, .y = 200 };
```

MISRAC2012-Rule-9.5_a

Synopsis

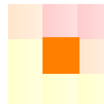
Arrays, initialized with designated initializers but with no fixed length, were found.

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

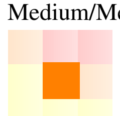
A flexible array member was found that is initialized with a designated initializer.

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

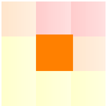
An operand was found that is not of essentially Boolean type, despite being interpreted as a Boolean value.

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 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:

```
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;

}
```

MISRAC2012-Rule-10.1_R3

Synopsis

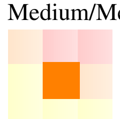
An operand was found that is of essentially Boolean type, despite being interpreted as a numeric value.

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;
}
```

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

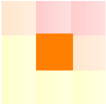
```
void func()
{
  bool x;
  bool y;
  y = x && y;
}
typedef charboolean_t;/* Compliant: Boolean-by-enforcement */

void example(void)
{
    boolean_t d;
    boolean_t c = 1;
    boolean_t b = 0;
    boolean_t a = 1;

    d = ( c && a ) && b;

}
```

MISRAC2012-Rule-10.1_R4

Synopsis | An operand was found that is of essentially character type, despite being 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';
  char b = 'b';
  char c;
  c = 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

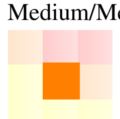
An operand that is of essentially enum type is used in an arithmetic operation, because an enum object uses an implementation-defined integer type.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Operands shall not be of an inappropriate essential type.

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;
}
```

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;
}
```

MISRAC2012-Rule-10.1_R6

Synopsis

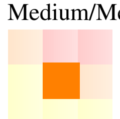
Shift and bitwise operations were found performed on operands of essentially signed type.

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 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

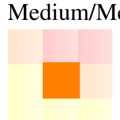
The right-hand operand of a shift operator is not of essentially unsigned type.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Operands shall not be of an inappropriate essential type. The right-hand operand of a shift operator is not of essentially unsigned type, meaning that undefined behavior might result from a negative shift.

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

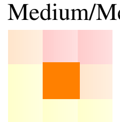
An operand of essentially unsigned typed is used as the operand to the unary minus operator.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Operands shall not be of an inappropriate essential type. An operand of essentially unsigned typed is used as the operand to the unary minus operator. This is problematic because the signedness of the result is determined by the implementation-defined size of int. This check is identical to MISRAC++2008-5-3-2_a, MISRAC2004-12.9

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:

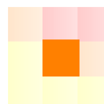
```
void example(void) {
   int neg_one = -1;
}
```

MISRAC2012-Rule-10.2

Synopsis
Expressions of essentially character type were found used inappropriately 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:

```
void example(void) {
  char a = '9';
  char c = a + '0';
}
```

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

```
void example(void) {
  int a = 9;
  char dig = a + '0';
}
```

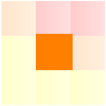
MISRAC2012-Rule-10.3

Synopsis
The value of an expression was found 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 This check is identical to

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_a

Synopsis

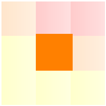
Operands of an operator in which the usual arithmetic conversions are performed were found, that do not have the same essential type category.

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. This check is identical to

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-10.4_b

Synopsis

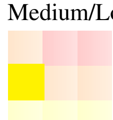
The second and third operands of the ternary operator do not have the same essential type category.

Enabled by default

Yes

Severity/Certainty

Medium/Low



Full description

(Required) The second and third operands of the ternary operator shall have the same essential type category. This check is identical to

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) {
  int x;
  float y;
  int z = (x > 0)?x:y;
}
```

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

```
void example(void) {
  int x;
  float y;
  int z = (x > 0)?x:(x+1);
}
```

MISRAC2012-Rule-10.5

Synopsis

A value of an expression was found that is cast to an inappropriate essential type.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

(Advisory) The value of an expression should not be cast to an inappropriate essential type.

Coding standards

MISRA C:2012 Rule-10.5

(Advisory) The value of an expression should not be cast to an inappropriate essential type

Code examples

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

```
#include <stdbool.h>

void example(void) {
  bool a = false;
  int s32a = (int) a;
}
```

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

```
#include <stdbool.h>

void example(void) {
  bool a = false;
  bool b = (bool) a;
}
```

MISRAC2012-Rule-10.6

Synopsis

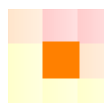
The value of a composite expression is assigned to an object with wider essential type.

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

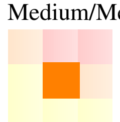
An operator in which the usual arithmetic conversions are performed was found, where a composite expression is used as one of the operands, but the other operand is of wider essential type.

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:

```
void example(long l, short s) {
  l * ( s + s ); /* Implicit conversion of (ua + ua) */
}
```

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

```
void example(long l, short s) {
  l * s + s; /* No composite conversion */
}
```

MISRAC2012-Rule-10.8

Synopsis

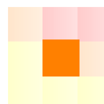
A composite expression was found whose value is cast to a different essential type category or a wider essential type.

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:

```
void example(void) {
  int s16a = 3;
  int s16b = 3;

  // arithmetic makes it a complex expression
  long long x = (long long)(s16a + s16b);
}
```

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

```
void example(void) {
   int array[10];

   // A non complex expression is considered safe
   long x = (long)(array[5]);
}
```

MISRAC2012-Rule-11.1

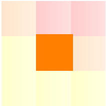
Synopsis

Conversion between a pointer to a function and another type were found.

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:

```
typedef void ( *fp16 ) ( int n );
typedef fp16 ( *pfp16 ) ( void );

void example(void) {
    pfp16 pfp1;
    ( void ) ( *pfp1 ( ) );   /* Compliant - exception 2 - cast
function
                               * pointer into void */
}
```

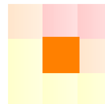
MISRAC2012-Rule-11.2

Synopsis A conversion from or to an incomplete type pointer was found.

Enabled by default Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Conversions shall not be performed between a pointer to an incomplete type and any other types.

Coding standards

MISRA C:2012 Rule-11.2

> (Required) Conversions shall not be performed between a pointer to an incomplete type and any other type

Code examples

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

```
struct a;
struct b;
void example(void) {
  struct a * p1;
  struct b * p2;
  unsigned int x;
  p1 = (struct a *) 0x12345678;
  x = (unsigned int) p2;
  p1 = (struct a *) p2;
}
```

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

```
#include <stdlib.h>

struct a;
extern struct a *f (void);

void example(void) {
  struct a * p;
  unsigned int x;
  /* exception 1: NULL -> incomplete type ptr */
  p = (struct a *) NULL;
  /* exception 2: incomplete type ptr -> void */
  (void) f();
}
```

MISRAC2012-Rule-11.3

Synopsis

A pointer to object type is cast to a pointer to a 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 A pointer to object type is cast to a pointer to a different object type. Conversions of this type might be invalid if the new pointer type requires 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 between a pointer type and an integral type was found.

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 This check is identical to MISRAC2004-11.3, MISRAC++2008-5-2-9

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-11.5

Synopsis

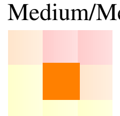
A conversion from a pointer to void into a pointer to object was found.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

(Advisory) A conversion should not be performed from pointer to void into pointer to object.

Coding standards

MISRA C:2012 Rule-11.5

(Advisory) A conversion should not be performed from pointer to void into pointer to object

Code examples

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

```
void example(void) {
  int * x;
  void * y;
  x = y;
}
```

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

```
void example(void) {}
```

MISRAC2012-Rule-11.6

Synopsis

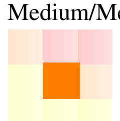
A conversion between a pointer to void and an arithmetic type was found.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) A cast shall not be performed between pointer to void and an arithmetic type.

Coding standards

MISRA C:2012 Rule-11.6

(Required) A cast shall not be performed between pointer to void and an arithmetic type

Code examples

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

```
void example(void) {
  void * x;
  unsigned int y;
  x = (void *) 0x12345678;
  y = (unsigned int) x;
}
```

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

```
void example(void) {
  void * x;
  void * y;
  x = (void *) y;
}
```

MISRAC2012-Rule-11.7

Synopsis

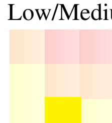
A cast between a pointer to object and a non-integer arithmetic type was found.

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:

```
void example(void) {
  int *p;
  float f;
  f = (float)p;    /* Non-compliant */
}
```

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

```
void example(void) {
  int *p;
  short f;
  f = (short)p;
}
```

MISRAC2012-Rule-11.8

Synopsis

A cast that removes a const or volatile qualification was found.

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 A cast that removes a const or volatile qualification was found. This violates the principle of type qualification. Changes to the qualification of the pointer during the cast were not checked for. This check is identical to MISRAC2004-11.5, MISRAC++2008-5-2-5

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

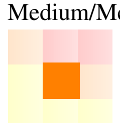
An integer constant was found where the NULL macro should be.

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

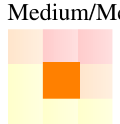
Implicit operator precedence was detected, without parenthesis to make it explicit.

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

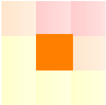
Out of range shifts were found

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 This check is identical to ATH-shift-bounds, MISRAC2004-12.8, MISRAC++2008-5-8-1

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 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;
}
```

MISRAC2012-Rule-12.3

Synopsis

There are uses of the comma operator.

Enabled by default

No

Severity/Certainty

Low/High

Full description

(Advisory) The comma operator should not be used This check is identical to MISRAC2004-12.10, MISRAC++2008-5-18-1

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-13.1

Synopsis

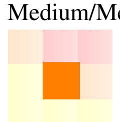
The initalization list of an array contains side effects.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Initializer lists shall not contain persistent side effects This check is identical to

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

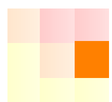
Expressions that depend on order of evaluation were found.

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 This check is identical to SPC-order, MISRAC2004-12.2_a, MISRAC++2008-5-0-1_a, MISRAC2012-Rule-1.3_i

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

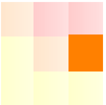
There are multiple read accesses with volatile-qualified type within one and the same sequence point.

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 This check is identical to SPC-volatile-reads, MISRAC2004-12.2_b, MISRAC++2008-5-0-1_b

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 the same under all permitted evaluation orders

Code examples

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

```
void example(void) {
  int x;
  volatile int 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

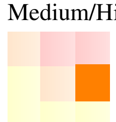
There are multiple write accesses with volatile-qualified type within one and the same sequence point.

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 This check is identical to SPC-volatile-writes, MISRAC2004-12.2_c, MISRAC++2008-5-0-1_c

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:

```
void example(void) {
  int x;
  volatile int 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

The increment (++) and decrement (--) operators are being used 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 This check is identical to MISRAC2004-12.13, MISRAC++2008-5-2-10

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 might 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 check is identical to EXP-cond-assign

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

Assignments were found 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 This check is identical to MISRAC++2008-6-2-1

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

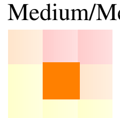
There are right-hand operands of && or || operators that contain side effects.

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 This check is identical to MISRAC2004-12.4, MISRAC++2008-5-14-1

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

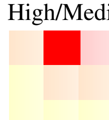
The operand of the sizeof operator contains an expression that has potential side effects.

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

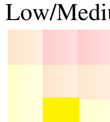
Floating-point values were found in the controlling expression of a for statement.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) A loop counter shall not have essentially floating type This check is identical to MISRAC2004-13.4, MISRAC++2008-6-5-1_a

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

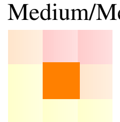
A variable of essentially float type that is used in the loop condition, is then modified in the loop body.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) A loop counter shall not have essentially floating type This check is identical to

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 malformed `for` loop was found.

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

(Required) A `for` loop shall be well-formed.

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

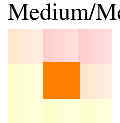
The condition in an if, for, while, do-while, or ternary operator will always be true.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Controlling expressions shall not be invariant This check is identical to RED-cond-always, MISRAC++2008-0-1-2_a

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

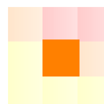
The condition in if, for, while, do-while, or ternary operator will never be true.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Controlling expressions shall not be invariant This check is identical to RED-cond-never, MISRAC++2008-0-1-2_b

Coding standards

CERT EXP17-C

Do not perform bitwise operations in conditional expressions

CWE 570

Expression is Always False

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 were found 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 This check is identical to MISRAC2004-13.2_a, MISRAC++2008-5-0-13_a

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:

```
int 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
}
```

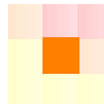
MISRAC2012-Rule-14.4_b

Synopsis Non-Boolean termination conditions were found 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 This check is identical to MISRAC2004-13.2_b, MISRAC++2008-5-0-13_b

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 were found 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 This check is identical to MISRAC2004-13.2_c, MISRAC++2008-5-0-13_c

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 were found 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 This check is identical to MISRAC2004-13.2_d, MISRAC++2008-5-0-13_d

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 the goto statement were found.
Enabled by default	No

Severity/Certainty Low/Medium

Full description (Advisory) The goto statement should not be used This check is identical to MISRAC2004-14.4

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 A goto statement is declared after the destination 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 This check is identical to MISRAC++2008-6-6-2

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 destination of a goto statement 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 This check is identical to MISRAC++2008-6-6-1

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

One or more iteration statements are terminated by more than one break or goto statements.

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 This check is identical to MISRAC++2008-6-6-4

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:

```
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;
      }
   }
}
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
    }
  }
}
```

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

```
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
    {
    }
  }
}
void example(void)
{
   int i = 0;
   for (i = 0; i < 10 && i != 9; i++) {
      if (i == 9) {
         break;
      }
   }
}
```

MISRAC2012-Rule-15.5

Synopsis: One or more functions have multiple exit points or an exit point that is not 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 This check is identical to MISRAC2004-14.7, MISRAC++2008-6-6-5

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

There are 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 This check is identical to MISRAC2004-14.8_a, MISRAC++2008-6-3-1_a

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

There are 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 This check is identical to MISRAC2004-14.8_b, MISRAC++2008-6-3-1_b

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

There are missing braces in `if`, `else`, or `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 This check is identical to MISRAC2004-14.9, MISRAC++2008-6-4-1

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

There are 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 This check is identical to MISRAC2004-14.8_c, MISRAC++2008-6-3-1_c

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

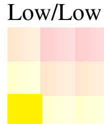
There are missing braces in `while` statements.

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 This check is identical to MISRAC2004-14.8_d, MISRAC++2008-6-3-1_d

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 were detected.

Enabled by default Yes

Severity/Certainty Low/High

Full description (Required) All if ... else if constructs shall be terminated with an else statement This check is identical to MISRAC2004-14.10, MISRAC++2008-6-4-2

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 Detected 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 This check is identical to MISRAC2004-15.0, MISRAC++2008-6-4-3

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 were found 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 This check is identical to MISRAC2004-15.1, MISRAC++2008-6-4-4

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

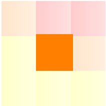
Non-empty switch cases were found that are not terminated by a break.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) An unconditional break statement shall terminate every switch-clause This check is identical to MISRAC2004-15.2, MISRAC++2008-6-4-5

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) {

  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;
  }

}
```

MISRAC2012-Rule-16.4

Synopsis

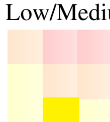
Switch statements without a default clause were found.

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

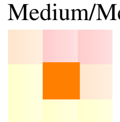
A switch was found whose default label is neither the first nor the last label of the switch.

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 without case clauses were found.

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 was found that represents 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 This check is identical to MISRAC2004-15.4, MISRAC++2008-6-4-7

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

Inclusion of the stdarg header file was detected.

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

There are 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 This check is identical to MISRAC2004-16.2_a, MISRAC++2008-7-5-4_a

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

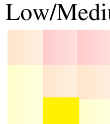
There are functions that call themselves indirectly.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) Functions shall not call themselves, either directly or indirectly This check is identical to MISRAC2004-16.2_b, MISRAC++2008-7-5-4_b

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

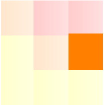
Functions are used without prototyping.

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

(Mandatory) A function shall not be declared implicitly This check is identical to FUNC-implicit-decl, MISRAC2004-8.1

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

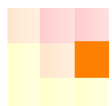
For some execution paths, no return statement is executed in a function with a non-`void` return type.

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 This check is identical to SPC-return, MISRAC2004-16.8, MISRAC++2008-8-4-3

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:

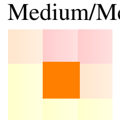
```
#include <stdio.h>

int example(void) {
  int x;

  scanf("%d",&x);

  if (x > 10) {
    return 10;
  }

  return 0;
}
```

MISRAC2012-Rule-17.5

Synopsis
A function call is made with the wrong array type argument.

Enabled by default
No

Severity/Certainty
Medium/Medium



Full description
(Advisory) The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements.

Coding standards
MISRA C:2012 Rule-17.5

> (Advisory) The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements

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

```
void callee(int array[10]);

void caller(void) {
  int arr4[4];
  callee(arr4);
}
```

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

```
void callee(int array[10]);

void caller(void) {
   int arr4[10];
   callee(arr4);
}
```

MISRAC2012-Rule-17.6

Synopsis

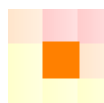
There are array parameters with the `static` keyword between the [].

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

There are unused function return values (other than 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
This check is identical to RED-unused-return-val, MISRAC++2008-0-1-7

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-17.8

Synopsis

A function parameter was found that is modified.

Enabled by default

No

Severity/Certainty

Low/High

Full description

(Advisory) A function parameter should not be modified.

Coding standards

MISRA C:2012 Rule-17.8

(Advisory) A function parameter should not be modified

Code examples

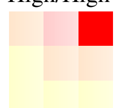
The following code example fails the check and will give a warning:

```
void example(int p) {
  int a = p + 5;
  p = a;
}
```

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

```
void example(int *p) {
  *p = 5;
}
```

MISRAC2012-Rule-18.1_a

Synopsis	An 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 check is identical to ARR-inv-index, MISRAC++2008-5-0-16_c

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 example(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

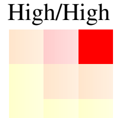
An array access might be out of bounds, depending on which path is executed.

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 check is identical to ARR-inv-index-pos, MISRAC++2008-5-0-16_d

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:

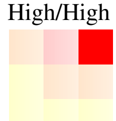
```
int cond;

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 This check is identical to ARR-inv-index-ptr, MISRAC++2008-5-0-16_e

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

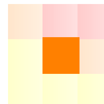
A pointer to an array is potentially used outside the array bounds.

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 This check is identical to ARR-inv-index-ptr-pos, MISRAC++2008-5-0-16_f

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-18.2

Synopsis

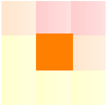
A subtraction was found between pointers that address elements of different arrays.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Subtraction between pointers shall only be applied to pointers that address elements of the same array. Note: This rule will only accept arrays of the form '<type> <name>[<size>]'. This check is identical to MISRAC2004-17.2

Coding standards

MISRA C:2004 17.2

> (Required) Pointer subtraction shall only be applied to pointers that address elements of the same array.

MISRA C:2012 Rule-18.2

> (Required) Subtraction between pointers shall only be applied to pointers that address elements of the same array

Code examples

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

```
#include <stddef.h>

void example(void) {
  int a[20];
  int b[20];
  int *p1 = &a[5];
  int *p2 = &b[2];
  ptrdiff_t diff;
  diff = p2 - p1;
}
```

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

```
#include <stddef.h>

void example(void) {
  int arr[10];
  int *p1 = &arr[5];
  int *p2 = &arr[5];
  ptrdiff_t diff;
  diff = p2 - p1;
}
```

MISRAC2012-Rule-18.3

Synopsis

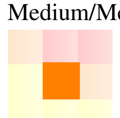
A relational operator was found applied to an object of pointer type that does not point into the same object.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object. This check is identical to MISRAC2004-17.3

Coding standards

MISRA C:2004 17.3

(Required) >, >=, <, <= shall not be applied to pointer types except where they point to the same array.

MISRA C:2012 Rule-18.3

> (Required) The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object

Code examples

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

```
void example(void) {
  int a[10];
  int b[10];
  int *p1 = &a[1];
  if (p1 < b) {

  }
}
```

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];
  int *p1 = &a[1];
  if (p1 < a) {

  }
}
```

MISRAC2012-Rule-18.4

Synopsis

A +, -, +=, or -= operator was found applied to an expression of pointer type.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The +, -, += and -= operators should not be applied to an expression of pointer type.

Coding standards

MISRA C:2012 Rule-18.4

> (Advisory) The +, -, += and -= operators should not be applied to an expression of pointer type

Code examples

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

```
void example(int *ptr) {
  int a = *(ptr + 1);
}
```

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

```
void example(int *ptr) {
  int a = ptr[1];
}
```

MISRAC2012-Rule-18.5

Synopsis

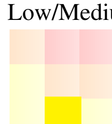
Declarations that contain more than two levels of pointer indirection have been found.

Enabled by default

No

Severity/Certainty

Low/Medium



Full description

(Advisory) Declarations should contain no more than two levels of pointer nesting This check is identical to MISRAC2004-17.5, MISRAC++2008-5-0-19

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:

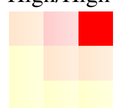
```
void example(void) {
    int **p;
}
```

MISRAC2012-Rule-18.6_a

Synopsis
Might 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 This check is identical to MEM-stack, MISRAC2004-17.6_a, MISRAC++2008-7-5-1_b

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 *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

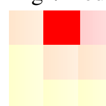
A stack address is stored in a global pointer.

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 check is identical to MEM-stack-global, MISRAC2004-17.6_b, MISRAC++2008-7-5-2_a

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

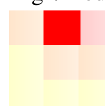
A stack address is stored in the field of a global struct.

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 check is identical to MEM-stack-global-field, MISRAC2004-17.6_c, MISRAC++2008-7-5-2_b

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

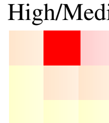
A stack address is stored outside a function via a parameter.

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 check is identical to MEM-stack-param, MISRAC2004-17.6_d, MISRAC++2008-7-5-2_c, MISRAC2012-Rule-1.3_s

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

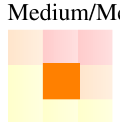
Flexible array members are declared.

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:

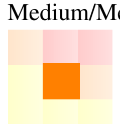
```
struct example {
  int size;
  int data[5];
} example;

void function(void) {
  struct example *e;
}
```

MISRAC2012-Rule-18.8

Synopsis There are arrays 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];
}
```

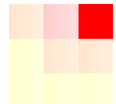
MISRAC2012-Rule-19.1

Synopsis Assignments from one field of a union to another were found.

Enabled by default Yes

Severity/Certainty

High/High



Full description

(Mandatory) An object shall not be assigned or copied to an overlapping object This check is identical to UNION-overlap-assign, MISRAC2004-18.2, MISRAC++2008-0-2-1

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:

```
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;
}
```

MISRAC2012-Rule-19.2

Synopsis

Unions were found.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

(Advisory) The union keyword should not be used This check is identical to MISRAC2004-18.4, MISRAC++2008-9-5-1

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-20.1

Synopsis

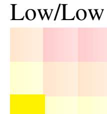
#include directives were found that are not first in the source file.

Enabled by default

No

Severity/Certainty

Low/Low



Full description

(Advisory) #include directives should only be preceded by preprocessor directives or comments. This check is identical to MISRAC2004-19.1

Coding standards

MISRA C:2004 19.1

> (Advisory) #include statements in a file should only be preceded by other preprocessor directives or comments.

MISRA C:2012 Rule-20.1

> (Advisory) #include directives should only be preceded by preprocessor directives or comments

Code examples

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

```
int x;
#include <cstdio>
void example(void) {}
```

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

```
#include <cstdio>
void example(void) {}
```

MISRAC2012-Rule-20.2

Synopsis

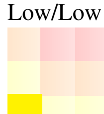
Illegal characters were found in the names of header files.

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 This check is identical to MISRAC2004-19.2

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

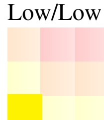
A macro was found defined with the same name as a keyword.

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 was found defined with the same name as a keyword.

Enabled by default

Yes

oror

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 Found occurrences of #undef.

Enabled by default No

Severity/Certainty Low/Low

Full description (Advisory) #undef should not be used This check is identical to MISRAC2004-19.6, MISRAC++2008-16-0-3

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:

```
#define SYM
#undef SYM
```

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

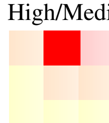
```
#define SYM
```

MISRAC2012-Rule-20.7

Synopsis
An expansion of macro parameters was found that is not enclosed in parentheses.

Enabled by default
Yes

Severity/Certainty
High/Medium



Full description
(Required) The expansion of macro parameters shall be enclosed in parentheses.

Coding standards
MISRA C:2012 Rule-20.7

> (Required) Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses

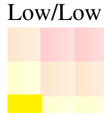
Code examples
The following code example fails the check and will give a warning:

```
void example(void) {
  int r;

#define M( x, y ) ( x / y )

  r = M ( 1 + 2, 1 - 2 );
}
```

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

```
static struct str {
  int val;
} s;

void example(void) {
  int r;
  int a[10];

  /* already enclosed in macro def*/
#define M( x, y ) ( ( x ) << ( y ) )
  r = M( 1 + 2, 3 + 4 );

  /* no need after ## or # */
#define N( x ) a [ ##x ] = (x)
  N ( 0 + 2 );

  /* no need after . or ->, member name */
#define MEMBER( S, M ) ( S ).M
   r = MEMBER ( s, val );

  /* enclosed in inner macro */
#define F( X ) G( X )
#define G( Y ) ( Y )
   r = F ( 2 );

  /* enclosed at invocation site,
     even single literal should have parentheses */
#define M( x, y ) ( x / y )
   r = M ( ( 1 ), ( 2 + 3 ) );
}
```

MISRAC2012-Rule-20.10

Synopsis	# and ## operators were found in macro definitions.
Enabled by default	No
Severity/Certainty	Low/Low 
Full description	(Advisory) The # and ## preprocessor operators should not be used This check is identical to MISRAC2004-19.13, MISRAC++2008-16-3-2

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:

```
#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

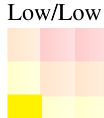
Detected a #define or #undef of a reserved identifier in the standard library.

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 This check is identical to MISRAC2004-20.1, MISRAC++2008-17-0-1

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

One or more library functions are being overridden.

Enabled by default
Yes

Severity/Certainty
Low/Medium

Full description
(Required) A reserved identifier or macro name shall not be declared This check is identical to MISRAC++2008-17-0-3, MISRAC2004-20.2

Coding standards
MISRA C:2004 20.2

> (Required) The names of Standard Library macros, objects, and functions shall not be reused.

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:

```
extern "C" void bar(void);
void foo(void) {}
```

MISRAC2012-Rule-21.3

Synopsis
Uses of malloc, calloc, realloc, or free were found.

Enabled by default
Yes

Severity/Certainty
Low/Medium

Full description
(Required) The memory allocation and deallocation functions of <stdlib.h> shall not be used This check is identical to MISRAC2004-20.4, MISRAC++2008-18-4-1

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

Found 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 This check is identical to MISRAC2004-20.7, MISRAC++2008-17-0-5

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

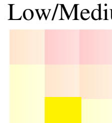
Uses of signal.h were found.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) The standard header file signal.h shall not be used This check is identical to MISRAC2004-20.8, MISRAC++2008-18-7-1

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

Uses of stdio.h were found.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The Standard Library input/output functions shall not be used This check is identical to MISRAC2004-20.9, MISRAC++2008-27-0-1

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

Uses of atof, atoi, atol, and atoll were found.

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 This check is identical to MISRAC2004-20.10, MISRAC++2008-18-0-2

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

Uses of abort, exit, getenv, and system were found.

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 This check is identical to MISRAC2004-20.11, MISRAC++2008-18-0-3

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:

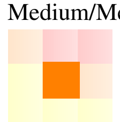
```
void example(void) {
}
```

MISRAC2012-Rule-21.9

Synopsis
Uses of the library functions bsearch and qsort in stdlib.h were found.

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	Use of the following time.h functions was found: 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 This check is identical to MISRAC2004-20.12, MISRAC++2008-18-0-4

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

Use of the standard header file tgmath.h was found.

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-21.12_a

Synopsis

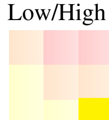
The exception-handling features of <fenv.h> are used.

Enabled by default

No

Severity/Certainty

Low/High



Full description

(Advisory) The exception-handling features of <fenv.h> should not be used.

Coding standards

MISRA C:2012 Rule-21.12

(Advisory) The exception handling features of <fenv.h> should not be used

Code examples

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

```
#include <fenv.h>
void f ()
{
  feclearexcept ( FE_DIVBYZERO );
}
```

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

```
#include <fenv.h>
void f ()
{
  /* ... */
}
```

MISRAC2012-Rule-21.12_b

Synopsis

Macros are used in <fenv.h>.

Enabled by default

No

Severity/Certainty

Low/High

Full description

(Advisory) The exception handling features of <fenv.h> should not be used.

Coding standards

MISRA C:2012 Rule-21.12

(Advisory) The exception handling features of <fenv.h> should not be used

Code examples

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

```
#include <fenv.h>

void example(void) {
  feclearexcept(FE_INEXACT);
}
```

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

```
#include <fenv.h>

void example(void) {
  /* including the header but not used its features */
}
```

MISRAC2012-Rule-22.1_a

Synopsis

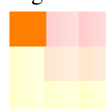
A memory leak due to incorrect deallocation was detected.

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 This check is identical to MEM-leak, SEC-BUFFER-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>

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;
}
```

MISRAC2012-Rule-22.1_b

Synopsis

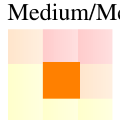
A file pointer is never closed.

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 This check is identical to RESOURCE-file-no-close-all, SEC-FILEOP-open-no-close, MISRAC2012-Dir-4.13_c

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);
}
```

MISRAC2012-Rule-22.2_a

Synopsis

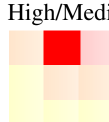
A memory location is freed more than once.

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 This check is identical to MEM-double-free

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

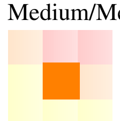
Freeing a memory location more than once on some paths but not others.

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 This check is identical to MEM-double-free-some

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

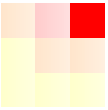
A stack address might be freed.

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 This check is identical to MEM-free-variable

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-22.3

Synopsis

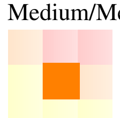
A file was found that is open for read and write access at the same time on different streams.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The same file shall not be open for read and write access at the same time on different streams.

Coding standards

MISRA C:2012 Rule-22.3

(Required) The same file shall not be open for read and write access at the same time on different streams

Code examples

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

```
#include <stdio.h>

void example(void) {
  FILE *f1 = fopen("foo", "r");
  FILE *f2;
  f2 = fopen("foo", "w");

}
```

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

```
#include <stdio.h>

void example(void) {
  FILE *f1 = fopen("foo", "r");
  FILE *f2;
  fclose(f1);
  f2 = fopen("foo", "r");

}
```

MISRAC2012-Rule-22.4

Synopsis

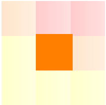
A file opened as read-only is written to.

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 This check is identical to RESOURCE-write-ronly-file

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 is dereferenced.

Enabled by default
Yes

Severity/Certainty
Low/Medium

Full description
(Mandatory) A pointer to a FILE object shall not be dereferenced This check is identical to RESOURCE-deref-file

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 *f1;
   FILE *f2;
   *f2 = *f1;
}
```

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

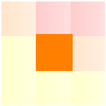
```
#include <stdio.h>

void example(void) {
  FILE *f1;
  FILE *f2;

  f1 = f2;
}
```

MISRAC2012-Rule-22.5_b

Synopsis
A file pointer was found that is implicitly dereferenced 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 This check is identical to RESOURCE-implicit-deref-file

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

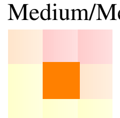
A file pointer was found that is used after it has been closed.

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
A part of the application is never executed.

Enabled by default
Yes

Severity/Certainty
Low/Medium

Full description
(Required) A project shall not contain unreachable code. This check is identical to RED-dead, MISRAC2004-14.1, MISRAC++2008-0-1-9, MISRAC2012-Rule-2.1_b

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

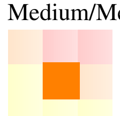
The condition in if, for, while, do-while statement sequences and the ternary operator is always met.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) A project shall not contain infeasible paths. This check is identical to RED-cond-always, MISRAC2012-Rule-14.3_a

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

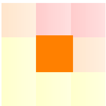
The condition in if, for, while, do-while statement sequences and the ternary operator will never be met.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) A project shall not contain infeasible paths. This check is identical to RED-cond-never, MISRAC2012-Rule-14.3_b

Coding standards

CERT EXP17-C

Do not perform bitwise operations in conditional expressions

CWE 570

Expression is Always False

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

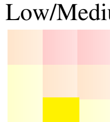
A case statement within a switch statement is unreachable.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) A project shall not contain infeasible paths. This check is identical to RED-case-reach, MISRAC2012-Rule-2.1_a

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

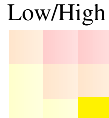
The following code example fails the check and will give a warning:

```
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 never read or written during execution.
Enabled by default	Yes
Severity/Certainty	Low/High 
Full description	(Required) A project shall not contain unused variables. This check is identical to RED-unused-var-all
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_a

Synopsis

A variable is only used once.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) A project shall not contain non-volatile POD variables having only one use.

Coding standards

CWE 563

Unused Variable

MISRA C++ 2008 0-1-4

(Required) A project shall not contain non-volatile POD variables having only one use.

Code examples

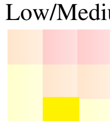
The following code example fails the check and will give a warning:

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

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

```
int example(void) {
  int x;

  x = 20;

  return x;
}
```

MISRAC++2008-0-1-4_b

Synopsis	A global variable is only used once.
Enabled by default	Yes
Severity/Certainty	Low/Medium 
Full description	(Required) A project shall not contain non-volatile POD variables having only one use.
Coding standards	CWE 563 Unused Variable 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 x = 1;
int example(void) {
  return 0;
}
```

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

```
int example(void) {
  int x;

  x = 20;

  return x;
}
```

MISRAC++2008-0-1-6

Synopsis

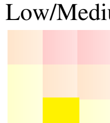
A variable is assigned a value that is never used.

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. This check is identical to RED-unused-val, MISRAC2012-Rule-2.2_c

Coding standards

CWE 563

Unused Variable

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;
}
```

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

```
int example(void) {
  int x;

  x = 20;

  return x;
}
```

MISRAC++2008-0-1-7

Synopsis

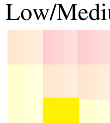
There are unused function return values (excluding overloaded operators)

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. This check is identical to RED-unused-return-val, MISRAC2012-Rule-17.7

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

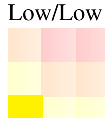
There are functions with no effect. A function with no return type and no side effects effectively does nothing.

Enabled by default

No

Severity/Certainty

Low/Low



Full description

(Required) All functions with void return type shall have external side effect(s). This check is identical to RED-func-no-effect

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

A part of the application is never executed.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) There shall be no dead code. This check is identical to RED-dead, MISRAC2004-14.1, MISRAC++2008-0-1-1, MISRAC2012-Rule-2.1_b

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. This check is identical to RED-unused-param, MISRAC2012-Rule-2.7

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

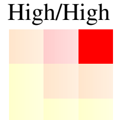
There are assignments from one field of a union to another.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

(Required) An object shall not be assigned to an overlapping object. This check is identical to UNION-overlap-assign, MISRAC2004-18.2, MISRAC2012-Rule-19.1

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:

```
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;
}
```

MISRAC++2008-0-3-2

Synopsis

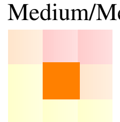
The return value for a library function that might return an error value is not used.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) If a function generates error information, then that error information shall be tested. This check is identical to LIB-return-error, MISRAC2004-16.10

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++2008-2-7-1

Synopsis

Detected /* 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. This check is identical to COMMENT-nested, MISRAC2004-2.3

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

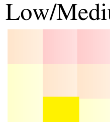
Commented-out code has been detected. (To allow comments to contain pseudo-code or code samples, only comments that end in `;`, `{`, or `}` characters are considered to be commented-out code.)

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) Sections of code shall not be "commented out" using C-style comments.

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

Commented-out code has been detected. (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.

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++2008-2-10-1

Synopsis

Two identifiers have names that can be confused with each other.

Enabled by default

Yes

Severity/Certainty

Low/Low

Full description

(Required) Different identifiers shall be typographically unambiguous.

Coding standards

MISRA C++ 2008 2-10-1

(Required) Different identifiers shall be typographically unambiguous.

Code examples

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

```
void example(void)
{
  char  idB_S;
  char  idB_5;
}
```

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

```
void example(void)
{
  char  idB_5rm;
  char  idB_irh;
}
```

MISRAC++2008-2-10-2 (C++ only)

Synopsis

There are identifier names that are not distinct from other names in an outer scope.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.

Coding standards

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:

```
extern int f2(void);
extern int f3(void);
extern int  n01_param_hides_var;
extern int  n02_var_hides_var;
void        n03_var_hides_function (void) {}

union       n04_var_hides_union_tag {
  int v1;
  unsigned int v2;
};
enum        n05_var_hides_enum_tag {
            n06_var_hides_enum_const,
};
extern int  n07_type_hides_var;

struct      n08_var_hides_class1 {
  int       n09_var_hides_mem;
};

class       n10_var_hides_class2 {
  int cm1;
};

void f1(int n01_param_hides_var) {
  int       n02_var_hides_var;
  int       n03_var_hides_function;
  int       n04_var_hides_union_tag;
  int       n05_var_hides_enum_tag;
  int       n06_var_hides_enum_const;

  switch(f2()) {
  case 1: {
    typedef int n07_type_hides_var;
    int n08_var_hides_class1;
    int n09_var_hides_mem;
    int n10_var_hides_class2;
    do {
      struct    n11_var_hides_struct_tag {
  int ff1;
      } b;
      if(f3()) {
  int     n11_var_hides_struct_tag = 1;
      }
    } while(f2());
  }
  }
}
```

```
namespace ns1 {
  int   n12_var_hides_var_ns;
  void f4(void) {
    int n12_var_hides_var_ns;
  }
}
```

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

```
namespace ns1 {
  int   n16_var_hides_var_ns;
}

namespace ns2 {
  void f2(void) {
    int n16_var_hides_var_ns;
  }
}
```

MISRAC++2008-2-10-3

Synopsis

A typedef with this name has already been declared.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) A typedef name (including qualification, if any) shall be a unique identifier. This check is identical to MISRAC2004-5.3, MISRAC2012-Rule-5.6

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;

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 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. This check is identical to MISRAC2004-5.4, MISRAC2012-Rule-5.7

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

An identifier is used that might 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. This check is identical to MISRAC2004-5.5

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++2008-2-10-6 (C++ only)

Synopsis

There is a clash with type names.

Enabled by default Yes

Severity/Certainty Low/Medium

Full description (Required) If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.

Coding standards MISRA C++ 2008 2-10-6

(Required) If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.

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

```
struct foo
{
  int x;
};

void foo();
```

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

```
void func()
{
  typedef struct vector { int x ; int y; int z; } a_vector;
          struct vector2 { int x ; int y; int z; } a_vector2;
}
```

MISRAC++2008-2-13-2

Synopsis Octal integer constants are used.

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. This check is identical to MISRAC2004-7.1, MISRAC2012-Rule-7.1

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

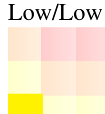
There are unsigned integer constants without a `U` suffix.

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. This check is identical to MISRAC2004-10.6, MISRAC2012-Rule-7.2

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:

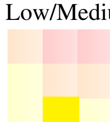
```
void example(void) {
  unsigned int x = 0x80000000u;
}
```

MISRAC++2008-2-13-4_a

Synopsis
Suffixes on floating-point constants are lower case.

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 = 2.41;
}
```

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

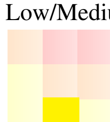
Suffixes on integer constants are lower case.

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

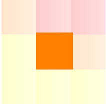
Non-inline functions have been defined in header files.

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. This check is identical to MISRAC2004-8.5_b

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

One or more external arrays are declared without their size being 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. This check is identical to MISRAC2004-8.12, MISRAC2012-Rule-8.11

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

There are uses of the basic types char, int, short, long, double, and float without a 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. This check is identical to MISRAC2004-6.3, MISRAC2012-Dir-4.6_a

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 char SCHAR;
typedef int INT;
typedef float FLOAT;

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 char SCHAR;
typedef int INT;
typedef float FLOAT;

INT func(FLOAT f, INT *pi)
{
  INT x;
  INT (*fp)(const SCHAR *);
}
```

MISRAC++2008-3-9-3

Synopsis

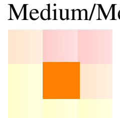
An expression provides access to the bit-representation of a floating-point variable.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) The underlying bit representations of floating-point values shall not be used. This check is identical to MISRAC2004-12.12_b

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

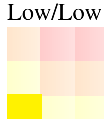
Arithmetic operators are used on boolean operands.

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. This check is identical to MISRAC2004-12.6_b

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;
}
```

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

```
void func()
{
  bool x;
  bool y;
  y = x && y;
}
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;

}
```

MISRAC++2008-4-5-2

Synopsis

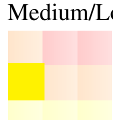
Unsafe operators are used on variables of enumeration type.

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 <, <=, >, >=.

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

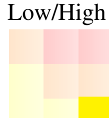
Arithmetic is performed on objects of type plain char, without an explicit signed or unsigned qualifier.

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. This check is identical to MISRAC2004-6.1

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:

```
typedef signed char INT8;
typedef unsigned char UINT8;

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:

```
typedef signed char INT8;
typedef unsigned char UINT8;

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

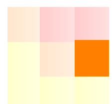
There are expressions that depend on the order of evaluation.

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. This check is identical to SPC-order, MISRAC2004-12.2_a, MISRAC2012-Rule-13.2_a, MISRAC2012-Rule-1.3_i

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

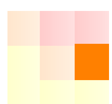
There are more than one read access with volatile-qualified type within a single sequence point.

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. This check is identical to SPC-volatile-reads, MISRAC2004-12.2_b, MISRAC2012-Rule-13.2_b

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:

```
void example(void) {
  int x;
  volatile int v;
  x = v + v;
}
```

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

```
int main(void) {
  volatile int i = 0;
  int x = i;
  i++;
  x = x * i;  //OK - statement is broken up
  return 0;
}
```

MISRAC++2008-5-0-1_c

Synopsis

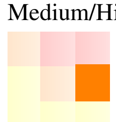
There are more than one modification access with volatile-qualified type within a single sequence point.

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. This check is identical to SPC-volatile-writes, MISRAC2004-12.2_c, MISRAC2012-Rule-13.2_c

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:

```
void example(void) {
  int x;
  volatile int 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

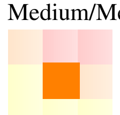
Parentheses to avoid implicit operator precedence are missing.

Enabled by default

No

Severity/Certainty

Medium/Medium



Full description

(Advisory) Limited dependence should be placed on C++ operator precedence rules in expressions. This check is identical to MISRAC2004-12.1

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

One or more cvalue expressions have been 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.
}
```

MISRAC++2008-5-0-4

Synopsis

One or more implicit integral conversions have been found that 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;
  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 One or more implicit floating-integral conversions were found.

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:

```
void f()
{
  float f32;
  int s32;
  s32 = f32; // Non-compliant
}
```

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

```
void f()
{
  float f32;
  int s32;
  f32 = static_cast< float > ( s32 ); // Compliant
}
```

MISRAC++2008-5-0-6 (C++ only)

Synopsis

One or more implicit integral or floating-point conversion were found that 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

One or more explicit floating-integral conversions of a cvalue expression were found.

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:

```
void f1 ( )
{
  int i;
  int j;
  float f;
  f = static_cast< float > ( i / j ); // Non-compliant
}
```

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

```
void f1 ( )
{
  int i;
  int j;
  int k;
  float f;
  k = i / j;
  f = static_cast< float > ( k ); // Compliant
}
```

MISRAC++2008-5-0-8

Synopsis

One or more explicit integral or floating-point conversions were found that 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

One or more explicit integral conversions were found that 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

A bitwise operation on unsigned char or unsigned short was found, that was 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. This check is identical to MISRAC2004-10.5

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;
  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
}
```

MISRAC++2008-5-0-13_a

Synopsis

Non-Boolean termination conditions were found 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. This check is identical to MISRAC2004-13.2_a, MISRAC2012-Rule-14.4_a

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
}
```

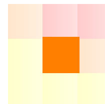
MISRAC++2008-5-0-13_b

Synopsis: Non-boolean termination conditions were found 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. This check is identical to MISRAC2004-13.2_b, MISRAC2012-Rule-14.4_b

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 were found 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. This check is identical to MISRAC2004-13.2_c, MISRAC2012-Rule-14.4_c

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 were found 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. This check is identical to MISRAC2004-13.2_d, MISRAC2012-Rule-14.4_d

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 were found.

Enabled by default Yes

Severity/Certainty

Low/Medium

Full description

(Required) The first operand of a conditional-operator shall have type bool. This check is identical to MISRAC2004-13.2_e

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(bool b) {
  int x;
  x = b ? 1 : 2;  //OK - b is a bool
}
```

MISRAC++2008-5-0-15_a

Synopsis

Pointer arithmetic that is not array indexing was found.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Array indexing shall be the only form of pointer arithmetic. This check is identical to MISRAC2004-17.4_a

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 applied to objects not defined as an array type was found.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Array indexing shall be the only form of pointer arithmetic. This check is identical to MISRAC2004-17.4_b

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 char UINT8;
typedef unsigned int UINT;

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 char UINT8;
typedef unsigned int UINT;

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 was found.

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 is identical to PTR-arith-stack, MISRAC2004-17.1_b

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 was found.

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 is identical to PTR-arith-var, MISRAC2004-17.1_c

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

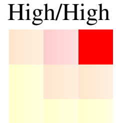
An array access is out of bounds.

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 check is identical to ARR-inv-index, MISRAC2012-Rule-18.1_a

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 example(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

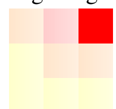
An array access might be out of bounds for some execution paths.

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 check is identical to ARR-inv-index-pos, MISRAC2012-Rule-18.1_b

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;
}
```

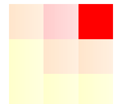
MISRAC++2008-5-0-16_e

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. This check is identical to ARR-inv-index-ptr, MISRAC2012-Rule-18.1_c

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

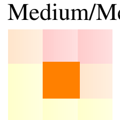
A pointer to an array might be used outside the array bounds.

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. This check is identical to ARR-inv-index-ptr-pos, MISRAC2012-Rule-18.1_d

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

Declarations that contain more than two levels of pointer indirection have been found.

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. This check is identical to MISRAC2004-17.5, MISRAC2012-Rule-18.5

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 were found.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Bitwise operators shall only be applied to operands of unsigned underlying type. This check is identical to MISRAC2004-12.7

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

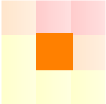
Old style casts (other than void casts) were found.

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. This check is identical to CAST-old-style

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 a const or volatile qualification were found.

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 check is identical to MISRAC2004-11.5, MISRAC2012-Rule-11.8

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:

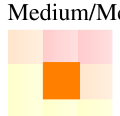
```
typedef unsigned short uint16_t;

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
}
```

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;
}
```

MISRAC++2008-5-2-7

Synopsis

A pointer to object type is cast to a pointer to a 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. This check is identical to MISRAC2004-11.4

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 from a pointer type to an integral type was found.

Enabled by default

No

Severity/Certainty

Low/Medium

Full description

(Advisory) A cast should not convert a pointer type to an integral type. This check is identical to MISRAC2004-11.3, MISRAC2012-Rule-11.4

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

The increment (++) and decrement (--) operators are being used 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. This check is identical to MISRAC2004-12.13, MISRAC2012-Rule-13.3

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:

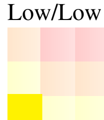
```
void example(char *src, char *dst) {
   while (*src) {
      *dst = *src;
      src++;
      dst++;
   }
}
```

MISRAC++2008-5-2-11_a (C++ only)

Synopsis
Overloaded && and || operators were found.

Enabled by default
Yes

Severity/Certainty
Low/Low



Full description
(Required) The comma operator, && operator and the || operator shall not be overloaded. This check is identical to LOGIC-overload

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

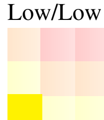
Overloaded comma operators were found.

Enabled by default

Yes

Severity/Certainty

Low/Low



Full description

(Required) The comma operator, && operator and the || operator shall not be overloaded. This check is identical to COMMA-overload

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

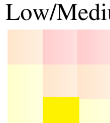
Operands of the logical operators (&&, ||, and !) were found that are not of type bool.

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. This check is identical to MISRAC2004-12.6_a

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 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:

```
typedef char boolean_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;
}
```

MISRAC++2008-5-3-2_a

Synopsis

Uses of unary minus on unsigned expressions were found.

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. This check is identical to MISRAC2012-Rule-10.1_R8, MISRAC2004-12.9

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 minus on unsigned expressions were found.

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. This check is identical to MISRAC2004-12.9

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
Occurances of overloaded & operators were found.

Enabled by default
Yes

Severity/Certainty
Low/Low

Full description

(Required) The unary & operator shall not be overloaded. This check is identical to PTR-overload

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

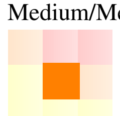
There are sizeof expressions that contain side effects.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) Evaluation of the operand to the sizeof operator shall not contain side effects. This check is identical to SIZEOF-side-effect, MISRAC2004-12.3

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

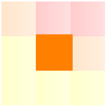
Possible out-of-range shifts were found.

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. This check is identical to ATH-shift-bounds, MISRAC2004-12.8, MISRAC2012-Rule-12.2

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 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;
}
```

MISRAC++2008-5-14-1

Synopsis

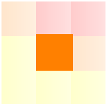
There are right-hand operands of && or || operators that contain side effects.

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. This check is identical to MISRAC2004-12.4, MISRAC2012-Rule-13.5

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

There are uses of the comma operator.

Enabled by default

Yes

Severity/Certainty

Low/High

Full description

(Required) The comma operator shall not be used. This check is identical to MISRAC2004-12.10, MISRAC2012-Rule-12.3

Coding standards

MISRA C++ 2008 5-18-1

(Required) The comma operator shall not be used.

Code examples

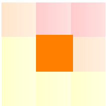
The following code example fails the check and will give a warning:

```
#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. This check is identical to EXPR-const-overflow, MISRAC2004-12.11
Coding standards	CWE 190

Integer Overflow or Wraparound

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

One or more assignment operators are used in sub-expressions.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Assignment operators shall not be used in sub-expressions. This check is identical to MISRAC2012-Rule-13.4_b

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

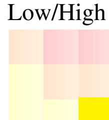
There are floating-point comparisons that use the == or != operators.

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. This check is identical to ATH-cmp-float, MISRAC2004-13.3

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

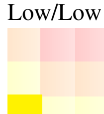
There are stray semicolons on the same line as other code.

Enabled by default

No

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. This check is identical to EXP-stray-semicolon, MISRAC2004-14.3

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

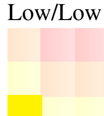
There are missing braces in `do ... while` statements.

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. This check is identical to MISRAC2004-14.8_a, MISRAC2012-Rule-15.6_a

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

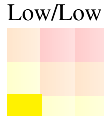
There are missing braces in `for` statements.

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. This check is identical to MISRAC2004-14.8_b, MISRAC2012-Rule-15.6_b

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

There are 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. This check is identical to MISRAC2004-14.8_c, MISRAC2012-Rule-15.6_d

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

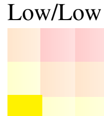
There are missing braces in `while` statements.

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. This check is identical to MISRAC2004-14.8_d, MISRAC2012-Rule-15.6_e

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

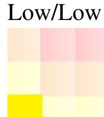
There are missing braces in `if`, `else`, or `else if` statements.

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. This check is identical to MISRAC2004-14.9, MISRAC++2008-6-4-1, MISRAC2012-Rule-15.6_c

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 <stdlib.h>

void example(void) {
  if (rand());
  if (rand());
  else;
}
```

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

```
#include <stdlib.h>

void example(void) {
  if (rand()) {
  }
  if (rand()) {
  } else {
  }
  if (rand()) {
  } else if (rand()) {
  }
}
```

MISRAC++2008-6-4-2

Synopsis

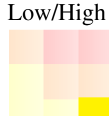
`If ... else if` constructs that are not terminated with an `else` clause were detected.

Enabled by default

Yes

Severity/Certainty

Low/High



Full description

(Required) All if ... else if constructs shall be terminated with an else clause. This check is identical to MISRAC2004-14.10, MISRAC2012-Rule-15.7

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

Detected 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. This check is identical to MISRAC2004-15.0, MISRAC2012-Rule-16.1

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 were found 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. This check is identical to MISRAC2004-15.1, MISRAC2012-Rule-16.2

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

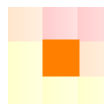
Non-empty switch cases were found that are not terminated by a break.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) An unconditional throw or break statement shall terminate every non-empty switch-clause. This check is identical to MISRAC2004-15.2, MISRAC2012-Rule-16.3

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) {

  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;
  }

}
```

MISRAC++2008-6-4-6

Synopsis

Switch statements without a default clause, or with a default clause that is not the final clause, were found.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The final clause of a switch statement shall be the default-clause. This check is identical to MISRAC2004-15.3

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 was found that represents 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. This check is identical to MISRAC2004-15.4, MISRAC2012-Rule-16.7

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

One or more switch statements without a case clause were found.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Every switch statement shall have at least one case-clause. This check is identical to MISRAC2004-15.5

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 were found 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. This check is identical to MISRAC2004-13.4, MISRAC2012-Rule-14.1_a

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

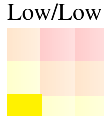
A loop counter was found that might not match the loop condition test.

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+= 2) {}
}
```

MISRAC++2008-6-5-3

Synopsis

A `for` loop counter variable was found that 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. This check is identical to MISRAC2004-13.6, MISRAC2012-Rule-14.2

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

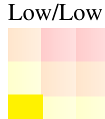
A potentially inconsistent loop counter modification was found.

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) {}
}
```

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

```
void example(void)
{
  bool b;
  for(int i = 0; i != 10 || b; i-=2) {}
}
```

MISRAC++2008-6-5-6

Synopsis

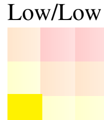
A non-boolean variable was detected that is modified in the loop and used as loop condition.

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

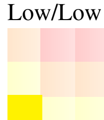
The destination of a goto statement is a nested code block.

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. This check is identical to MISRAC2012-Rule-15.3

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

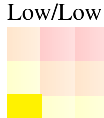
A goto statement is declared after the destination label.

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. This check is identical to MISRAC2012-Rule-15.2

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

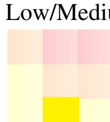
One or more loops have more than one termination point.

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. This check is identical to MISRAC2012-Rule-15.4

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:

```
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;
    }
  }
}
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
    }
  }
}
```

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

```
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
    {
    }
  }
}
void example(void)
{
   int i = 0;
   for (i = 0; i < 10 && i != 9; i++) {
     if (i == 9) {
       break;
     }
   }
}
```

MISRAC++2008-6-6-5

Synopsis

One or more functions have multiple exit points or an exit point that is not 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. This check is identical to MISRAC2004-14.7, MISRAC2012-Rule-15.5

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 that is not modified after its initialization is not `const` qualified.

Enabled by default Yes

Severity/Certainty

Low/Medium

Full description

(Required) A variable which is not modified shall be const qualified. This check is identical to

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 parameter in a function that is not modified by the function is not `const` qualified.

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. This check is identical to MISRAC2004-16.7

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

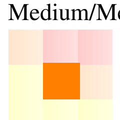
There are conversions to enum type that are out of range of the enumeration.

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. This check is identical to ENUM-bounds

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;
}
```

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

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

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

MISRAC++2008-7-4-3

Synopsis There are inline assembler 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. This check is identical to MISRAC2004-2.1, MISRAC2012-Dir-4.3

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 example(void)
{
  int r;
  asm("");
  return r + 1;
}
```

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

```
int example(int x)
{
  asm("");
  return x;
}
```

MISRAC++2008-7-5-1_a (C++ only)

Synopsis

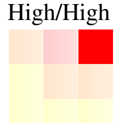
A stack object is returned from a function as a reference.

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. This check is identical to MEM-stack-ref

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

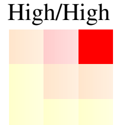
A function might return an address on the stack.

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. This check is identical to MEM-stack, MISRAC2004-17.6_a, MISRAC2012-Rule-18.6_a

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 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:

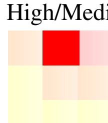
```
#include <stdlib.h>

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 Detected a stack address stored 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 check is identical to MEM-stack-global, MISRAC2004-17.6_b, MISRAC2012-Rule-18.6_b

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

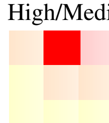
Detected a stack address in the field of a global struct.

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 check is identical to MEM-stack-global-field, MISRAC2004-17.6_c, MISRAC2012-Rule-18.6_c

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

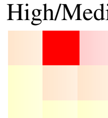
Detected a stack address stored in a parameter of pointer or array type.

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 check is identical to MEM-stack-param, MISRAC2004-17.6_d, MISRAC2012-Rule-18.6_d, MISRAC2012-Rule-1.3_s

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

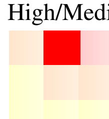
Detected a stack address stored via a reference parameter.

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. This check is identical to MEM-stack-param-ref, MISRAC2012-Rule-1.3_s

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

There are 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. This check is identical to MISRAC2004-16.2_a, MISRAC2012-Rule-17.2_a

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

There are 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. This check is identical to MISRAC2004-16.2_b, MISRAC2012-Rule-17.2_b

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

There are declarations that contain more than 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

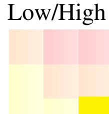
There are functions defined using the ellipsis (...) notation.

Enabled by default

Yes

Severity/Certainty

Low/High



Full description

(Required) Functions shall not be defined using the ellipsis notation. This check is identical to MISRAC2004-16.1

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

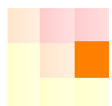
For some execution paths, no return statements are executed in functions with a non-void return type.

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. This check is identical to SPC-return, MISRAC2004-16.8, MISRAC2012-Rule-17.4

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

The addresses of one or more functions are taken without an 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 &. This check is identical to MISRAC2004-16.9

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

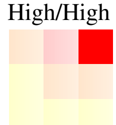
In all execution paths, variables are read before they are assigned a value.

Enabled by default

Yes

Severity/Certainty

High/High



Full description

(Required) All variables shall have a defined value before they are used. This check is identical to SPC-uninit-var-all, MISRAC2004-9.1_a, MISRAC2012-Rule-9.1_e, MISRAC2012-Rule-1.3_j

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

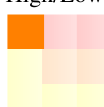
In some execution paths, variables might be read before they are assigned a value.

Enabled by default

Yes

Severity/Certainty

High/Low



Full description

(Required) All variables shall have a defined value before they are used. This check is identical to SPC-uninit-var-some, MISRAC2004-9.1_b, MISRAC2012-Rule-9.1_f, MISRAC2012-Rule-1.3_k

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

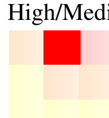
One or more uninitialized or NULL pointers are dereferenced.

Enabled by default

Yes

Severity/Certainty

High/Medium



Full description

(Required) All variables shall have a defined value before they are used. This check is identical to PTR-uninit, MISRAC2004-9.1_c

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

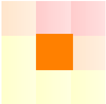
There are one or more non-zero array initializations that do not exactly match the structure of the array declaration.

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. This check is identical to MISRAC2004-9.2

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][3] = { { 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

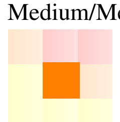
A member function qualified as `const` returns a pointer member variable.

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 check is identical to CONST-member-ret

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

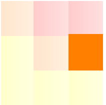
Member functions return non-const handles to members.

Enabled by default

Yes

Severity/Certainty

Medium/High



Full description

(Required) Member functions shall not return non-const handles to class-data. This check is identical to CPU-return-ref-to-class-data

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 Unions were found.

Enabled by default Yes

Severity/Certainty Low/Medium

Full description (Required) Unions shall not be used. This check is identical to MISRAC2004-18.4, MISRAC2012-Rule-19.2

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

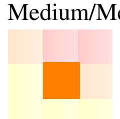
Bitfields of plain int type were found.

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;
};
```

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

```
struct good {
  unsigned int x:3;
};
```

MISRAC++2008-9-6-3

Synopsis

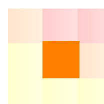
Bitfields of plain int type were found.

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 {
  unsigned int x:3;
};
```

MISRAC++2008-9-6-4

Synopsis

Signed single-bit bitfields (excluding anonymous fields) were found.

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. This check is identical to STRUCT-signed-bit, MISRAC2004-6.5, MISRAC2012-Rule-6.2

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

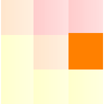
A virtual member function is called in a class constructor.

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. This check is identical to CPU-ctor-call-virt

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>

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

class B: public A {
public:
  virtual void f() const { std::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>

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

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

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

MISRAC++2008-12-1-1_b (C++ only)

Synopsis

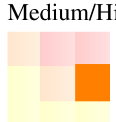
A virtual member function is called in a class destructor.

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. This check is identical to CPU-dtor-call-virt

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>

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

class B: public A {
public:
  virtual void f() const { std::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>

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

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

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

MISRAC++2008-12-1-3 (C++ only)

Synopsis

Constructors that can be called with a single argument of fundamental type are not 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. This check is identical to CPU-ctor-implicit

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:

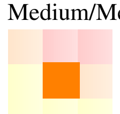
```
class C{
  explicit C(double x){} //OK
};
```

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. This check is identical to THROW-ptr

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

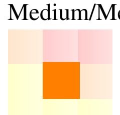
Throw of NULL integer constant.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) NULL shall not be thrown explicitly. This check is identical to THROW-null

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:

```
#include <stdlib.h>

void example(void)
{
  try {
    throw ( NULL );           // Non-compliant
  }
  catch ( int i ) {       // NULL exception handled here
    // ...
  }
  catch ( const char * ) { // Developer may expect it to be
caught here
    // ...
  }
}
```

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

```
#include <stdlib.h>

void example(void)
{
  char * p = NULL;
  try {
    throw ( p );              // Compliant
  }
  catch ( int i ) {
    // ...
  }
  catch ( const char * ) { // Exception handled here
    // ...
  }
}
```

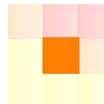
MISRAC++2008-15-1-3 (C++ only)

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. This check is identical to THROW-empty

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

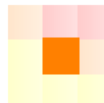
There are exceptions thrown without a handler in some call paths that lead to that point.

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. This check is identical to THROW-static

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
};

// 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
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;
    }
}
```

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

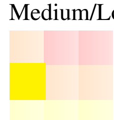
There are no default exception handlers for try.

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 This check is identical to THROW-main

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:

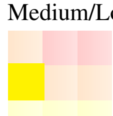
```
int main()
{
  try
  {
    throw;
  }
  catch (...) {}
  // spacer
  try {}
  catch (int i) {}
  catch (...) {}
  return 0;
}
```

MISRAC++2008-15-3-3 (C++ only)

Synopsis
One or more exception handlers in a constructor or destructor accesses a non-static member variable that might 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. This check is identical to CATCH-xtor-bad-member

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
      }
    }
  }
};
```

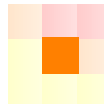
MISRAC++2008-15-3-4 (C++ only)

Synopsis There are calls to functions that are explicitly declared to throw an exception type that are 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. This check is identical to THROW-unhandled

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);
}
```

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

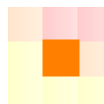
Exception objects are caught by value, not by reference.

Enabled by default

Yes

Severity/Certainty

Medium/Medium



Full description

(Required) A class type exception shall always be caught by reference. This check is identical to CATCH-object-slicing

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:

```
typedef char char_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:

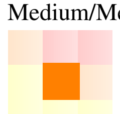
```
typedef char char_t;

// 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 might 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. This check is identical to COP-dtor-throw

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;
};
```

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

Found occurrences of #undef.

Enabled by default

Yes

Severity/Certainty Low/Low

Full description (Required) #undef shall not be used. This check is identical to MISRAC2004-19.6, MISRAC2012-Rule-20.5

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:

```
#define SYM
#undef SYM
```

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

```
#define SYM
```

MISRAC++2008-16-0-4

Synopsis Definitions of function-like macros were found.

Enabled by default Yes

Severity/Certainty Low/Low

Full description (Required) Function-like macros shall not be defined. This check is identical to MISRAC2004-19.7, MISRAC2012-Dir-4.9

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

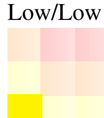
Definitions of macros that are not include guards were found.

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.

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 were found.

Enabled by default

Yes

Severity/Certainty

Low/Low

Full description

(Required) Include guards shall be provided. This check is identical to MISRAC2004-19.15, MISRAC2012-Dir-4.10

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

There are 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.

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

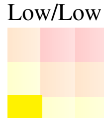
There are illegal characters in header file names.

Enabled by default

No

Severity/Certainty

Low/Low



Full description

(Advisory) The backslash character should not occur in a header file name.

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

There are 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 check is identical to DEFINE-hash-multiple, MISRAC2004-19.12

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:

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

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

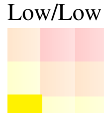
```
#define A(x)#x/* Compliant */
```

MISRAC++2008-16-3-2

Synopsis — # and ## operators were found in macro definitions.

Enabled by default — No

Severity/Certainty — Low/Low



Full description — (Advisory) The # and ## operators should not be used. This check is identical to MISRAC2004-19.13, MISRAC2012-Rule-20.10

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:

```
#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

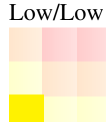
Detected a #define or #undef of a reserved identifier in the standard library.

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. This check is identical to MISRAC2004-20.1, MISRAC2012-Rule-21.1

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

One or more library functions are being overridden.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The names of standard library functions shall not be overridden. This check is identical to MISRAC2004-20.2, MISRAC2012-Rule-21.2

Coding standards

MISRA C:2004 20.2

(Required) The names of Standard Library macros, objects, and functions shall not be reused.

MISRA C:2012 Rule-21.2

(Required) A reserved identifier or macro name shall not be declared

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:

```
extern "C" void bar(void);
void foo(void) {}
```

MISRAC++2008-17-0-5

Synopsis

Found 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. This check is identical to MISRAC2004-20.7, MISRAC2012-Rule-21.4

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

C library includes were found.

Enabled by default

Yes

Severity/Certainty

Low/Low

Full description

(Required) The C library shall not be 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

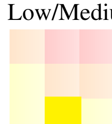
Uses of atof, atoi, atol and atoll were found.

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. This check is identical to MISRAC2004-20.10, MISRAC2012-Rule-21.7

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

Uses of abort, exit, getenv, and system were found.

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. This check is identical to MISRAC2004-20.11, MISRAC2012-Rule-21.8

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

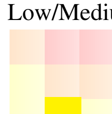
Uses of time.h functions: asctime, clock, ctime, difftime, gmtime, localtime, mktime, strftime, and time were found.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description (Required) The time handling functions of library ctime shall not be used. This check is identical to MISRAC2004-20.12, MISRAC2012-Rule-21.10

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 Uses of strcpy, strcmp, strcat, strchr, strspn, strcspn, strpbrk, strrchr, strstr, strtok, or strlen were found.

Enabled by default Yes

Severity/Certainty Low/Medium

Full description (Required) The unbounded functions of library <cstring> shall not 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

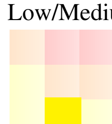
Uses of the built-in function offsetof were found.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) The macro offsetof shall not be used. This check is identical to MISRAC2004-20.6

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>

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

Uses of malloc, calloc, realloc, or free were found.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) Dynamic heap memory allocation shall not be used. This check is identical to MISRAC2004-20.4, MISRAC2012-Rule-21.3

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

Uses of signal.h were found.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The signal handling facilities of csignal shall not be used. This check is identical to MISRAC2004-20.8, MISRAC2012-Rule-21.5

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

Uses of errno were found.

Enabled by default

Yes

Severity/Certainty

Low/Medium

Full description

(Required) The error indicator errno shall not be used. This check is identical to MISRAC2004-20.5

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 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

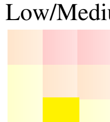
Uses of stdio.h were found.

Enabled by default

Yes

Severity/Certainty

Low/Medium



Full description

(Required) The stream input/output library cstdio shall not be used. This check is identical to MISRAC2004-20.9, MISRAC2012-Rule-21.6

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) {
}
```

Mapping of CERT rules to C-STAT checks

The following pages contain information about:

- Computer Emergency Response Team (CERT)

Computer Emergency Response Team (CERT)

The *Computer Emergency Response Team* (CERT) Secure Coding Standard is a collection of guidelines—either rules or recommendations—designed to eliminate vulnerabilities in C and C++ code.

This table lists all CERT guidelines that can be mapped to one or more C-STAT checks. This helps you to identify which checks to enable or disable to verify a certain CERT guideline. Note that code with one of the listed guidelines will not necessarily fail each associated check, but it might fail some.

CERT ID	CERT guideline	Associated C-STAT checks
DCL01-C	Do not reuse variable names in sub-scopes.	RED-local-hides-global RED-local-hides-local RED-local-hides-member (C++ only) RED-local-hides-param
DCL16-C	Use `L` or `l` to indicate a long value.	MISRAC++2008-2-13-4_b
DCL20-C	Always specify `void` if a function accepts no arguments.	FUNC-unprototyped-all FUNC-unprototyped-used MISRAC2004-16.5 MISRAC2012-Rule-8.2_a

Table 7: Mapping of CERT rules to C-STAT checks

CERT ID	CERT guideline	Associated C-STAT checks
DCL30-C	Declare objects with appropriate storage duration.	MEM-stack MEM-stack-global MEM-stack-global-field MEM-stack-param MEM-stack-param-ref (C++ only) MISRAC2004-17.6_a MISRAC2004-17.6_b MISRAC2004-17.6_c MISRAC2004-17.6_d MISRAC++2008-7-5-1_a (C++ only) MISRAC++2008-7-5-1_b MISRAC++2008-7-5-2_a MISRAC++2008-7-5-2_b MISRAC++2008-7-5-2_c MISRAC++2008-7-5-2_d (C++ only) MISRAC2012-Rule-1.3_q MISRAC2012-Rule-1.3_r MISRAC2012-Rule-1.3_s MISRAC2012-Rule-18.6_a MISRAC2012-Rule-18.6_b MISRAC2012-Rule-18.6_c MISRAC2012-Rule-18.6_d
DCL31-C	Declare identifier before using them.	DECL-implicit-int FUNC-implicit-decl FUNC-unprototyped-used MISRAC2004-8.1 MISRAC2004-8.2 MISRAC2012-Rule-8.1 MISRAC2012-Rule-17.3
EXP01-C	Do not take the size of a pointer to determine the size of the pointed–to type.	MEM-malloc-sizeof-ptr
EXP06-C	Operands to the `sizeof` operator should not contain side effects.	SIZEOF-side-effect MISRAC2004-12.3 MISRAC++2008-5-3-4 MISRAC2012-Rule-13.6

Table 7: Mapping of CERT rules to C-STAT checks

CERT ID	CERT guideline	Associated C-STAT checks
EXP10-C	Do not depend on the order of evaluation of subexpressions or the order in which size effects take place.	SPC-order SPC-volatile-reads SPC-volatile-writes MISRAC2004-12.2_a MISRAC2004-12.2_b MISRAC2004-12.2_c MISRAC++2008-5-0-1_a MISRAC++2008-5-0-1_b MISRAC++2008-5-0-1_c MISRAC2012-Rule-1.3_i MISRAC2012-Rule-13.2_a MISRAC2012-Rule-13.2_b MISRAC2012-Rule-13.2_c
EXP12-C	Do not ignore values returned by functions.	LIB-return-const
EXP15-C	Do not place a semicolon on the same line as an `if`, `for`, or `while` statement.	EXP-null-stmt EXP-stray-semicolon MISRAC2004-14.3 MISRAC++2008-6-2-3
EXP16-C	Do not compare function pointers to constant values.	FPT-misuse MISRAC2012-Rule-1.3_m
EXP17-C	Do not perform bitwise operations in conditional expressions.	RED-cond-always RED-cond-never MISRAC++2008-0-1-2_a MISRAC++2008-0-1-2_b MISRAC2012-Rule-14.3_a MISRAC2012-Rule-14.3_b
EXP18-C	Do not perform assignments in selection statements.	EXP-cond-assign MISRAC2012-Rule-13.4_a

Table 7: Mapping of CERT rules to C-STAT checks

CERT ID	CERT guideline	Associated C-STAT checks
EXP19-C	Use braces for the body of an `if`, `for`, or `while` statement.	MISRAC2004-14.8_a MISRAC2004-14.8_b MISRAC2004-14.8_c MISRAC2004-14.8_d MISRAC2004-14.9 MISRAC++2008-6-3-1_a MISRAC++2008-6-3-1_b MISRAC++2008-6-3-1_c MISRAC++2008-6-3-1_d MISRAC++2008-6-4-1 MISRAC2012-Rule-15.6_a MISRAC2012-Rule-15.6_b MISRAC2012-Rule-15.6_c MISRAC2012-Rule-15.6_d MISRAC2012-Rule-15.6_e
EXP30-C	Do not depend on order of evaluation between sequence points.	SPC-order SPC-volatile-reads SPC-volatile-writes MISRAC2004-12.2_a MISRAC2004-12.2_b MISRAC2004-12.2_c MISRAC++2008-5-0-1_a MISRAC++2008-5-0-1_b MISRAC++2008-5-0-1_c MISRAC2012-Rule-1.3_i MISRAC2012-Rule-13.2_a MISRAC2012-Rule-13.2_b MISRAC2012-Rule-13.2_c

Table 7: Mapping of CERT rules to C-STAT checks

CERT ID	CERT guideline	Associated C-STAT checks
EXP33-C	Do not reference uninitialized memory.	ITR-uninit (C++ only) PTR-uninit PTR-uninit-pos SPC-uninit-arr-all SPC-uninit-struct SPC-uninit-struct-field SPC-uninit-struct-field-heap SPC-uninit-var-all MISRAC2004-1.2_a MISRAC2004-1.2_b MISRAC2004-9.1_a MISRAC2004-9.1_c MISRAC++2008-8-5-1_a MISRAC++2008-8-5-1_c MISRAC2012-Rule-1.3_j MISRAC2012-Rule-9.1_a MISRAC2012-Rule-9.1_b MISRAC2012-Rule-9.1_c MISRAC2012-Rule-9.1_d MISRAC2012-Rule-9.1_e
EXP34-C	Do not dereference null pointers.	PTR-null-assign PTR-null-assign-fun-pos PTR-null-assign-pos PTR-null-cmp-aft PTR-null-cmp-bef PTR-null-cmp-bef-fun PTR-null-fun-pos SEC-NULL-assignment SEC-NULL-assignment-fun-pos SEC-NULL-cmp-aft SEC-NULL-cmp-bef SEC-NULL-cmp-bef-fun
EXP39-C	Do not access a variable through a pointer of an incompatible type.	UNION-type-punning MISRAC2004-12.12_a
INT04-C	Enforce limits on integer values originating from untrusted sources.	SEC-BUFFER-tainted-alloc-size SEC-BUFFER-tainted-copy-length SEC-BUFFER-tainted-index

Table 7: Mapping of CERT rules to C-STAT checks

CERT ID	CERT guideline	Associated C-STAT checks
INT06-C	Use `strtol()` or a related function to convert a string token to an integer.	MISRAC2004-20.10 MISRAC++2008-18-0-2 MISRAC2012-Rule-21.7
INT07-C	Use only explicitly signed or unsigned char type for numeric values.	MISRAC2004-6.1 MISRAC++2008-4-5-3
INT13-C	Use bitwise operators only on unsigned operands.	MISRAC2004-12.7 MISRAC++2008-5-0-21
INT31-C	Ensure that integer conversions do not result in lost or misinterpreted data.	ATH-overflow ATH-overflow-cast
INT33-C	Ensure that division and modulo operations do not result in divide-by-zero errors.	ATH-div-0 ATH-div-0-assign ATH-div-0-cmp-aft ATH-div-0-cmp-bef ATH-div-0-interval ATH-div-0-pos MISRAC2004-1.2_c MISRAC2004-1.2_d MISRAC2004-1.2_e MISRAC2004-1.2_f MISRAC2004-1.2_g MISRAC2004-1.2_h MISRAC2012-Rule-1.3_a MISRAC2012-Rule-1.3_b MISRAC2012-Rule-1.3_c MISRAC2012-Rule-1.3_d MISRAC2012-Rule-1.3_e MISRAC2012-Rule-1.3_f SEC-DIV-0-compare-after SEC-DIV-0-compare-before
INT34-C	Do not shift a negative number of bits or more bits than exist in the operand.	ATH-shift-bounds MISRAC2004-12.8 MISRAC++2008-5-8-1 MISRAC2012-Rule-12.2
FLP00-C	Understand the limitations of floating-point numbers.	ATH-cmp-float
FLP06-C	Understand that floating-point arithmetic in C is inexact.	MISRAC2004-13.3 MISRAC++2008-6-2-2

Table 7: Mapping of CERT rules to C-STAT checks

CERT ID	CERT guideline	Associated C-STAT checks
ARR01-C	Do not apply the `sizeof` operator to a pointer when taking the size of an array.	MEM-malloc-sizeof-ptr
ARR33-C	Guarantee that copies are made into storage of sufficient size.	ARR-inv-index ARR-inv-index-pos ARR-inv-index-ptr ARR-inv-index-ptr-pos MISRAC++2008-5-0-16_c MISRAC++2008-5-0-16_d MISRAC++2008-5-0-16_e MISRAC++2008-5-0-16_f MISRAC2012-Rule-18.1_a MISRAC2012-Rule-18.1_b MISRAC2012-Rule-18.1_c MISRAC2012-Rule-18.1_d
ARR37-C	Do not add or subtract an integer to a pointer to a non-array object.	PTR-arith-field MISRAC2004-17.1_a
STR31-C	Guarantee that storage for strings has sufficient space for character data and the null terminator.	LIB-sprintf-overrun LIB-strcat-overrun LIB-strcat-overrun-pos LIB-strcpy-overrun LIB-strcpy-overrun-pos LIB-strcpy-overrun LIB-strcpy-overrun-pos MISRAC2012-Rule-1.3_v MISRAC2012-Rule-1.3_w SEC-BUFFER-sprintf-overrun SEC-BUFFER-strcat-overrun SEC-BUFFER-strcat-overrun-pos SEC-BUFFER-strcpy-overrun SEC-BUFFER-strcpy-overrun-pos SEC-BUFFER-strncpy-overrun SEC-BUFFER-strncpy-overrun-pos

Table 7: Mapping of CERT rules to C-STAT checks

CERT ID	CERT guideline	Associated C-STAT checks
MSC07-C	Detect and remove dead code.	RED-case-reach RED-dead MISRAC++2008-0-1-1 MISRAC++2008-0-1-2_c MISRAC++2008-0-1-9 MISRAC2012-Rule-2.1_a MISRAC2012-Rule-2.1_b
MSC12-C	Detect and remove code that has no effect.	RED-no-effect MISRAC2004-14.2 MISRAC2012-Rule-2.2_a
MSC13-C	Detect and remove unused values.	RED-unused-assign RED-unused-var-all MISRAC++2008-0-1-3 MISRAC2012-Rule-2.2_b
MSC17-C	Finish every set of statements associated with a case label, with a break statement.	SWITCH-fall-through MISRAC2004-15.2 MISRAC++2008-6-4-5 MISRAC2012-Rule-16.3
MSC21-C	Use robust loop termination conditions.	MISRAC++2008-6-5-2
MSC37-C	Ensure that control never reaches the end of a non-void function.	MISRAC2004-16.8 MISRAC++2008-8-4-3 MISRAC2012-Rule-17.4
DCL01-CPP	Do not reuse variable names in sub-scopes.	RED-local-hides-global RED-local-hides-local RED-local-hides-member (C++ only) RED-local-hides-param
DCL16-CPP	Use `L`, not `l`, to indicate a long value.	MISRAC++2008-2-13-4_b
EXP05-CPP	Do not use C-style casts.	CAST-old-style (C++ only) MISRAC++2008-5-2-4 (C++ only)
EXP06-CPP	Operands to the `sizeof` operator should not contain side effects.	SIZEOF-side-effect MISRAC2004-12.3 MISRAC++2008-5-3-4 MISRAC2012-Rule-13.6
EXP19-CPP	Do not perform assignments in conditional expressions.	EXP-cond-assign MISRAC2012-Rule-13.4_a

Table 7: Mapping of CERT rules to C-STAT checks

CERT ID	CERT guideline	Associated C-STAT checks
FLP35-CPP	Take granularity into account when comparing floating-point values.	ATH-cmp-float MISRAC2004-13.3 MISRAC++2008-6-2-2
ARR32-CPP	Do not use iterators invalidated by container modification.	ITR-invalidated (C++ only)
CTR35-CPP	Do not allow loops to iterate beyond the end of an array or container.	ITR-end-cmp-aft (C++ only)
MEM42-CPP	Ensure that copy assignment operators do not damage an object that is copied to itself.	COP-assign-op-self (C++ only)
ERR09-CPP	Throw anonymous temporaries (and catch by reference).	CATCH-object-slicing (C++ only) THROW-ptr MISRAC++2008-15-0-2 MISRAC++2008-15-3-5 (C++ only)
ERR33-CPP	Destructors must not throw exceptions.	COP-dtor-throw (C++ only) MISRAC++2008-15-5-1 (C++ only)
ERR34-CPP	Do not use `longjmp()` or setjmp().	MISRAC2004-20.7 MISRAC++2008-17-0-5 MISRAC2012-Rule-21.4
ERR38-CPP	Deallocation functions must not throw exceptions.	CPU-delete-throw (C++ only)
OOP30-CPP	Do not invoke virtual functions from constructors or destructors.	CPU-ctor-call-virt (C++ only) CPU-dtor-call-virt (C++ only) MISRAC++2008-12-1-1_a (C++ only) MISRAC++2008-12-1-1_b (C++ only)
OOP32-CPP	Ensure that single-argument constructors are marked `explicit`.	CPU-ctor-implicit (C++ only) MISRAC++2008-12-1-3 (C++ only)
OOP34-CPP	Ensure the proper destructor is called for polymorphic objects.	CPU-nonvirt-dtor (C++ only)
OOP35-CPP	Do not return references to private data.	CPU-return-ref-to-class-data (C++ only) MISRAC++2008-9-3-2 (C++ only)
OOP37-CPP	Constructor initializers should be ordered correctly.	COP-init-order (C++ only)

Table 7: Mapping of CERT rules to C-STAT checks

CERT ID	CERT guideline	Associated C-STAT checks
MSC215-CPP	Use inequality to terminate a loop whose counter changes by more than one.	MISRAC++2008-6-5-2

Table 7: Mapping of CERT rules to C-STAT checks