NAME

extraclangtools - Extra Clang Tools Documentation

Welcome to the clang-tools-extra project which contains extra tools built using Clang's tooling APIs.

EXTRA CLANG TOOLS 15.0.7 RELEASE NOTES

Written by the *LLVM Team*

Introduction

This document contains the release notes for the Extra Clang Tools, part of the Clang release 15.0.7. Here we describe the status of the Extra Clang Tools in some detail, including major improvements from the previous release and new feature work. All LLVM releases may be downloaded from the *LLVM releases web site*.

For more information about Clang or LLVM, including information about the latest release, please see the *Clang Web Site* or the *LLVM Web Site*.

Note that if you are reading this file from a Git checkout or the main Clang web page, this document applies to the *next* release, not the current one. To see the release notes for a specific release, please see the *releases page*.

What's New in Extra Clang Tools 15.0.7?

Some of the major new features and improvements to Extra Clang Tools are listed here. Generic improvements to Extra Clang Tools as a whole or to its underlying infrastructure are described first, followed by tool-specific sections.

Major New Features

...

Improvements to clangd
Inlay hints

- **Provide hints for:**
  - Lambda return types.
  - Forwarding functions using the underlying function call.
- Support for standard LSP 3.17 inlay hints protocol.
- Designator inlay hints are enabled by default.

Diagnostics

- Improved Fix-its of some clang-tidy checks when applied with clangd.
- Clangd now produces diagnostics for forwarding functions like make_unique.
- Include cleaner analysis can be disabled with the **Diagnostics.Includes.IgnoreHeader** config option.
- Include cleaner doesn’t diagnose exporting headers.
- clang-tidy and include cleaner diagnostics have links to their documentation.

Semantic Highlighting

- Semantic highlighting works for tokens that span multiple lines.
- Mutable reference parameters in function calls receive **usedAsMutableReference** modifier.

Hover

- Hover displays desugared types by default now.

Code completion

- Improved ranking/filtering for ObjC method selectors.

- Support for C++20 concepts and requires expressions.

Signature help

- Signature help for function pointers.
- Provides hints using underlying functions in forwarded calls.

Cross-references
Code Actions

- New code action to generate ObjC initializers.
- New code action to generate move/copy constructors/assignments.
- Extract to function works for methods in addition to free functions.
- Related diagnostics are attached to code actions response, if any.
- Extract variable works in C and ObjC files.
- Fix to define outline when the parameter has a braced initializer.

Miscellaneous

- Include fixer supports symbols inside macro arguments.
- Dependent autos are now deduced when there’s a single instantiation.
- Support for symbols exported with using declarations in all features.
- Fixed background-indexing priority for M1 chips.
- Indexing for standard library symbols.
- ObjC framework includes are spelled properly during include insertion operations.

Improvements to clang-doc

The improvements are...

Improvements to clang-query

The improvements are...

Improvements to clang-rename

The improvements are...

Improvements to clang-tidy

- Added trace code to help narrow down any checks and the relevant source code that result in crashes.

- Clang-tidy now consideres newlines as separators of single elements in the *Checks* section in *.clang-tidy* configuration files. Where previously a comma had to be used to distinguish elements in this list from each other, newline characters now also work as separators in the parsed YAML. That means it is advised to use YAML's block style initiated by the pipe character | for the *Checks* section in order to benefit from the easier syntax that works without commas.

- Fixed a regression introduced in clang-tidy 14.0.0, which prevented NOLINTs from suppressing diagnostics associated with macro arguments. This fixes *Issue 55134*.

- Added an option -verify-config which will check the config file to ensure each *Checks* and *CheckOptions* entries are recognised.

- .clang-tidy files can now use the more natural dictionary syntax for specifying *CheckOptions*.

New checks

- New *bugprone-shared-ptr-array-mismatch* check.

  Finds initializations of C++ shared pointers to non-array type that are initialized with an array.

- New *bugprone-unchecked-optional-access* check.

  Warns when the code is unwrapping a *std::optional<T>*, *absl::optional<T>*, or *base::Optional<T>* object without assuring that it contains a value.

- New *misc-confusable-identifiers* check.

  Detects confusable Unicode identifiers.

- New *bugprone-assignment-in-if-condition* check.

Warns when there is an assignment within an if statement condition expression.

- New *misc-const-correctness* check.

  Detects unmodified local variables and suggest adding **const** if the transformation is possible.

- New *modernize-macro-to-enum* check.

  Replaces groups of adjacent macros with an unscoped anonymous enum.

- New *portability-std-allocator-const* check.

  Report use of **std::vector<const T>** (and similar containers of const elements). These are not allowed in standard C++ due to undefined **std::allocator<const T>**. They do not compile with libstdc++ or MSVC. Future libc++ will remove the extension (*D120996 <https://reviews.llvm.org/D120996>*).

New check aliases

- New alias *cppcoreguidelines-macro-to-enum* to *modernize-macro-to-enum* was added.

Changes in existing checks

- Fixed nonsensical suggestion of *altera-struct-pack-align* check for empty structs.

- Fixed a false positive in *bugprone-branch-clone* when the branches involve unknown expressions.

- Fixed some false positives in *bugprone-infinite-loop* involving dependent expressions.

- Fixed a crash in *bugprone-sizeof-expression* when *sizeof(...)* is compared against a *__int128_t*.

- Fixed bugs in *bugprone-use-after-move*:

  - Treat a move in a lambda capture as happening in the function that defines the lambda, not within the body of the lambda (as we were previously doing erroneously).

  - Don’t emit an erroneous warning on self-moves.

- Improved *cert-dcl58-cpp* check.

  The check now detects explicit template specializations that are handled specially.

- Made *cert-oop57-cpp* more sensitive by checking for an arbitrary expression in the second argument of **memset**.

- Made the fix-it of *cppcoreguidelines-init-variables* use **false** to initialize boolean variables.

- Improved *cppcoreguidelines-prefer-member-initializer* check.

  Fixed an issue when there was already an initializer in the constructor and the check would try to create another initializer for the same member.

- Fixed a false positive in *cppcoreguidelines-virtual-class-destructor* involving **final** classes. The check will not diagnose classes marked **final**, since those cannot be used as base classes, consequently, they can not violate the rule.

- Fixed a crash in *llvmlibc-callee-namespace* when executing for C++ code that contain calls to advanced constructs, e.g. overloaded operators.

- Fixed false positives in *misc-redundant-expression*:

  - Fixed a false positive involving overloaded comparison operators.

  - Fixed a false positive involving assignments in conditions. This fixes *Issue 35853*.

- Fixed a false positive in *misc-unused-parameters* where invalid parameters were implicitly being treated as being unused. This fixes *Issue 56152*.

- Fixed false positives in *misc-unused-using-decls* where *using* statements bringing operators into the scope where incorrectly marked as unused. This fixes *issue 55095*.

- Fixed a false positive in *modernize-deprecated-headers* involving including C header files from C++ files wrapped by **extern "C" { ... }** blocks. Such includes will be ignored by now. By default now it doesn't warn for including deprecated headers from header files, since that header file might be used from C source files. By passing the **CheckHeaderFile=true** option if header files of the project only included by C++ source files.

- Improved *performance-inefficient-vector-operation* to work when the vector is a member of a structure.

- Fixed a crash in *performance-unnecessary-value-param* when the specialization template has an unnecessary value parameter. Removed the fix for a template.

- Fixed a crash in *readability-const-return-type* when a pure virtual function overrided has a const return type. Removed the fix for a virtual function.

- Skipped addition of extra parentheses around member accesses (**a.b**) in fix-it for *readability-container-data-pointer*.

- Fixed incorrect suggestions for *readability-container-size-empty* when smart pointers are involved.

- Fixed a false positive in *readability-non-const-parameter* when the parameter is referenced by an lvalue.

- Expanded *readability-simplify-boolean-expr* to simplify expressions using DeMorgan's Theorem.

Removed checks

Improvements to include-fixer

The improvements are...

Improvements to clang-include-fixer

The improvements are...

Improvements to modularize

The improvements are...

Improvements to pp-trace

- Added *HashLoc* information to *InclusionDirective* callback output.

Clang-tidy Visual Studio plugin

CLANG-TIDY

Contents

- *Clang-Tidy*
  - *Using clang-tidy*
  - *Suppressing Undesired Diagnostics*

See also:

Clang-Tidy Checks

abseil-cleanup-ctad

Suggests switching the initialization pattern of **absl::Cleanup** instances from the factory function to class template argument deduction (CTAD), in C++17 and higher.

```
auto c1 = absl::MakeCleanup([] {});

const auto c2 = absl::MakeCleanup(std::function<void()>([] {}));
```

becomes

```
absl::Cleanup c1 = [] {};

const absl::Cleanup c2 = std::function<void()>([] {});
```

abseil-duration-addition

Check for cases where addition should be performed in the **absl::Time** domain. When adding two values, and one is known to be an **absl::Time**, we can infer that the other should be interpreted as an **absl::Duration** of a similar scale, and make that inference explicit.

Examples:

```
// Original - Addition in the integer domain
int x;
absl::Time t;
int result = absl::ToUnixSeconds(t) + x;

// Suggestion - Addition in the absl::Time domain
int result = absl::ToUnixSeconds(t + absl::Seconds(x));
```

abseil-duration-comparison

Checks for comparisons which should be in the **absl::Duration** domain instead of the floating point or integer domains.

N.B.: In cases where a **Duration** was being converted to an integer and then compared against a floating-point value, truncation during the **Duration** conversion might yield a different result. In practice this is very rare, and still indicates a bug which should be fixed.

Examples:

```
// Original - Comparison in the floating point domain
double x;
absl::Duration d;
if (x < absl::ToDoubleSeconds(d)) ...

// Suggested - Compare in the absl::Duration domain instead
if (absl::Seconds(x) < d) ...


// Original - Comparison in the integer domain
int x;
absl::Duration d;
if (x < absl::ToInt64Microseconds(d)) ...

// Suggested - Compare in the absl::Duration domain instead
if (absl::Microseconds(x) < d) ...
```

abseil-duration-conversion-cast

Checks for casts of **absl::Duration** conversion functions, and recommends the right conversion function instead.

Examples:

```
// Original - Cast from a double to an integer
absl::Duration d;
int i = static_cast<int>(absl::ToDoubleSeconds(d));

// Suggested - Use the integer conversion function directly.
int i = absl::ToInt64Seconds(d);


// Original - Cast from a double to an integer
absl::Duration d;
double x = static_cast<double>(absl::ToInt64Seconds(d));

// Suggested - Use the integer conversion function directly.
double x = absl::ToDoubleSeconds(d);
```

Note: In the second example, the suggested fix could yield a different result, as the conversion to integer could truncate. In practice, this is very rare, and you should use **absl::Trunc** to perform

this operation explicitly instead.

abseil-duration-division

absl::Duration arithmetic works like it does with integers. That means that division of two **absl::Duration** objects returns an **int64** with any fractional component truncated toward 0. See *this link* for more information on arithmetic with **absl::Duration**.

For example:

```
absl::Duration d = absl::Seconds(3.5);
int64 sec1 = d / absl::Seconds(1);     // Truncates toward 0.
int64 sec2 = absl::ToInt64Seconds(d);  // Equivalent to division.
assert(sec1 == 3 && sec2 == 3);

double dsec = d / absl::Seconds(1);  // WRONG: Still truncates toward 0.
assert(dsec == 3.0);
```

If you want floating-point division, you should use either the **absl::FDivDuration()** function, or one of the unit conversion functions such as **absl::ToDoubleSeconds()**. For example:

```
absl::Duration d = absl::Seconds(3.5);
double dsec1 = absl::FDivDuration(d, absl::Seconds(1));  // GOOD: No truncation.
double dsec2 = absl::ToDoubleSeconds(d);                 // GOOD: No truncation.
assert(dsec1 == 3.5 && dsec2 == 3.5);
```

This check looks for uses of **absl::Duration** division that is done in a floating-point context, and recommends the use of a function that returns a floating-point value.

abseil-duration-factory-float

Checks for cases where the floating-point overloads of various **absl::Duration** factory functions are called when the more-efficient integer versions could be used instead.

This check will not suggest fixes for literals which contain fractional floating point values or non-literals. It will suggest removing superfluous casts.

Examples:

```
// Original - Providing a floating-point literal.
absl::Duration d = absl::Seconds(10.0);
```

```
// Suggested - Use an integer instead.
absl::Duration d = absl::Seconds(10);


// Original - Explicitly casting to a floating-point type.
absl::Duration d = absl::Seconds(static_cast<double>(10));

// Suggested - Remove the explicit cast
absl::Duration d = absl::Seconds(10);
```

abseil-duration-factory-scale

Checks for cases where arguments to **absl::Duration** factory functions are scaled internally and could be changed to a different factory function. This check also looks for arguments with a zero value and suggests using **absl::ZeroDuration()** instead.

Examples:

```
// Original - Internal multiplication.
int x;
absl::Duration d = absl::Seconds(60 * x);

// Suggested - Use absl::Minutes instead.
absl::Duration d = absl::Minutes(x);


// Original - Internal division.
int y;
absl::Duration d = absl::Milliseconds(y / 1000.);

// Suggested - Use absl:::Seconds instead.
absl::Duration d = absl::Seconds(y);


// Original - Zero-value argument.
absl::Duration d = absl::Hours(0);

// Suggested = Use absl::ZeroDuration instead
absl::Duration d = absl::ZeroDuration();
```

abseil-duration-subtraction

Checks for cases where subtraction should be performed in the **absl::Duration** domain. When subtracting two values, and the first one is known to be a conversion from **absl::Duration**, we can infer that the second should also be interpreted as an **absl::Duration**, and make that inference explicit.

Examples:

```
// Original - Subtraction in the double domain
double x;
absl::Duration d;
double result = absl::ToDoubleSeconds(d) - x;

// Suggestion - Subtraction in the absl::Duration domain instead
double result = absl::ToDoubleSeconds(d - absl::Seconds(x));

// Original - Subtraction of two Durations in the double domain
absl::Duration d1, d2;
double result = absl::ToDoubleSeconds(d1) - absl::ToDoubleSeconds(d2);

// Suggestion - Subtraction in the absl::Duration domain instead
double result = absl::ToDoubleSeconds(d1 - d2);
```

Note: As with other **clang-tidy** checks, it is possible that multiple fixes may overlap (as in the case of nested expressions), so not all occurrences can be transformed in one run. In particular, this may occur for nested subtraction expressions. Running **clang-tidy** multiple times will find and fix these overlaps.

abseil-duration-unnecessary-conversion

Finds and fixes cases where **absl::Duration** values are being converted to numeric types and back again.

Floating-point examples:

```
// Original - Conversion to double and back again
absl::Duration d1;
absl::Duration d2 = absl::Seconds(absl::ToDoubleSeconds(d1));

// Suggestion - Remove unnecessary conversions
absl::Duration d2 = d1;

// Original - Division to convert to double and back again
absl::Duration d2 = absl::Seconds(absl::FDivDuration(d1, absl::Seconds(1)));
```

```
// Suggestion - Remove division and conversion
absl::Duration d2 = d1;
```

Integer examples:

```
// Original - Conversion to integer and back again
absl::Duration d1;
absl::Duration d2 = absl::Hours(absl::ToInt64Hours(d1));

// Suggestion - Remove unnecessary conversions
absl::Duration d2 = d1;

// Original - Integer division followed by conversion
absl::Duration d2 = absl::Seconds(d1 / absl::Seconds(1));

// Suggestion - Remove division and conversion
absl::Duration d2 = d1;
```

Unwrapping scalar operations:

```
// Original - Multiplication by a scalar
absl::Duration d1;
absl::Duration d2 = absl::Seconds(absl::ToInt64Seconds(d1) * 2);

// Suggestion - Remove unnecessary conversion
absl::Duration d2 = d1 * 2;
```

Note: Converting to an integer and back to an **absl::Duration** might be a truncating operation if the value is not aligned to the scale of conversion. In the rare case where this is the intended result, callers should use **absl::Trunc** to truncate explicitly.

abseil-faster-strsplit-delimiter

Finds instances of **absl::StrSplit()** or **absl::MaxSplits()** where the delimiter is a single character string literal and replaces with a character. The check will offer a suggestion to change the string literal into a character. It will also catch code using **absl::ByAnyChar()** for just a single character and will transform that into a single character as well.

These changes will give the same result, but using characters rather than single character string literals is more efficient and readable.

Examples:

```
// Original - the argument is a string literal.
for (auto piece : absl::StrSplit(str, "B")) {

// Suggested - the argument is a character, which causes the more efficient
// overload of absl::StrSplit() to be used.
for (auto piece : absl::StrSplit(str, 'B')) {


// Original - the argument is a string literal inside absl::ByAnyChar call.
for (auto piece : absl::StrSplit(str, absl::ByAnyChar("B"))) {

// Suggested - the argument is a character, which causes the more efficient
// overload of absl::StrSplit() to be used and we do not need absl::ByAnyChar
// anymore.
for (auto piece : absl::StrSplit(str, 'B')) {


// Original - the argument is a string literal inside absl::MaxSplits call.
for (auto piece : absl::StrSplit(str, absl::MaxSplits("B", 1))) {

// Suggested - the argument is a character, which causes the more efficient
// overload of absl::StrSplit() to be used.
for (auto piece : absl::StrSplit(str, absl::MaxSplits('B', 1))) {
```

abseil-no-internal-dependencies

Warns if code using Abseil depends on internal details. If something is in a namespace that includes the word "internal", code is not allowed to depend upon it because it's an implementation detail. They cannot friend it, include it, you mention it or refer to it in any way. Doing so violates Abseil's compatibility guidelines and may result in breakage. See *https://abseil.io/about/compatibility* for more information.

The following cases will result in warnings:

```
absl::strings_internal::foo();
// warning triggered on this line
class foo {
  friend struct absl::container_internal::faa;
  // warning triggered on this line
```

```
};
absl::memory_internal::MakeUniqueResult();
// warning triggered on this line
```

abseil-no-namespace

Ensures code does not open **namespace absl** as that violates Abseil's compatibility guidelines. Code should not open **namespace absl** as that conflicts with Abseil's compatibility guidelines and may result in breakage.

Any code that uses:

```
namespace absl {
 ...
}
```

will be prompted with a warning.

See *the full Abseil compatibility guidelines* for more information.

abseil-redundant-strcat-calls

Suggests removal of unnecessary calls to **absl::StrCat** when the result is being passed to another call to **absl::StrCat** or **absl::StrAppend**.

The extra calls cause unnecessary temporary strings to be constructed. Removing them makes the code smaller and faster.

Examples:

```
std::string s = absl::StrCat("A", absl::StrCat("B", absl::StrCat("C", "D")));
//before

std::string s = absl::StrCat("A", "B", "C", "D");
//after

absl::StrAppend(&s, absl::StrCat("E", "F", "G"));
//before

absl::StrAppend(&s, "E", "F", "G");
//after
```

abseil-str-cat-append

Flags uses of **absl::StrCat()** to append to a **std::string**. Suggests **absl::StrAppend()** should be used instead.

The extra calls cause unnecessary temporary strings to be constructed. Removing them makes the code smaller and faster.

```
a = absl::StrCat(a, b); // Use absl::StrAppend(&a, b) instead.
```

Does not diagnose cases where **absl::StrCat()** is used as a template argument for a functor.

abseil-string-find-startswith

Checks whether a **std::string::find()** or **std::string::rfind()** result is compared with 0, and suggests replacing with **absl::StartsWith()**. This is both a readability and performance issue.

```
string s = "...";
if (s.find("Hello World") == 0) { /* do something */ }
if (s.rfind("Hello World", 0) == 0) { /* do something */ }
```

becomes

```
string s = "...";
if (absl::StartsWith(s, "Hello World")) { /* do something */ }
if (absl::StartsWith(s, "Hello World")) { /* do something */ }
```

Options

StringLikeClasses

Semicolon-separated list of names of string-like classes. By default only **std::basic_string** is considered. The list of methods to considered is fixed.

IncludeStyle

A string specifying which include-style is used, *llvm* or *google*. Default is *llvm*.

AbseilStringsMatchHeader

The location of Abseil's **strings/match.h**. Defaults to **absl/strings/match.h**.

abseil-string-find-str-contains

Finds **s.find(...) == string::npos** comparisons (for various string-like types) and suggests replacing with **absl::StrContains()**.

This improves readability and reduces the likelihood of accidentally mixing **find()** and **npos** from different string-like types.

By default, "string-like types" includes **::std::basic_string**, **::std::basic_string_view**, and **::absl::string_view**. See the StringLikeClasses option to change this.

```
std::string s = "...";
if (s.find("Hello World") == std::string::npos) { /* do something */ }

absl::string_view a = "...";
if (absl::string_view::npos != a.find("Hello World")) { /* do something */ }
```

becomes

```
std::string s = "...";
if (!absl::StrContains(s, "Hello World")) { /* do something */ }

absl::string_view a = "...";
if (absl::StrContains(a, "Hello World")) { /* do something */ }
```

Options

StringLikeClasses
Semicolon-separated list of names of string-like classes. By default includes **::std::basic_string**, **::std::basic_string_view**, and **::absl::string_view**.

IncludeStyle
A string specifying which include-style is used, *llvm* or *google*. Default is *llvm*.

AbseilStringsMatchHeader
The location of Abseil's **strings/match.h**. Defaults to **absl/strings/match.h**.

abseil-time-comparison

Prefer comparisons in the **absl::Time** domain instead of the integer domain.

N.B.: In cases where an **absl::Time** is being converted to an integer, alignment may occur. If the comparison depends on this alignment, doing the comparison in the **absl::Time** domain may yield a different result. In practice this is very rare, and still indicates a bug which should be fixed.

Examples:

```
// Original - Comparison in the integer domain
int x;
absl::Time t;
if (x < absl::ToUnixSeconds(t)) ...

// Suggested - Compare in the absl::Time domain instead
if (absl::FromUnixSeconds(x) < t) ...
```

abseil-time-subtraction

Finds and fixes **absl::Time** subtraction expressions to do subtraction in the Time domain instead of the numeric domain.

There are two cases of Time subtraction in which deduce additional type information:

- When the result is an **absl::Duration** and the first argument is an **absl::Time**.
- When the second argument is a **absl::Time**.

In the first case, we must know the result of the operation, since without that the second operand could be either an **absl::Time** or an **absl::Duration**. In the second case, the first operand *must* be an **absl::Time**, because subtracting an **absl::Time** from an **absl::Duration** is not defined.

Examples:

```
int x;
absl::Time t;

// Original - absl::Duration result and first operand is an absl::Time.
absl::Duration d = absl::Seconds(absl::ToUnixSeconds(t) - x);

// Suggestion - Perform subtraction in the Time domain instead.
absl::Duration d = t - absl::FromUnixSeconds(x);


// Original - Second operand is an absl::Time.
int i = x - absl::ToUnixSeconds(t);

// Suggestion - Perform subtraction in the Time domain instead.
int i = absl::ToInt64Seconds(absl::FromUnixSeconds(x) - t);
```

abseil-upgrade-duration-conversions

Finds calls to **absl::Duration** arithmetic operators and factories whose argument needs an explicit cast to continue compiling after upcoming API changes.

The operators ***=**, **/=**, *****, and **/** for **absl::Duration** currently accept an argument of class type that is convertible to an arithmetic type. Such a call currently converts the value to an **int64_t**, even in a case such as **std::atomic<float>** that would result in lossy conversion.

Additionally, the **absl::Duration** factory functions (**absl::Hours**, **absl::Minutes**, etc) currently accept an **int64_t** or a floating-point type. Similar to the arithmetic operators, calls with an argument of class type that is convertible to an arithmetic type go through the **int64_t** path.

These operators and factories will be changed to only accept arithmetic types to prevent unintended behavior. After these changes are released, passing an argument of class type will no longer compile, even if the type is implicitly convertible to an arithmetic type.

Here are example fixes created by this check:

```
std::atomic<int> a;
absl::Duration d = absl::Milliseconds(a);
d *= a;
```

becomes

```
std::atomic<int> a;
absl::Duration d = absl::Milliseconds(static_cast<int64_t>(a));
d *= static_cast<int64_t>(a);
```

Note that this check always adds a cast to **int64_t** in order to preserve the current behavior of user code. It is possible that this uncovers unintended behavior due to types implicitly convertible to a floating-point type.

altera-id-dependent-backward-branch

Finds ID-dependent variables and fields that are used within loops. This causes branches to occur inside the loops, and thus leads to performance degradation.

```
// The following code will produce a warning because this ID-dependent
// variable is used in a loop condition statement.
int ThreadID = get_local_id(0);
```

```
// The following loop will produce a warning because the loop condition
// statement depends on an ID-dependent variable.
for (int i = 0; i < ThreadID; ++i) {
  std::cout << i << std::endl;
}

// The following loop will not produce a warning, because the ID-dependent
// variable is not used in the loop condition statement.
for (int i = 0; i < 100; ++i) {
  std::cout << ThreadID << std::endl;
}
```

Based on the *Altera SDK for OpenCL: Best Practices Guide*.

altera-kernel-name-restriction

Finds kernel files and include directives whose filename is *kernel.cl*, *Verilog.cl*, or *VHDL.cl*. The check is case insensitive.

Such kernel file names cause the offline compiler to generate intermediate design files that have the same names as certain internal files, which leads to a compilation error.

Based on the *Guidelines for Naming the Kernel* section in the *Intel FPGA SDK for OpenCL Pro Edition: Programming Guide*.

altera-single-work-item-barrier

Finds OpenCL kernel functions that call a barrier function but do not call an ID function (**get_local_id**, **get_local_id**, **get_group_id**, or **get_local_linear_id**).

These kernels may be viable single work-item kernels, but will be forced to execute as NDRange kernels if using a newer version of the Altera Offline Compiler (>= v17.01).

If using an older version of the Altera Offline Compiler, these kernel functions will be treated as single work-item kernels, which could be inefficient or lead to errors if NDRange semantics were intended.

Based on the *Altera SDK for OpenCL: Best Practices Guide*.

Examples:

```
// error: function calls barrier but does not call an ID function.
void __kernel barrier_no_id(__global int * foo, int size) {
```

```
  for (int i = 0; i < 100; i++) {
    foo[i] += 5;
  }
  barrier(CLK_GLOBAL_MEM_FENCE);
}

// ok: function calls barrier and an ID function.
void __kernel barrier_with_id(__global int * foo, int size) {
  for (int i = 0; i < 100; i++) {
    int tid = get_global_id(0);
    foo[tid] += 5;
  }
  barrier(CLK_GLOBAL_MEM_FENCE);
}

// ok with AOC Version 17.01: the reqd_work_group_size turns this into
// an NDRange.
__attribute__((reqd_work_group_size(2,2,2)))
void __kernel barrier_with_id(__global int * foo, int size) {
  for (int i = 0; i < 100; i++) {
    foo[tid] += 5;
  }
  barrier(CLK_GLOBAL_MEM_FENCE);
}
```

Options

AOCVersion

Defines the version of the Altera Offline Compiler. Defaults to **1600** (corresponding to version 16.00).

altera-struct-pack-align

Finds structs that are inefficiently packed or aligned, and recommends packing and/or aligning of said structs as needed.

Structs that are not packed take up more space than they should, and accessing structs that are not well aligned is inefficient.

Fix-its are provided to fix both of these issues by inserting and/or amending relevant struct attributes.

Based on the *Altera SDK for OpenCL: Best Practices Guide*.

```
// The following struct is originally aligned to 4 bytes, and thus takes up
// 12 bytes of memory instead of 10. Packing the struct will make it use
// only 10 bytes of memory, and aligning it to 16 bytes will make it
// efficient to access.
struct example {
  char a;    // 1 byte
  double b;  // 8 bytes
  char c;    // 1 byte
};

// The following struct is arranged in such a way that packing is not needed.
// However, it is aligned to 4 bytes instead of 8, and thus needs to be
// explicitly aligned.
struct implicitly_packed_example {
  char a;  // 1 byte
  char b;  // 1 byte
  char c;  // 1 byte
  char d;  // 1 byte
  int e;   // 4 bytes
};

// The following struct is explicitly aligned and packed.
struct good_example {
  char a;    // 1 byte
  double b;  // 8 bytes
  char c;    // 1 byte
} __attribute__((packed)) __attribute__((aligned(16));

// Explicitly aligning a struct to the wrong value will result in a warning.
// The following example should be aligned to 16 bytes, not 32.
struct badly_aligned_example {
  char a;    // 1 byte
  double b;  // 8 bytes
  char c;    // 1 byte
} __attribute__((packed)) __attribute__((aligned(32)));
```

altera-unroll-loops

Finds inner loops that have not been unrolled, as well as fully unrolled loops with unknown loop

bounds or a large number of iterations.

Unrolling inner loops could improve the performance of OpenCL kernels. However, if they have unknown loop bounds or a large number of iterations, they cannot be fully unrolled, and should be partially unrolled.

Notes:

- This check is unable to determine the number of iterations in a **while** or **do..while** loop; hence if such a loop is fully unrolled, a note is emitted advising the user to partially unroll instead.

- In **for** loops, our check only works with simple arithmetic increments (**+**, **-**, *****, /). For all other increments, partial unrolling is advised.

- Depending on the exit condition, the calculations for determining if the number of iterations is large may be off by 1. This should not be an issue since the cut-off is generally arbitrary.

Based on the *Altera SDK for OpenCL: Best Practices Guide*.

```
for (int i = 0; i < 10; i++) {  // ok: outer loops should not be unrolled
  int j = 0;
  do {  // warning: this inner do..while loop should be unrolled
    j++;
  } while (j < 15);

  int k = 0;
  #pragma unroll
  while (k < 20) {  // ok: this inner loop is already unrolled
    k++;
  }
}

int A[1000];
#pragma unroll
// warning: this loop is large and should be partially unrolled
for (int a : A) {
  printf("%d", a);
}

#pragma unroll 5
```

```
// ok: this loop is large, but is partially unrolled
for (int a : A) {
  printf("%d", a);
}

#pragma unroll
// warning: this loop is large and should be partially unrolled
for (int i = 0; i < 1000; ++i) {
  printf("%d", i);
}

#pragma unroll 5
// ok: this loop is large, but is partially unrolled
for (int i = 0; i < 1000; ++i) {
  printf("%d", i);
}

#pragma unroll
// warning: << operator not supported, recommend partial unrolling
for (int i = 0; i < 1000; i<<1) {
  printf("%d", i);
}

std::vector<int> someVector (100, 0);
int i = 0;
#pragma unroll
// note: loop may be large, recommend partial unrolling
while (i < someVector.size()) {
  someVector[i]++;
}

#pragma unroll
// note: loop may be large, recommend partial unrolling
while (true) {
  printf("In loop");
}

#pragma unroll 5
// ok: loop may be large, but is partially unrolled
while (i < someVector.size()) {
```

```
  someVector[i]++;
}
```

Options

MaxLoopIterations
Defines the maximum number of loop iterations that a fully unrolled loop can have. By default, it is set to *100*.

In practice, this refers to the integer value of the upper bound within the loop statement's condition expression.

android-cloexec-accept
The usage of **accept()** is not recommended, it's better to use **accept4()**. Without this flag, an opened sensitive file descriptor would remain open across a fork+exec to a lower-privileged SELinux domain.

Examples:

```
accept(sockfd, addr, addrlen);

// becomes

accept4(sockfd, addr, addrlen, SOCK_CLOEXEC);
```

android-cloexec-accept4
accept4() should include **SOCK_CLOEXEC** in its type argument to avoid the file descriptor leakage. Without this flag, an opened sensitive file would remain open across a fork+exec to a lower-privileged SELinux domain.

Examples:

```
accept4(sockfd, addr, addrlen, SOCK_NONBLOCK);

// becomes

accept4(sockfd, addr, addrlen, SOCK_NONBLOCK | SOCK_CLOEXEC);
```

android-cloexec-creat
The usage of **creat()** is not recommended, it's better to use **open()**.

Examples:

```
int fd = creat(path, mode);

// becomes

int fd = open(path, O_WRONLY | O_CREAT | O_TRUNC | O_CLOEXEC, mode);
```

android-cloexec-dup

The usage of **dup()** is not recommended, it's better to use **fcntl()**, which can set the close-on-exec flag. Otherwise, an opened sensitive file would remain open across a fork+exec to a lower-privileged SELinux domain.

Examples:

```
int fd = dup(oldfd);

// becomes

int fd = fcntl(oldfd, F_DUPFD_CLOEXEC);
```

android-cloexec-epoll-create

The usage of **epoll_create()** is not recommended, it's better to use **epoll_create1()**, which allows close-on-exec.

Examples:

```
epoll_create(size);

// becomes

epoll_create1(EPOLL_CLOEXEC);
```

android-cloexec-epoll-create1

epoll_create1() should include **EPOLL_CLOEXEC** in its type argument to avoid the file descriptor leakage. Without this flag, an opened sensitive file would remain open across a fork+exec to a lower-privileged SELinux domain.

Examples:

```
epoll_create1(0);

// becomes

epoll_create1(EPOLL_CLOEXEC);
```

android-cloexec-fopen

fopen() should include **e** in their mode string; so **re** would be valid. This is equivalent to having set **FD_CLOEXEC on** that descriptor.

Examples:

```
fopen("fn", "r");

// becomes

fopen("fn", "re");
```

android-cloexec-inotify-init

The usage of **inotify_init()** is not recommended, it's better to use **inotify_init1()**.

Examples:

```
inotify_init();

// becomes

inotify_init1(IN_CLOEXEC);
```

android-cloexec-inotify-init1

inotify_init1() should include **IN_CLOEXEC** in its type argument to avoid the file descriptor leakage. Without this flag, an opened sensitive file would remain open across a fork+exec to a lower-privileged SELinux domain.

Examples:

```
inotify_init1(IN_NONBLOCK);

// becomes
```

```
inotify_init1(IN_NONBLOCK | IN_CLOEXEC);
```

android-cloexec-memfd-create

memfd_create() should include **MFD_CLOEXEC** in its type argument to avoid the file descriptor leakage. Without this flag, an opened sensitive file would remain open across a fork+exec to a lower-privileged SELinux domain.

Examples:

```
memfd_create(name, MFD_ALLOW_SEALING);

// becomes

memfd_create(name, MFD_ALLOW_SEALING | MFD_CLOEXEC);
```

android-cloexec-open

A common source of security bugs is code that opens a file without using the **O_CLOEXEC** flag. Without that flag, an opened sensitive file would remain open across a fork+exec to a lower-privileged SELinux domain, leaking that sensitive data. Open-like functions including **open()**, **openat()**, and **open64()** should include **O_CLOEXEC** in their flags argument.

Examples:

```
open("filename", O_RDWR);
open64("filename", O_RDWR);
openat(0, "filename", O_RDWR);

// becomes

open("filename", O_RDWR | O_CLOEXEC);
open64("filename", O_RDWR | O_CLOEXEC);
openat(0, "filename", O_RDWR | O_CLOEXEC);
```

android-cloexec-pipe

This check detects usage of **pipe()**. Using **pipe()** is not recommended, **pipe2()** is the suggested replacement. The check also adds the O_CLOEXEC flag that marks the file descriptor to be closed in child processes. Without this flag a sensitive file descriptor can be leaked to a child process, potentially into a lower-privileged SELinux domain.

Examples:

```
pipe(pipefd);
```

Suggested replacement:

```
pipe2(pipefd, O_CLOEXEC);
```

android-cloexec-pipe2

This check ensures that pipe2() is called with the O_CLOEXEC flag. The check also adds the O_CLOEXEC flag that marks the file descriptor to be closed in child processes. Without this flag a sensitive file descriptor can be leaked to a child process, potentially into a lower-privileged SELinux domain.

Examples:

```
pipe2(pipefd, O_NONBLOCK);
```

Suggested replacement:

```
pipe2(pipefd, O_NONBLOCK | O_CLOEXEC);
```

android-cloexec-socket

socket() should include **SOCK_CLOEXEC** in its type argument to avoid the file descriptor leakage. Without this flag, an opened sensitive file would remain open across a fork+exec to a lower-privileged SELinux domain.

Examples:

```
socket(domain, type, SOCK_STREAM);

// becomes

socket(domain, type, SOCK_STREAM | SOCK_CLOEXEC);
```

android-comparison-in-temp-failure-retry

Diagnoses comparisons that appear to be incorrectly placed in the argument to the **TEMP_FAILURE_RETRY** macro. Having such a use is incorrect in the vast majority of cases, and will often silently defeat the purpose of the **TEMP_FAILURE_RETRY** macro.

For context, **TEMP_FAILURE_RETRY** is *a convenience macro* provided by both glibc and Bionic. Its purpose is to repeatedly run a syscall until it either succeeds, or fails for reasons other than being

interrupted.

Example buggy usage looks like:

```
char cs[1];
while (TEMP_FAILURE_RETRY(read(STDIN_FILENO, cs, sizeof(cs)) != 0)) {
  // Do something with cs.
}
```

Because TEMP_FAILURE_RETRY will check for whether the result *of the comparison* is **-1**, and retry if so.

If you encounter this, the fix is simple: lift the comparison out of the **TEMP_FAILURE_RETRY** argument, like so:

```
char cs[1];
while (TEMP_FAILURE_RETRY(read(STDIN_FILENO, cs, sizeof(cs))) != 0) {
  // Do something with cs.
}
```

Options

RetryMacros
A comma-separated list of the names of retry macros to be checked.

boost-use-to-string

This check finds conversion from integer type like **int** to **std::string** or **std::wstring** using **boost::lexical_cast**, and replace it with calls to **std::to_string** and **std::to_wstring**.

It doesn’t replace conversion from floating points despite the **to_string** overloads, because it would change the behavior.

```
auto str = boost::lexical_cast<std::string>(42);
auto wstr = boost::lexical_cast<std::wstring>(2137LL);

// Will be changed to
auto str = std::to_string(42);
auto wstr = std::to_wstring(2137LL);
```

bugprone-argument-comment

Checks that argument comments match parameter names.

The check understands argument comments in the form **/*parameter_name=*/** that are placed right before the argument.

```
void f(bool foo);

...

f(/*bar=*/true);
// warning: argument name 'bar' in comment does not match parameter name 'foo'
```

The check tries to detect typos and suggest automated fixes for them.

Options

StrictMode
When *false* (default value), the check will ignore leading and trailing underscores and case when comparing names -- otherwise they are taken into account.

IgnoreSingleArgument
When *true*, the check will ignore the single argument.

CommentBoolLiterals
When *true*, the check will add argument comments in the format **/*ParameterName=*/** right before the boolean literal argument.

Before:

```
void foo(bool TurnKey, bool PressButton);

foo(true, false);
```

After:

```
void foo(bool TurnKey, bool PressButton);

foo(/*TurnKey=*/true, /*PressButton=*/false);
```

CommentIntegerLiterals

When true, the check will add argument comments in the format **/*ParameterName=*/** right before the integer literal argument.

Before:

```
void foo(int MeaningOfLife);

foo(42);
```

After:

```
void foo(int MeaningOfLife);

foo(/*MeaningOfLife=*/42);
```

CommentFloatLiterals

When true, the check will add argument comments in the format **/*ParameterName=*/** right before the float/double literal argument.

Before:

```
void foo(float Pi);

foo(3.14159);
```

After:

```
void foo(float Pi);

foo(/*Pi=*/3.14159);
```

CommentStringLiterals

When true, the check will add argument comments in the format **/*ParameterName=*/** right before the string literal argument.

Before:

```
void foo(const char *String);
void foo(const wchar_t *WideString);
```

```
foo("Hello World");
foo(L"Hello World");
```

After:

```
void foo(const char *String);
void foo(const wchar_t *WideString);

foo(/*String=*/"Hello World");
foo(/*WideString=*/L"Hello World");
```

CommentCharacterLiterals

When true, the check will add argument comments in the format **/*ParameterName=*/** right before the character literal argument.

Before:

```
void foo(char *Character);

foo('A');
```

After:

```
void foo(char *Character);

foo(/*Character=*/'A');
```

CommentUserDefinedLiterals

When true, the check will add argument comments in the format **/*ParameterName=*/** right before the user defined literal argument.

Before:

```
void foo(double Distance);

double operator"" _km(long double);

foo(402.0_km);
```

After:

```
void foo(double Distance);

double operator"" _km(long double);

foo(/*Distance=*/402.0_km);
```

CommentNullPtrs
When true, the check will add argument comments in the format **/*ParameterName=*/** right before the nullptr literal argument.

Before:

```
void foo(A* Value);

foo(nullptr);
```

After:

```
void foo(A* Value);

foo(/*Value=*/nullptr);
```

bugprone-assert-side-effect
Finds **assert()** with side effect.

The condition of **assert()** is evaluated only in debug builds so a condition with side effect can cause different behavior in debug / release builds.

Options

AssertMacros
A comma-separated list of the names of assert macros to be checked.

CheckFunctionCalls
Whether to treat non-const member and non-member functions as they produce side effects. Disabled by default because it can increase the number of false positive warnings.

IgnoredFunctions
A semicolon-separated list of the names of functions or methods to be considered as not having side-effects. Regular expressions are accepted, e.g. *[Rr]ef(erence)?$* matches every type with

suffix *Ref*, *ref*, *Reference* and *reference*. The default is empty. If a name in the list contains the sequence *::* it is matched against the qualified typename (i.e. *namespace::Type*, otherwise it is matched against only the type name (i.e. *Type*).

bugprone-assignment-in-if-condition

Finds assignments within conditions of *if* statements. Such assignments are bug-prone because they may have been intended as equality tests.

This check finds all assignments within *if* conditions, including ones that are not flagged by *-Wparentheses* due to an extra set of parentheses, and including assignments that call an overloaded *operator=()*. The identified assignments violate *BARR group "Rule 8.2.c"*.

```
int f = 3;
if(f = 4) { // This is identified by both ‘Wparentheses‘ and this check - should it have been: ‘if (f == 4)‘ ?
  f = f + 1;
}

if((f == 5) || (f = 6)) { // the assignment here ‘(f = 6)‘ is identified by this check, but not by ‘-Wparentheses‘. Shoul
  f = f + 2;
}
```

bugprone-bad-signal-to-kill-thread

Finds **pthread_kill** function calls when a thread is terminated by raising **SIGTERM** signal and the signal kills the entire process, not just the individual thread. Use any signal except **SIGTERM**.

This check corresponds to the CERT C Coding Standard rule *POS44-C. Do not use signals to terminate threads*.

bugprone-bool-pointer-implicit-conversion

Checks for conditions based on implicit conversion from a **bool** pointer to **bool**.

Example:

```
bool *p;
if (p) {
  // Never used in a pointer-specific way.
}
```

bugprone-branch-clone

Checks for repeated branches in **if/else if/else** chains, consecutive repeated branches in **switch**

statements and identical true and false branches in conditional operators.

```
if (test_value(x)) {
  y++;
  do_something(x, y);
} else {
  y++;
  do_something(x, y);
}
```

In this simple example (which could arise e.g. as a copy-paste error) the **then** and **else** branches are identical and the code is equivalent the following shorter and cleaner code:

```
test_value(x); // can be omitted unless it has side effects
y++;
do_something(x, y);
```

If this is the intended behavior, then there is no reason to use a conditional statement; otherwise the issue can be solved by fixing the branch that is handled incorrectly.

The check also detects repeated branches in longer **if/else if/else** chains where it would be even harder to notice the problem.

In **switch** statements the check only reports repeated branches when they are consecutive, because it is relatively common that the **case:** labels have some natural ordering and rearranging them would decrease the readability of the code. For example:

```
switch (ch) {
case 'a':
  return 10;
case 'A':
  return 10;
case 'b':
  return 11;
case 'B':
  return 11;
default:
  return 10;
}
```

Here the check reports that the **’a’** and **’A’** branches are identical (and that the **’b’** and **’B’** branches are also identical), but does not report that the **default:** branch is also identical to the first two branches. If this is indeed the correct behavior, then it could be implemented as:

```
switch (ch) {
case ’a’:
case ’A’:
  return 10;
case ’b’:
case ’B’:
  return 11;
default:
  return 10;
}
```

Here the check does not warn for the repeated **return 10;**, which is good if we want to preserve that **’a’** is before **’b’** and **default:** is the last branch.

Finally, the check also examines conditional operators and reports code like:

```
return test_value(x) ? x : x;
```

Unlike if statements, the check does not detect chains of conditional operators.

Note: This check also reports situations where branches become identical only after preprocessing.

bugprone-copy-constructor-init

Finds copy constructors where the constructor doesn’t call the copy constructor of the base class.

```
class Copyable {
public:
  Copyable() = default;
  Copyable(const Copyable &) = default;
};
class X2 : public Copyable {
  X2(const X2 &other) {} // Copyable(other) is missing
};
```

Also finds copy constructors where the constructor of the base class don’t have parameter.

```
class X4 : public Copyable {
  X4(const X4 &other) : Copyable() {} // other is missing
};
```

The check also suggests a fix-its in some cases.

bugprone-dangling-handle

Detect dangling references in value handles like **std::string_view**. These dangling references can be a result of constructing handles from temporary values, where the temporary is destroyed soon after the handle is created.

Examples:

```
string_view View = string();  // View will dangle.
string A;
View = A + "A";  // still dangle.

vector<string_view> V;
V.push_back(string());  // V[0] is dangling.
V.resize(3, string());  // V[1] and V[2] will also dangle.

string_view f() {
  // All these return values will dangle.
  return string();
  string S;
  return S;
  char Array[10]{};
  return Array;
}
```

Options

HandleClasses

A semicolon-separated list of class names that should be treated as handles. By default only **std::basic_string_view** and **std::experimental::basic_string_view** are considered.

bugprone-dynamic-static-initializers

Finds instances of static variables that are dynamically initialized in header files.

This can pose problems in certain multithreaded contexts. For example, when disabling compiler

generated synchronization instructions for static variables initialized at runtime (e.g. by **-fno-threadsafe-statics**), even if a particular project takes the necessary precautions to prevent race conditions during initialization by providing their own synchronization, header files included from other projects may not. Therefore, such a check is helpful for ensuring that disabling compiler generated synchronization for static variable initialization will not cause problems.

Consider the following code:

```
int foo() {
  static int k = bar();
  return k;
}
```

When synchronization of static initialization is disabled, if two threads both call *foo* for the first time, there is the possibility that *k* will be double initialized, creating a race condition.

bugprone-easily-swappable-parameters

Finds function definitions where parameters of convertible types follow each other directly, making call sites prone to calling the function with swapped (or badly ordered) arguments.

```
void drawPoint(int X, int Y) { /* ... */ }
FILE *open(const char *Dir, const char *Name, Flags Mode) { /* ... */ }
```

A potential call like **drawPoint(-2, 5)** or **openPath("a.txt", "tmp", Read)** is perfectly legal from the language's perspective, but might not be what the developer of the function intended.

More elaborate and type-safe constructs, such as opaque typedefs or strong types should be used instead, to prevent a mistaken order of arguments.

```
struct Coord2D { int X; int Y; };
void drawPoint(const Coord2D Pos) { /* ... */ }

FILE *open(const Path &Dir, const Filename &Name, Flags Mode) { /* ... */ }
```

Due to the potentially elaborate refactoring and API-breaking that is necessary to strengthen the type safety of a project, no automatic fix-its are offered.

Options

Extension/relaxation options

Relaxation (or extension) options can be used to broaden the scope of the analysis and fine-tune the

enabling of more mixes between types. Some mixes may depend on coding style or preference specific to a project, however, it should be noted that enabling *all* of these relaxations model the way of mixing at call sites the most. These options are expected to make the check report for more functions, and report longer mixable ranges.

QualifiersMix

Whether to consider parameters of some *cvr-qualified* **T** and a differently *cvr-qualified* **T** (i.e. **T** and **const T**, **const T** and **volatile T**, etc.) mixable between one another. If *false*, the check will consider differently qualified types unmixable. *True* turns the warnings on. Defaults to *false*.

The following example produces a diagnostic only if *QualifiersMix* is enabled:

```
void *memcpy(const void *Destination, void *Source, std::size_t N) { /* ... */ }
```

ModelImplicitConversions

Whether to consider parameters of type **T** and **U** mixable if there exists an implicit conversion from **T** to **U** and **U** to **T**. If *false*, the check will not consider implicitly convertible types for mixability. *True* turns warnings for implicit conversions on. Defaults to *true*.

The following examples produce a diagnostic only if *ModelImplicitConversions* is enabled:

```
void fun(int Int, double Double) { /* ... */ }
void compare(const char *CharBuf, std::string String) { /* ... */ }
```

NOTE:

Changing the qualifiers of an expression's type (e.g. from **int** to **const int**) is defined as an *implicit conversion* in the C++ Standard. However, the check separates this decision-making on the mixability of differently qualified types based on whether *QualifiersMix* was enabled.

For example, the following code snippet will only produce a diagnostic if **both** *QualifiersMix* and *ModelImplicitConversions* are enabled:

```
void fun2(int Int, const double Double) { /* ... */ }
```

Filtering options

Filtering options can be used to lessen the size of the diagnostics emitted by the checker, whether the aim is to ignore certain constructs or dampen the noisiness.

MinimumLength

The minimum length required from an adjacent parameter sequence to be diagnosed. Defaults to *2*. Might be any positive integer greater or equal to *2*. If *0* or *1* is given, the default value *2* will be used instead.

For example, if *3* is specified, the examples above will not be matched.

IgnoredParameterNames

The list of parameter **names** that should never be considered part of a swappable adjacent parameter sequence. The value is a *;*-separated list of names. To ignore unnamed parameters, add *""* to the list verbatim (not the empty string, but the two quotes, potentially escaped!). **This option is case-sensitive!**

By default, the following parameter names, and their Uppercase-initial variants are ignored: *""* (unnamed parameters), *iterator*, *begin*, *end*, *first*, *last*, *lhs*, *rhs*.

IgnoredParameterTypeSuffixes

The list of parameter **type name suffixes** that should never be considered part of a swappable adjacent parameter sequence. Parameters which type, as written in the source code, end with an element of this option will be ignored. The value is a *;*-separated list of names. **This option is case-sensitive!**

By default, the following, and their lowercase-initial variants are ignored: *bool*, *It*, *Iterator*, *InputIt*, *ForwardIt*, *BidirIt*, *RandomIt*, *random_iterator*, *ReverseIt*, *reverse_iterator*, *reverse_const_iterator*, *RandomIt*, *random_iterator*, *ReverseIt*, *reverse_iterator*, *reverse_const_iterator*, *Const_Iterator*, *ConstIterator*, *const_reverse_iterator*, *ConstReverseIterator*. In addition, *_Bool* (but not *_bool*) is also part of the default value.

SuppressParametersUsedTogether

Suppresses diagnostics about parameters that are used together or in a similar fashion inside the function's body. Defaults to *true*. Specifying *false* will turn off the heuristics.

Currently, the following heuristics are implemented which will suppress the warning about the parameter pair involved:

- The parameters are used in the same expression, e.g. **f(a, b)** or **a < b**.

- The parameters are further passed to the same function to the same parameter of that function, of the same overload. E.g. **f(a, 1)** and **f(b, 2)** to some **f(T, int)**.

  NOTE:

The check does not perform path-sensitive analysis, and as such, "same function" in this context means the same function declaration. If the same member function of a type on two distinct instances are called with the parameters, it will still be regarded as "same function".

- The same member field is accessed, or member method is called of the two parameters, e.g. **a.foo()** and **b.foo()**.

- Separate **return** statements return either of the parameters on different code paths.

NamePrefixSuffixSilenceDissimilarityTreshold

The number of characters two parameter names might be different on *either* the head or the tail end with the rest of the name the same so that the warning about the two parameters are silenced. Defaults to *1*. Might be any positive integer. If *0*, the filtering heuristic based on the parameters' names is turned off.

This option can be used to silence warnings about parameters where the naming scheme indicates that the order of those parameters do not matter.

For example, the parameters **LHS** and **RHS** are 1-dissimilar suffixes of each other: **L** and **R** is the different character, while **HS** is the common suffix. Similarly, parameters **text1, text2, text3** are 1-dissimilar prefixes of each other, with the numbers at the end being the dissimilar part. If the value is at least *1*, such cases will not be reported.

Limitations

This check is designed to check function signatures!

The check does not investigate functions that are generated by the compiler in a context that is only determined from a call site. These cases include variadic functions, functions in C code that do not have an argument list, and C++ template instantiations. Most of these cases, which are otherwise swappable from a caller's standpoint, have no way of getting "fixed" at the definition point. In the case of C++ templates, only primary template definitions and explicit specializations are matched and analyzed.

None of the following cases produce a diagnostic:

```
int printf(const char *Format, ...) { /* ... */ }
int someOldCFunction() { /* ... */ }

template <typename T, typename U>
```

```
int add(T X, U Y) { return X + Y };

void theseAreNotWarnedAbout() {
  printf("%d %d\n", 1, 2);   // Two ints passed, they could be swapped.
  someOldCFunction(1, 2, 3); // Similarly, multiple ints passed.

  add(1, 2); // Instantiates 'add<int, int>', but that's not a user-defined function.
}
```

Due to the limitation above, parameters which type are further dependent upon template instantiations to *prove* that they mix with another parameter's is not diagnosed.

```
template <typename T>
struct Vector {
 typedef T element_type;
};

// Diagnosed: Explicit instantiation was done by the user, we can prove it
// is the same type.
void instantiated(int A, Vector<int>::element_type B) { /* ... */ }

// Diagnosed: The two parameter types are exactly the same.
template <typename T>
void exact(typename Vector<T>::element_type A,
           typename Vector<T>::element_type B) { /* ... */ }

// Skipped: The two parameters are both 'T' but we cannot prove this
// without actually instantiating.
template <typename T>
void falseNegative(T A, typename Vector<T>::element_type B) { /* ... */ }
```

In the context of *implicit conversions* (when *ModelImplicitConversions* is enabled), the modelling performed by the check warns if the parameters are swappable and the swapped order matches implicit conversions. It does not model whether there exists an unrelated third type from which *both* parameters can be given in a function call. This means that in the following example, even while **strs()** clearly carries the possibility to be called with swapped arguments (as long as the arguments are string literals), will not be warned about.

```
struct String {
  String(const char *Buf);
```

```
};

struct StringView {
  StringView(const char *Buf);
  operator const char *() const;
};

// Skipped: Directly swapping expressions of the two type cannot mix.
// (Note: StringView -> const char * -> String would be **two**
// user-defined conversions, which is disallowed by the language.)
void strs(String Str, StringView SV) { /* ... */ }

// Diagnosed: StringView implicitly converts to and from a buffer.
void cStr(StringView SV, const char *Buf() { /* ... */ }
```

bugprone-exception-escape

Finds functions which may throw an exception directly or indirectly, but they should not. The functions which should not throw exceptions are the following:

- Destructors
- Move constructors
- Move assignment operators
- The **main()** functions
- **swap()** functions
- Functions marked with **throw()** or **noexcept**
- Other functions given as option

A destructor throwing an exception may result in undefined behavior, resource leaks or unexpected termination of the program. Throwing move constructor or move assignment also may result in undefined behavior or resource leak. The **swap()** operations expected to be non throwing most of the cases and they are always possible to implement in a non throwing way. Non throwing **swap()** operations are also used to create move operations. A throwing **main()** function also results in unexpected termination.

WARNING! This check may be expensive on large source files.

Options

FunctionsThatShouldNotThrow
Comma separated list containing function names which should not throw. An example value for this parameter can be **WinMain** which adds function **WinMain()** in the Windows API to the list of the functions which should not throw. Default value is an empty string.

IgnoredExceptions
Comma separated list containing type names which are not counted as thrown exceptions in the check. Default value is an empty string.

bugprone-fold-init-type
The check flags type mismatches in *folds* like **std::accumulate** that might result in loss of precision. **std::accumulate** folds an input range into an initial value using the type of the latter, with **operator+** by default. This can cause loss of precision through:

- Truncation: The following code uses a floating point range and an int initial value, so truncation will happen at every application of **operator+** and the result will be *0*, which might not be what the user expected.

```
auto a = {0.5f, 0.5f, 0.5f, 0.5f};
return std::accumulate(std::begin(a), std::end(a), 0);
```

- Overflow: The following code also returns *0*.

```
auto a = {65536LL * 65536 * 65536};
return std::accumulate(std::begin(a), std::end(a), 0);
```

bugprone-forward-declaration-namespace
Checks if an unused forward declaration is in a wrong namespace.

The check inspects all unused forward declarations and checks if there is any declaration/definition with the same name existing, which could indicate that the forward declaration is in a potentially wrong namespace.

```
namespace na { struct A; }
namespace nb { struct A {}; }
nb::A a;
```

```
// warning : no definition found for 'A', but a definition with the same name
// 'A' found in another namespace 'nb::'
```

This check can only generate warnings, but it can't suggest a fix at this point.

bugprone-forwarding-reference-overload

The check looks for perfect forwarding constructors that can hide copy or move constructors. If a non const lvalue reference is passed to the constructor, the forwarding reference parameter will be a better match than the const reference parameter of the copy constructor, so the perfect forwarding constructor will be called, which can be confusing. For detailed description of this issue see: Scott Meyers, Effective Modern C++, Item 26.

Consider the following example:

```
class Person {
public:
  // C1: perfect forwarding ctor
  template<typename T>
  explicit Person(T&& n) {}

  // C2: perfect forwarding ctor with parameter default value
  template<typename T>
  explicit Person(T&& n, int x = 1) {}

  // C3: perfect forwarding ctor guarded with enable_if
  template<typename T, typename X = enable_if_t<is_special<T>, void>>
  explicit Person(T&& n) {}

  // C4: variadic perfect forwarding ctor guarded with enable_if
  template<typename... A,
    enable_if_t<is_constructible_v<tuple<string, int>, A&&...>, int> = 0>
  explicit Person(A&&... a) {}

  // (possibly compiler generated) copy ctor
  Person(const Person& rhs);
};
```

The check warns for constructors C1 and C2, because those can hide copy and move constructors. We suppress warnings if the copy and the move constructors are both disabled (deleted or private), because there is nothing the perfect forwarding constructor could hide in

this case. We also suppress warnings for constructors like C3 and C4 that are guarded with an **enable_if**, assuming the programmer was aware of the possible hiding.

Background

For deciding whether a constructor is guarded with enable_if, we consider the types of the constructor parameters, the default values of template type parameters and the types of non-type template parameters with a default literal value. If any part of these types is **std::enable_if** or **std::enable_if_t**, we assume the constructor is guarded.

bugprone-implicit-widening-of-multiplication-result

The check diagnoses instances where a result of a multiplication is implicitly widened, and suggests (with fix-it) to either silence the code by making widening explicit, or to perform the multiplication in a wider type, to avoid the widening afterwards.

This is mainly useful when operating on very large buffers. For example, consider:

```
void zeroinit(char* base, unsigned width, unsigned height) {
  for(unsigned row = 0; row != height; ++row) {
    for(unsigned col = 0; col != width; ++col) {
      char* ptr = base + row * width + col;
      *ptr = 0;
    }
  }
}
```

This is fine in general, but if **width * height** overflows, you end up wrapping back to the beginning of **base** instead of processing the entire requested buffer.

Indeed, this only matters for pretty large buffers (4GB+), but that can happen very easily for example in image processing, where for that to happen you "only" need a ~269MPix image.

Options

UseCXXStaticCastsInCppSources

When suggesting fix-its for C++ code, should C++-style **static_cast<>()**'s be suggested, or C-style casts. Defaults to **true**.

UseCXXHeadersInCppSources

When suggesting to include the appropriate header in C++ code, should **<cstddef>** header be suggested, or **<stddef.h>**. Defaults to **true**.

Examples:

```
long mul(int a, int b) {
  return a * b; // warning: performing an implicit widening conversion to type ’long’ of a multiplication performed
}

char* ptr_add(char *base, int a, int b) {
  return base + a * b; // warning: result of multiplication in type ’int’ is used as a pointer offset after an implicit wid
}

char ptr_subscript(char *base, int a, int b) {
  return base[a * b]; // warning: result of multiplication in type ’int’ is used as a pointer offset after an implicit wide
}
```

bugprone-inaccurate-erase

Checks for inaccurate use of the **erase()** method.

Algorithms like **remove()** do not actually remove any element from the container but return an iterator to the first redundant element at the end of the container. These redundant elements must be removed using the **erase()** method. This check warns when not all of the elements will be removed due to using an inappropriate overload.

For example, the following code erases only one element:

```
std::vector<int> xs;
...
xs.erase(std::remove(xs.begin(), xs.end(), 10));
```

Call the two-argument overload of **erase()** to remove the subrange:

```
std::vector<int> xs;
...
xs.erase(std::remove(xs.begin(), xs.end(), 10), xs.end());
```

bugprone-incorrect-roundings

Checks the usage of patterns known to produce incorrect rounding. Programmers often use:

```
(int)(double_expression + 0.5)
```

to round the double expression to an integer. The problem with this:

1. It is unnecessarily slow.

2. It is incorrect. The number 0.499999975 (smallest representable float number below 0.5) rounds to 1.0. Even worse behavior for negative numbers where both -0.5f and -1.4f both round to 0.0.

bugprone-infinite-loop

Finds obvious infinite loops (loops where the condition variable is not changed at all).

Finding infinite loops is well-known to be impossible (halting problem). However, it is possible to detect some obvious infinite loops, for example, if the loop condition is not changed. This check detects such loops. A loop is considered infinite if it does not have any loop exit statement (**break**, **continue**, **goto**, **return**, **throw** or a call to a function called as **[[noreturn]]**) and all of the following conditions hold for every variable in the condition:

- It is a local variable.

- It has no reference or pointer aliases.

- It is not a structure or class member.

Furthermore, the condition must not contain a function call to consider the loop infinite since functions may return different values for different calls.

For example, the following loop is considered infinite *i* is not changed in the body:

```
int i = 0, j = 0;
while (i < 10) {
  ++j;
}
```

bugprone-integer-division

Finds cases where integer division in a floating point context is likely to cause unintended loss of precision.

No reports are made if divisions are part of the following expressions:

- operands of operators expecting integral or bool types,

- call expressions of integral or bool types, and

- explicit cast expressions to integral or bool types,

  as these are interpreted as signs of deliberateness from the programmer.

  Examples:

```
float floatFunc(float);
int intFunc(int);
double d;
int i = 42;

// Warn, floating-point values expected.
d = 32 * 8 / (2 + i);
d = 8 * floatFunc(1 + 7 / 2);
d = i / (1 << 4);

// OK, no integer division.
d = 32 * 8.0 / (2 + i);
d = 8 * floatFunc(1 + 7.0 / 2);
d = (double)i / (1 << 4);

// OK, there are signs of deliberateness.
d = 1 << (i / 2);
d = 9 + intFunc(6 * i / 32);
d = (int)(i / 32) - 8;
```

bugprone-lambda-function-name

Checks for attempts to get the name of a function from within a lambda expression. The name of a lambda is always something like **operator()**, which is almost never what was intended.

Example:

```
void FancyFunction() {
  [] { printf("Called from %s\n", __func__); }();
  [] { printf("Now called from %s\n", __FUNCTION__); }();
}
```

Output:

```
Called from operator()
```

Now called from operator()

Likely intended output:

Called from FancyFunction
Now called from FancyFunction

bugprone-macro-parentheses
Finds macros that can have unexpected behavior due to missing parentheses.

Macros are expanded by the preprocessor as-is. As a result, there can be unexpected behavior; operators may be evaluated in unexpected order and unary operators may become binary operators, etc.

When the replacement list has an expression, it is recommended to surround it with parentheses. This ensures that the macro result is evaluated completely before it is used.

It is also recommended to surround macro arguments in the replacement list with parentheses. This ensures that the argument value is calculated properly.

bugprone-macro-repeated-side-effects
Checks for repeated argument with side effects in macros.

bugprone-misplaced-operator-in-strlen-in-alloc
Finds cases where **1** is added to the string in the argument to **strlen()**, **strnlen()**, **strnlen_s()**, **wcslen()**, **wcsnlen()**, and **wcsnlen_s()** instead of the result and the value is used as an argument to a memory allocation function (**malloc()**, **calloc()**, **realloc()**, **alloca()**) or the **new[]** operator in *C++*. The check detects error cases even if one of these functions (except the **new[]** operator) is called by a constant function pointer. Cases where **1** is added both to the parameter and the result of the **strlen()**-like function are ignored, as are cases where the whole addition is surrounded by extra parentheses.

C example code:

```
void bad_malloc(char *str) {
  char *c = (char*) malloc(strlen(str + 1));
}
```

The suggested fix is to add **1** to the return value of **strlen()** and not to its argument. In the example above the fix would be

```
char *c = (char*) malloc(strlen(str) + 1);
```

C++ example code:

```
void bad_new(char *str) {
  char *c = new char[strlen(str + 1)];
}
```

As in the *C* code with the **malloc()** function, the suggested fix is to add **1** to the return value of **strlen()** and not to its argument. In the example above the fix would be

```
char *c = new char[strlen(str) + 1];
```

Example for silencing the diagnostic:

```
void bad_malloc(char *str) {
  char *c = (char*) malloc(strlen((str + 1)));
}
```

bugprone-misplaced-pointer-arithmetic-in-alloc

Finds cases where an integer expression is added to or subtracted from the result of a memory allocation function (**malloc()**, **calloc()**, **realloc()**, **alloca()**) instead of its argument. The check detects error cases even if one of these functions is called by a constant function pointer.

Example code:

```
void bad_malloc(int n) {
  char *p = (char*) malloc(n) + 10;
}
```

The suggested fix is to add the integer expression to the argument of **malloc** and not to its result. In the example above the fix would be

```
char *p = (char*) malloc(n + 10);
```

bugprone-misplaced-widening-cast

This check will warn when there is a cast of a calculation result to a bigger type. If the intention of the cast is to avoid loss of precision then the cast is misplaced, and there can be loss of precision. Otherwise the cast is ineffective.

Example code:

```
long f(int x) {
  return (long)(x * 1000);
}
```

The result **x * 1000** is first calculated using **int** precision. If the result exceeds **int** precision there is loss of precision. Then the result is casted to **long**.

If there is no loss of precision then the cast can be removed or you can explicitly cast to **int** instead.

If you want to avoid loss of precision then put the cast in a proper location, for instance:

```
long f(int x) {
  return (long)x * 1000;
}
```

Implicit casts

Forgetting to place the cast at all is at least as dangerous and at least as common as misplacing it. If *CheckImplicitCasts* is enabled the check also detects these cases, for instance:

```
long f(int x) {
  return x * 1000;
}
```

Floating point

Currently warnings are only written for integer conversion. No warning is written for this code:

```
double f(float x) {
  return (double)(x * 10.0f);
}
```

Options

CheckImplicitCasts

If *true*, enables detection of implicit casts. Default is *false*.

bugprone-move-forwarding-reference

Warns if **std::move** is called on a forwarding reference, for example:

```
template <typename T>
```

```
void foo(T&& t) {
  bar(std::move(t));
}
```

Forwarding references should typically be passed to **std::forward** instead of **std::move**, and this is the fix that will be suggested.

(A forwarding reference is an rvalue reference of a type that is a deduced function template argument.)

In this example, the suggested fix would be

```
bar(std::forward<T>(t));
```

Background

Code like the example above is sometimes written with the expectation that **T&&** will always end up being an rvalue reference, no matter what type is deduced for **T**, and that it is therefore not possible to pass an lvalue to **foo()**. However, this is not true. Consider this example:

```
std::string s = "Hello, world";
foo(s);
```

This code compiles and, after the call to **foo()**, **s** is left in an indeterminate state because it has been moved from. This may be surprising to the caller of **foo()** because no **std::move** was used when calling **foo()**.

The reason for this behavior lies in the special rule for template argument deduction on function templates like **foo()** -- i.e. on function templates that take an rvalue reference argument of a type that is a deduced function template argument. (See section [temp.deduct.call]/3 in the C++11 standard.)

If **foo()** is called on an lvalue (as in the example above), then **T** is deduced to be an lvalue reference. In the example, **T** is deduced to be **std::string &**. The type of the argument **t** therefore becomes **std::string& &&**; by the reference collapsing rules, this collapses to **std::string&**.

This means that the **foo(s)** call passes **s** as an lvalue reference, and **foo()** ends up moving **s** and thereby placing it into an indeterminate state.

bugprone-multiple-statement-macro

Detect multiple statement macros that are used in unbraced conditionals. Only the first statement of the

macro will be inside the conditional and the other ones will be executed unconditionally.

Example:

```
#define INCREMENT_TWO(x, y) (x)++; (y)++
if (do_increment)
  INCREMENT_TWO(a, b);  // (b)++ will be executed unconditionally.
```

bugprone-narrowing-conversions

The bugprone-narrowing-conversions check is an alias, please see *cppcoreguidelines-narrowing-conversions* for more information.

bugprone-no-escape

Finds pointers with the **noescape** attribute that are captured by an asynchronously-executed block. The block arguments in **dispatch_async()** and **dispatch_after()** are guaranteed to escape, so it is an error if a pointer with the **noescape** attribute is captured by one of these blocks.

The following is an example of an invalid use of the **noescape** attribute.

```
void foo(__attribute__((noescape)) int *p) {
  dispatch_async(queue, ^{
    *p = 123;
  });
});
```

bugprone-not-null-terminated-result

Finds function calls where it is possible to cause a not null-terminated result. Usually the proper length of a string is **strlen(src) + 1** or equal length of this expression, because the null terminator needs an extra space. Without the null terminator it can result in undefined behavior when the string is read.

The following and their respective **wchar_t** based functions are checked:

memcpy, **memcpy_s**, **memchr**, **memmove**, **memmove_s**, **strerror_s**, **strncmp**, **strxfrm**

The following is a real-world example where the programmer forgot to increase the passed third argument, which is **size_t length**. That is why the length of the allocated memory is not enough to hold the null terminator.

```
static char *stringCpy(const std::string &str) {
  char *result = reinterpret_cast<char *>(malloc(str.size()));
```

```
  memcpy(result, str.data(), str.size());
  return result;
}
```

In addition to issuing warnings, fix-it rewrites all the necessary code. It also tries to adjust the capacity of the destination array:

```
static char *stringCpy(const std::string &str) {
  char *result = reinterpret_cast<char *>(malloc(str.size() + 1));
  strcpy(result, str.data());
  return result;
}
```

Note: It cannot guarantee to rewrite every of the path-sensitive memory allocations.

Transformation rules of 'memcpy()'

It is possible to rewrite the **memcpy()** and **memcpy_s()** calls as the following four functions: **strcpy()**, **strncpy()**, **strcpy_s()**, **strncpy_s()**, where the latter two are the safer versions of the former two. It rewrites the **wchar_t** based memory handler functions respectively.

Rewrite based on the destination array

- If copy to the destination array cannot overflow [1] the new function should be the older copy function (ending with **cpy**), because it is more efficient than the safe version.
- If copy to the destination array can overflow [1] and *WantToUseSafeFunctions* is set to *true* and it is possible to obtain the capacity of the destination array then the new function could be the safe version (ending with **cpy_s**).
- If the new function is could be safe version and C++ files are analyzed and the destination array is plain **char**/**wchar_t** without **un/signed** then the length of the destination array can be omitted.
- If the new function is could be safe version and the destination array is **un/signed** it needs to be casted to plain **char ***/**wchar_t ***.

[1] It is possible to overflow:

- If the capacity of the destination array is unknown.
- If the given length is equal to the destination array's capacity.

Rewrite based on the length of the source string

- If the given length is **strlen(source)** or equal length of this expression then the new function should be the older copy function (ending with **cpy**), as it is more efficient than the safe version (ending with **cpy_s**).

- Otherwise we assume that the programmer wanted to copy 'N' characters, so the new function is **ncpy**-like which copies 'N' characters.

Transformations with 'strlen()' or equal length of this expression

It transforms the **wchar_t** based memory and string handler functions respectively (where only **strerror_s** does not have **wchar_t** based alias).

Memory handler functions

memcpy Please visit the *Transformation rules of 'memcpy()'* section.

memchr Usually there is a C-style cast and it is needed to be removed, because the new function **strchr**'s return type is correct. The given length is going to be removed.

memmove If safe functions are available the new function is **memmove_s**, which has a new second argument which is the length of the destination array, it is adjusted, and the length of the source string is incremented by one. If safe functions are not available the given length is incremented by one.

memmove_s The given length is incremented by one.

String handler functions

strerror_s The given length is incremented by one.

strncmp If the third argument is the first or the second argument's **length + 1** it has to be truncated without the **+ 1** operation.

strxfrm The given length is incremented by one.

Options

WantToUseSafeFunctions

The value *true* specifies that the target environment is considered to implement '_s' suffixed memory and string handler functions which are safer than older versions (e.g. 'memcpy_s()'). The default value is *true*.

bugprone-parent-virtual-call

Detects and fixes calls to grand-...parent virtual methods instead of calls to overridden parent's virtual methods.

```
struct A {
  int virtual foo() {...}
};

struct B: public A {
  int foo() override {...}
};

struct C: public B {
  int foo() override { A::foo(); }
//                     ^^^^^^^^
// warning: qualified name A::foo refers to a member overridden in subclass; did you mean 'B'?  [bugprone-parent-
};
```

bugprone-posix-return

Checks if any calls to **pthread_*** or **posix_*** functions (except **posix_openpt**) expect negative return values. These functions return either **0** on success or an **errno** on failure, which is positive only.

Example buggy usage looks like:

```
if (posix_fadvise(...) < 0) {
```

This will never happen as the return value is always non-negative. A simple fix could be:

```
if (posix_fadvise(...) > 0) {
```

bugprone-redundant-branch-condition

Finds condition variables in nested **if** statements that were also checked in the outer **if** statement and were not changed.

Simple example:

```
bool onFire = isBurning();
if (onFire) {
  if (onFire)
    scream();
```

```
}
```

Here *onFire* is checked both in the outer **if** and the inner **if** statement without a possible change between the two checks. The check warns for this code and suggests removal of the second checking of variable *onFire*.

The checker also detects redundant condition checks if the condition variable is an operand of a logical "and" (**&&**) or a logical "or" (||) operator:

```
bool onFire = isBurning();
if (onFire) {
  if (onFire && peopleInTheBuilding > 0)
    scream();
}

bool onFire = isBurning();
if (onFire) {
  if (onFire || isCollapsing())
    scream();
}
```

In the first case (logical "and") the suggested fix is to remove the redundant condition variable and keep the other side of the **&&**. In the second case (logical "or") the whole **if** is removed similarly to the simple case on the top.

The condition of the outer **if** statement may also be a logical "and" (**&&**) expression:

```
bool onFire = isBurning();
if (onFire && fireFighters < 10) {
  if (someOtherCondition()) {
    if (onFire)
      scream();
  }
}
```

The error is also detected if both the outer statement is a logical "and" (**&&**) and the inner statement is a logical "and" (**&&**) or "or" (||). The inner **if** statement does not have to be a direct descendant of the outer one.

No error is detected if the condition variable may have been changed between the two checks:

```
bool onFire = isBurning();
if (onFire) {
  tryToExtinguish(onFire);
  if (onFire && peopleInTheBuilding > 0)
    scream();
}
```

Every possible change is considered, thus if the condition variable is not a local variable of the function, it is a volatile or it has an alias (pointer or reference) then no warning is issued.

Known limitations

The **else** branch is not checked currently for negated condition variable:

```
bool onFire = isBurning();
if (onFire) {
  scream();
} else {
  if (!onFire) {
    continueWork();
  }
}
```

The checker currently only detects redundant checking of single condition variables. More complex expressions are not checked:

```
if (peopleInTheBuilding == 1) {
  if (peopleInTheBuilding == 1) {
    doSomething();
  }
}
```

bugprone-reserved-identifier

cert-dcl37-c and *cert-dcl51-cpp* redirect here as an alias for this check.

Checks for usages of identifiers reserved for use by the implementation.

The C and C++ standards both reserve the following names for such use:

- identifiers that begin with an underscore followed by an uppercase letter;

- identifiers in the global namespace that begin with an underscore.

  The C standard additionally reserves names beginning with a double underscore, while the C++ standard strengthens this to reserve names with a double underscore occurring anywhere.

  Violating the naming rules above results in undefined behavior.

```
namespace NS {
  void __f(); // name is not allowed in user code
  using _Int = int; // same with this
  #define cool__macro // also this
}
int _g(); // disallowed in global namespace only
```

  The check can also be inverted, i.e. it can be configured to flag any identifier that is _not_ a reserved identifier. This mode is for use by e.g. standard library implementors, to ensure they don't infringe on the user namespace.

  This check does not (yet) check for other reserved names, e.g. macro names identical to language keywords, and names specifically reserved by language standards, e.g. C++ 'zombie names' and C future library directions.

  This check corresponds to CERT C Coding Standard rule *DCL37-C. Do not declare or define a reserved identifier* as well as its C++ counterpart, *DCL51-CPP. Do not declare or define a reserved identifier*.

Options

Invert
If *true*, inverts the check, i.e. flags names that are not reserved. Default is *false*.

AllowedIdentifiers
Semicolon-separated list of names that the check ignores. Default is an empty list.

bugprone-shared-ptr-array-mismatch
Finds initializations of C++ shared pointers to non-array type that are initialized with an array.

If a shared pointer **std::shared_ptr<T>** is initialized with a new-expression **new T[]** the memory is not deallocated correctly. The pointer uses plain **delete** in this case to deallocate the target memory. Instead a **delete[]** call is needed. A **std::shared_ptr<T[]>** calls the correct delete operator.

The check offers replacement of **shared_ptr<T>** to **shared_ptr<T[]>** if it is used at a single variable declaration (one variable in one statement).

Example:

```
std::shared_ptr<Foo> x(new Foo[10]); // -> std::shared_ptr<Foo[]> x(new Foo[10]);
//                   ^ warning: shared pointer to non-array is initialized with array [bugprone-shared-ptr-array-mismatch
std::shared_ptr<Foo> x1(new Foo), x2(new Foo[10]); // no replacement
//                               ^ warning: shared pointer to non-array is initialized with array [bugprone-shared-ptr-array-m

std::shared_ptr<Foo> x3(new Foo[10], [](const Foo *ptr) { delete[] ptr; }); // no warning

struct S {
  std::shared_ptr<Foo> x(new Foo[10]); // no replacement in this case
  //                   ^ warning: shared pointer to non-array is initialized with array [bugprone-shared-ptr-array-mismatc
};
```

This check partially covers the CERT C++ Coding Standard rule *MEM51-CPP. Properly deallocate dynamically allocated resources* However, only the **std::shared_ptr** case is detected by this check.

bugprone-signal-handler

Finds functions registered as signal handlers that call non asynchronous-safe functions. Any function that cannot be determined to be an asynchronous-safe function call is assumed to be non-asynchronous-safe by the checker, including user functions for which only the declaration is visible. User function calls with visible definition are checked recursively. The check handles only C code. Only the function names are considered and the fact that the function is a system-call, but no other restrictions on the arguments passed to the functions (the **signal** call is allowed without restrictions).

This check corresponds to the CERT C Coding Standard rule *SIG30-C. Call only asynchronous-safe functions within signal handlers* and has an alias name **cert-sig30-c**.

AsyncSafeFunctionSet

Selects which set of functions is considered as asynchronous-safe (and therefore allowed in signal handlers). Value **minimal** selects a minimal set that is defined in the CERT SIG30-C rule and includes functions **abort()**, **_Exit()**, **quick_exit()** and **signal()**. Value **POSIX** selects a larger set of functions that is listed in POSIX.1-2017 (see *this link* for more information). The function **quick_exit** is not included in the shown list. It is assumable that the reason is that the list was not updated for C11. The checker includes **quick_exit** in the set of safe functions. Functions

registered as exit handlers are not checked.

Default is **POSIX**.

bugprone-signed-char-misuse

cert-str34-c redirects here as an alias for this check. For the CERT alias, the *DiagnoseSignedUnsignedCharComparisons* option is set to *false*.

Finds those **signed char** -> integer conversions which might indicate a programming error. The basic problem with the **signed char**, that it might store the non-ASCII characters as negative values. This behavior can cause a misunderstanding of the written code both when an explicit and when an implicit conversion happens.

When the code contains an explicit **signed char** -> integer conversion, the human programmer probably expects that the converted value matches with the character code (a value from [0..255]), however, the actual value is in [-128..127] interval. To avoid this kind of misinterpretation, the desired way of converting from a **signed char** to an integer value is converting to **unsigned char** first, which stores all the characters in the positive [0..255] interval which matches the known character codes.

In case of implicit conversion, the programmer might not actually be aware that a conversion happened and char value is used as an integer. There are some use cases when this unawareness might lead to a functionally imperfect code. For example, checking the equality of a **signed char** and an **unsigned char** variable is something we should avoid in C++ code. During this comparison, the two variables are converted to integers which have different value ranges. For **signed char**, the non-ASCII characters are stored as a value in [-128..-1] interval, while the same characters are stored in the [128..255] interval for an **unsigned char**.

It depends on the actual platform whether plain **char** is handled as **signed char** by default and so it is caught by this check or not. To change the default behavior you can use **-funsigned-char** and **-fsigned-char** compilation options.

Currently, this check warns in the following cases: - **signed char** is assigned to an integer variable - **signed char** and **unsigned char** are compared with equality/inequality operator - **signed char** is converted to an integer in the array subscript

See also: *STR34-C. Cast characters to unsigned char before converting to larger integer sizes*

A good example from the CERT description when a **char** variable is used to read from a file that might contain non-ASCII characters. The problem comes up when the code uses the **-1** integer value as EOF, while the 255 character code is also stored as **-1** in two’s complement form of char type. See a simple

example of this bellow. This code stops not only when it reaches the end of the file, but also when it gets a character with the 255 code.

```
#define EOF (-1)

int read(void) {
  char CChar;
  int IChar = EOF;

  if (readChar(CChar)) {
    IChar = CChar;
  }
  return IChar;
}
```

A proper way to fix the code above is converting the **char** variable to an **unsigned char** value first.

```
#define EOF (-1)

int read(void) {
  char CChar;
  int IChar = EOF;

  if (readChar(CChar)) {
    IChar = static_cast<unsigned char>(CChar);
  }
  return IChar;
}
```

Another use case is checking the equality of two **char** variables with different signedness. Inside the non-ASCII value range this comparison between a **signed char** and an **unsigned char** always returns **false**.

```
bool compare(signed char SChar, unsigned char USChar) {
  if (SChar == USChar)
    return true;
  return false;
}
```

The easiest way to fix this kind of comparison is casting one of the arguments, so both arguments will have the same type.

```
bool compare(signed char SChar, unsigned char USChar) {
  if (static_cast<unsigned char>(SChar) == USChar)
    return true;
  return false;
}
```

CharTypdefsToIgnore

A semicolon-separated list of typedef names. In this list, we can list typedefs for **char** or **signed char**, which will be ignored by the check. This is useful when a typedef introduces an integer alias like **sal_Int8** or **int8_t**. In this case, human misinterpretation is not an issue.

DiagnoseSignedUnsignedCharComparisons

When *true*, the check will warn on **signed char**/**unsigned char** comparisons, otherwise these comparisons are ignored. By default, this option is set to *true*.

bugprone-sizeof-container

The check finds usages of **sizeof** on expressions of STL container types. Most likely the user wanted to use **.size()** instead.

All class/struct types declared in namespace **std::** having a const **size()** method are considered containers, with the exception of **std::bitset** and **std::array**.

Examples:

```
std::string s;
int a = 47 + sizeof(s); // warning: sizeof() doesn't return the size of the container. Did you mean .size()?

int b = sizeof(std::string); // no warning, probably intended.

std::string array_of_strings[10];
int c = sizeof(array_of_strings) / sizeof(array_of_strings[0]); // no warning, definitely intended.

std::array<int, 3> std_array;
int d = sizeof(std_array); // no warning, probably intended.
```

bugprone-sizeof-expression

The check finds usages of **sizeof** expressions which are most likely errors.

The **sizeof** operator yields the size (in bytes) of its operand, which may be an expression or the parenthesized name of a type. Misuse of this operator may be leading to errors and possible software vulnerabilities.

Suspicious usage of 'sizeof(K)'

A common mistake is to query the **sizeof** of an integer literal. This is equivalent to query the size of its type (probably **int**). The intent of the programmer was probably to simply get the integer and not its size.

```
#define BUFLEN 42
char buf[BUFLEN];
memset(buf, 0, sizeof(BUFLEN));  // sizeof(42) ==> sizeof(int)
```

Suspicious usage of 'sizeof(expr)'

In cases, where there is an enum or integer to represent a type, a common mistake is to query the **sizeof** on the integer or enum that represents the type that should be used by **sizeof**. This results in the size of the integer and not of the type the integer represents:

```
enum data_type {
  FLOAT_TYPE,
  DOUBLE_TYPE
};

struct data {
  data_type type;
  void* buffer;
  data_type get_type() {
    return type;
  }
};

void f(data d, int numElements) {
  // should be sizeof(float) or sizeof(double), depending on d.get_type()
  int numBytes = numElements * sizeof(d.get_type());
  ...
}
```

Suspicious usage of 'sizeof(this)'

The **this** keyword is evaluated to a pointer to an object of a given type. The expression **sizeof(this)** is returning the size of a pointer. The programmer most likely wanted the size of the object and not the

size of the pointer.

```
class Point {
 [...]
 size_t size() { return sizeof(this); }  // should probably be sizeof(*this)
 [...]
};
```

Suspicious usage of 'sizeof(char*)'

There is a subtle difference between declaring a string literal with **char* A = ""** and **char A[] = ""**. The first case has the type **char*** instead of the aggregate type **char[]**. Using **sizeof** on an object declared with **char*** type is returning the size of a pointer instead of the number of characters (bytes) in the string literal.

```
const char* kMessage = "Hello World!";      // const char kMessage[] = "...";
void getMessage(char* buf) {
 memcpy(buf, kMessage, sizeof(kMessage));  // sizeof(char*)
}
```

Suspicious usage of 'sizeof(A*)'

A common mistake is to compute the size of a pointer instead of its pointee. These cases may occur because of explicit cast or implicit conversion.

```
int A[10];
memset(A, 0, sizeof(A + 0));

struct Point point;
memset(point, 0, sizeof(&point));
```

Suspicious usage of 'sizeof(...)/sizeof(...)'

Dividing **sizeof** expressions is typically used to retrieve the number of elements of an aggregate. This check warns on incompatible or suspicious cases.

In the following example, the entity has 10-bytes and is incompatible with the type **int** which has 4 bytes.

```
char buf[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };  // sizeof(buf) => 10
void getMessage(char* dst) {
 memcpy(dst, buf, sizeof(buf) / sizeof(int));  // sizeof(int) => 4  [incompatible sizes]
}
```

In the following example, the expression **sizeof(Values)** is returning the size of **char***. One can easily be fooled by its declaration, but in parameter declaration the size ’10’ is ignored and the function is receiving a **char***.

```
char OrderedValues[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
return CompareArray(char Values[10]) {
 return memcmp(OrderedValues, Values, sizeof(Values)) == 0;  // sizeof(Values) ==> sizeof(char*) [implicit cast
}
```

Suspicious ’sizeof’ by ’sizeof’ expression

Multiplying **sizeof** expressions typically makes no sense and is probably a logic error. In the following example, the programmer used ***** instead of **/**.

```
const char kMessage[] = "Hello World!";
void getMessage(char* buf) {
 memcpy(buf, kMessage, sizeof(kMessage) * sizeof(char));  //  sizeof(kMessage) / sizeof(char)
}
```

This check may trigger on code using the arraysize macro. The following code is working correctly but should be simplified by using only the **sizeof** operator.

```
extern Object objects[100];
void InitializeObjects() {
 memset(objects, 0, arraysize(objects) * sizeof(Object));  // sizeof(objects)
}
```

Suspicious usage of ’sizeof(sizeof(...))’

Getting the **sizeof** of a **sizeof** makes no sense and is typically an error hidden through macros.

```
#define INT_SZ sizeof(int)
int buf[] = { 42 };
void getInt(int* dst) {
 memcpy(dst, buf, sizeof(INT_SZ));  // sizeof(sizeof(int)) is suspicious.
}
```

Options

WarnOnSizeOfConstant

When *true*, the check will warn on an expression like **sizeof(CONSTANT)**. Default is *true*.

WarnOnSizeOfIntegerExpression
When *true*, the check will warn on an expression like **sizeof(expr)** where the expression results in an integer. Default is *false*.

WarnOnSizeOfThis
When *true*, the check will warn on an expression like **sizeof(this)**. Default is *true*.

WarnOnSizeOfCompareToConstant
When *true*, the check will warn on an expression like **sizeof(expr) <= k** for a suspicious constant *k* while *k* is *0* or greater than *0x8000*. Default is *true*.

bugprone-spuriously-wake-up-functions
Finds **cnd_wait**, **cnd_timedwait**, **wait**, **wait_for**, or **wait_until** function calls when the function is not invoked from a loop that checks whether a condition predicate holds or the function has a condition parameter.

This check corresponds to the CERT C++ Coding Standard rule *CON54-CPP. Wrap functions that can spuriously wake up in a loop*. and CERT C Coding Standard rule *CON36-C. Wrap functions that can spuriously wake up in a loop*.

bugprone-string-constructor
Finds string constructors that are suspicious and probably errors.

A common mistake is to swap parameters to the ’fill’ string-constructor.

Examples:

std::string str(’x’, 50); // should be str(50, ’x’)

Calling the string-literal constructor with a length bigger than the literal is suspicious and adds extra random characters to the string.

Examples:

std::string("test", 200); // Will include random characters after "test".
std::string_view("test", 200);

Creating an empty string from constructors with parameters is considered suspicious. The programmer should use the empty constructor instead.

Examples:

```
std::string("test", 0);   // Creation of an empty string.
std::string_view("test", 0);
```

Options

WarnOnLargeLength
When *true*, the check will warn on a string with a length greater than *LargeLengthThreshold*. Default is *true*.

LargeLengthThreshold
An integer specifying the large length threshold. Default is *0x800000*.

StringNames
Default is *::std::basic_string;::std::basic_string_view*.

Semicolon-delimited list of class names to apply this check to. By default *::std::basic_string* applies to **std::string** and **std::wstring**. Set to e.g. *::std::basic_string;llvm::StringRef;QString* to perform this check on custom classes.

bugprone-string-integer-assignment
The check finds assignments of an integer to **std::basic_string<CharT>** (**std::string**, **std::wstring**, etc.). The source of the problem is the following assignment operator of **std::basic_string<CharT>**:

```
basic_string& operator=( CharT ch );
```

Numeric types can be implicitly casted to character types.

```
std::string s;
int x = 5965;
s = 6;
s = x;
```

Use the appropriate conversion functions or character literals.

```
std::string s;
int x = 5965;
s = '6';
s = std::to_string(x);
```

In order to suppress false positives, use an explicit cast.

```
std::string s;
s = static_cast<char>(6);
```

bugprone-string-literal-with-embedded-nul

Finds occurrences of string literal with embedded NUL character and validates their usage.

Invalid escaping

Special characters can be escaped within a string literal by using their hexadecimal encoding like **\x42**. A common mistake is to escape them like this **\0x42** where the **\0** stands for the NUL character.

```
const char* Example[] = "Invalid character: \0x12 should be \x12";
const char* Bytes[] = "\x03\0x02\0x01\0x00\0xFF\0xFF\0xFF";
```

Truncated literal

String-like classes can manipulate strings with embedded NUL as they are keeping track of the bytes and the length. This is not the case for a **char*** (NUL-terminated) string.

A common mistake is to pass a string-literal with embedded NUL to a string constructor expecting a NUL-terminated string. The bytes after the first NUL character are truncated.

```
std::string str("abc\0def");  // "def" is truncated
str += "\0";                  // This statement is doing nothing
if (str == "\0abc") return;   // This expression is always true
```

bugprone-stringview-nullptr

Checks for various ways that the **const CharT*** constructor of **std::basic_string_view** can be passed a null argument and replaces them with the default constructor in most cases. For the comparison operators, braced initializer list does not compile so instead a call to **.empty()** or the empty string literal are used, where appropriate.

This prevents code from invoking behavior which is unconditionally undefined. The single-argument **const CharT*** constructor does not check for the null case before dereferencing its input. The standard is slated to add an explicitly-deleted overload to catch some of these cases: wg21.link/p2166

To catch the additional cases of **NULL** (which expands to **__null**) and **0**, first run the **modernize-use-nullptr** check to convert the callers to **nullptr**.

```
std::string_view sv = nullptr;
```

```
sv = nullptr;

bool is_empty = sv == nullptr;
bool isnt_empty = sv != nullptr;

accepts_sv(nullptr);

accepts_sv({{}});  // A

accepts_sv({nullptr, 0});  // B
```

is translated into...

```
std::string_view sv = {};

sv = {};

bool is_empty = sv.empty();
bool isnt_empty = !sv.empty();

accepts_sv("");

accepts_sv("");  // A

accepts_sv({nullptr, 0});  // B
```

NOTE:

The source pattern with trailing comment "A" selects the **(const CharT*)** constructor overload and then value-initializes the pointer, causing a null dereference. It happens to not include the **nullptr** literal, but it is still within the scope of this ClangTidy check.

NOTE:

The source pattern with trailing comment "B" selects the **(const CharT*, size_type)** constructor which is perfectly valid, since the length argument is **0**. It is not changed by this ClangTidy check.

bugprone-suspicious-enum-usage

The checker detects various cases when an enum is probably misused (as a bitmask).

1. When "ADD" or "bitwise OR" is used between two enum which come from different types and these types value ranges are not disjoint.

The following cases will be investigated only using *StrictMode*. We regard the enum as a (suspicious) bitmask if the three conditions below are true at the same time:

- at most half of the elements of the enum are non pow-of-2 numbers (because of short enumerations)
- there is another non pow-of-2 number than the enum constant representing all choices (the result "bitwise OR" operation of all enum elements)
- enum type variable/enumconstant is used as an argument of a + or "bitwise OR " operator

So whenever the non pow-of-2 element is used as a bitmask element we diagnose a misuse and give a warning.

2. Investigating the right hand side of += and |= operator.
3. Check only the enum value side of a | and + operator if one of them is not enum val.
4. Check both side of | or + operator where the enum values are from the same enum type.

Examples:

```
enum { A, B, C };
enum { D, E, F = 5 };
enum { G = 10, H = 11, I = 12 };

unsigned flag;
flag =
  A |
  H; // OK, disjoint value intervals in the enum types ->probably good use.
flag = B | F; // Warning, have common values so they are probably misused.

// Case 2:
enum Bitmask {
  A = 0,
  B = 1,
  C = 2,
  D = 4,
  E = 8,
  F = 16,
  G = 31 // OK, real bitmask.
```

```
};

enum Almostbitmask {
  AA = 0,
  BB = 1,
  CC = 2,
  DD = 4,
  EE = 8,
  FF = 16,
  GG // Problem, forgot to initialize.
};

unsigned flag = 0;
flag |= E; // OK.
flag |=
    EE; // Warning at the decl, and note that it was used here as a bitmask.
```

Options

StrictMode
Default value: 0. When non-null the suspicious bitmask usage will be investigated additionally to the different enum usage check.

bugprone-suspicious-include
The check detects various cases when an include refers to what appears to be an implementation file, which often leads to hard-to-track-down ODR violations.

Examples:

```
#include "Dinosaur.hpp"      // OK, .hpp files tend not to have definitions.
#include "Pterodactyl.h"     // OK, .h files tend not to have definitions.
#include "Velociraptor.cpp" // Warning, filename is suspicious.
#include_next <stdio.c>      // Warning, filename is suspicious.
```

Options

HeaderFileExtensions
Default value: **";h;hh;hpp;hxx"** A semicolon-separated list of filename extensions of header files (the filename extensions should not contain a "." prefix). For extension-less header files, use an empty string or leave an empty string between ";" if there are other filename extensions.

ImplementationFileExtensions

Default value: **"c;cc;cpp;cxx"** Likewise, a semicolon-separated list of filename extensions of implementation files.

bugprone-suspicious-memory-comparison

Finds potentially incorrect calls to **memcmp()** based on properties of the arguments. The following cases are covered:

Case 1: Non-standard-layout type

Comparing the object representations of non-standard-layout objects may not properly compare the value representations.

Case 2: Types with no unique object representation

Objects with the same value may not have the same object representation. This may be caused by padding or floating-point types.

See also: *EXP42-C. Do not compare padding data* and *FLP37-C. Do not use object representations to compare floating-point values*

This check is also related to and partially overlaps the CERT C++ Coding Standard rules *OOP57-CPP. Prefer special member functions and overloaded operators to C Standard Library functions* and *EXP62-CPP. Do not access the bits of an object representation that are not part of the object's value representation*

bugprone-suspicious-memset-usage

This check finds **memset()** calls with potential mistakes in their arguments. Considering the function as **void* memset(void* destination, int fill_value, size_t byte_count)**, the following cases are covered:

Case 1: Fill value is a character ‘’0’‘

Filling up a memory area with ASCII code 48 characters is not customary, possibly integer zeroes were intended instead. The check offers a replacement of **'0'** with **0**. Memsetting character pointers with **'0'** is allowed.

Case 2: Fill value is truncated

Memset converts **fill_value** to **unsigned char** before using it. If **fill_value** is out of unsigned character range, it gets truncated and memory will not contain the desired pattern.

Case 3: Byte count is zero

Calling memset with a literal zero in its **byte_count** argument is likely to be unintended and swapped with **fill_value**. The check offers to swap these two arguments.

Corresponding cpplint.py check name: **runtime/memset**.

Examples:

```
void foo() {
  int i[5] = {1, 2, 3, 4, 5};
  int *ip = i;
  char c = '1';
  char *cp = &c;
  int v = 0;

  // Case 1
  memset(ip, '0', 1); // suspicious
  memset(cp, '0', 1); // OK

  // Case 2
  memset(ip, 0xabcd, 1); // fill value gets truncated
  memset(ip, 0x00, 1);   // OK

  // Case 3
  memset(ip, sizeof(int), v); // zero length, potentially swapped
  memset(ip, 0, 1);           // OK
}
```

bugprone-suspicious-missing-comma

String literals placed side-by-side are concatenated at translation phase 6 (after the preprocessor). This feature is used to represent long string literal on multiple lines.

For instance, the following declarations are equivalent:

```
const char* A[] = "This is a test";
const char* B[] = "This" " is a "    "test";
```

A common mistake done by programmers is to forget a comma between two string literals in an array initializer list.

```
const char* Test[] = {
  "line 1",
  "line 2"     // Missing comma!
  "line 3",
  "line 4",
  "line 5"
};
```

The array contains the string "line 2line3" at offset 1 (i.e. Test[1]). Clang won't generate warnings at compile time.

This check may warn incorrectly on cases like:

```
const char* SupportedFormat[] = {
  "Error %s",
  "Code " PRIu64,   // May warn here.
  "Warning %s",
};
```

Options

SizeThreshold

An unsigned integer specifying the minimum size of a string literal to be considered by the check. Default is **5U**.

RatioThreshold

A string specifying the maximum threshold ratio [0, 1.0] of suspicious string literals to be considered. Default is **".2"**.

MaxConcatenatedTokens

An unsigned integer specifying the maximum number of concatenated tokens. Default is **5U**.

bugprone-suspicious-semicolon

Finds most instances of stray semicolons that unexpectedly alter the meaning of the code. More specifically, it looks for **if**, **while**, **for** and **for-range** statements whose body is a single semicolon, and then analyzes the context of the code (e.g. indentation) in an attempt to determine whether that is intentional.

```
if (x < y);
{
```

```
  x++;
}
```

Here the body of the **if** statement consists of only the semicolon at the end of the first line, and *x* will be incremented regardless of the condition.

```
while ((line = readLine(file)) != NULL);
  processLine(line);
```

As a result of this code, *processLine()* will only be called once, when the **while** loop with the empty body exits with *line == NULL*. The indentation of the code indicates the intention of the programmer.

```
if (x >= y);
x -= y;
```

While the indentation does not imply any nesting, there is simply no valid reason to have an *if* statement with an empty body (but it can make sense for a loop). So this check issues a warning for the code above.

To solve the issue remove the stray semicolon or in case the empty body is intentional, reflect this using code indentation or put the semicolon in a new line. For example:

```
while (readWhitespace());
  Token t = readNextToken();
```

Here the second line is indented in a way that suggests that it is meant to be the body of the *while* loop - whose body is in fact empty, because of the semicolon at the end of the first line.

Either remove the indentation from the second line:

```
while (readWhitespace());
Token t = readNextToken();
```

... or move the semicolon from the end of the first line to a new line:

```
while (readWhitespace())
  ;

  Token t = readNextToken();
```

In this case the check will assume that you know what you are doing, and will not raise a warning.

bugprone-suspicious-string-compare

Find suspicious usage of runtime string comparison functions. This check is valid in C and C++.

Checks for calls with implicit comparator and proposed to explicitly add it.

```
if (strcmp(...))       // Implicitly compare to zero
if (!strcmp(...))      // Won't warn
if (strcmp(...) != 0)  // Won't warn
```

Checks that compare function results (i.e., **strcmp**) are compared to valid constant. The resulting value is

```
<  0    when lower than,
>  0    when greater than,
== 0    when equals.
```

A common mistake is to compare the result to *1* or *-1*.

```
if (strcmp(...) == -1)  // Incorrect usage of the returned value.
```

Additionally, the check warns if the results value is implicitly cast to a *suspicious* non-integer type. It's happening when the returned value is used in a wrong context.

```
if (strcmp(...) < 0.)  // Incorrect usage of the returned value.
```

Options

WarnOnImplicitComparison

When *true*, the check will warn on implicit comparison. *true* by default.

WarnOnLogicalNotComparison

When *true*, the check will warn on logical not comparison. *false* by default.

StringCompareLikeFunctions

A string specifying the comma-separated names of the extra string comparison functions. Default is an empty string. The check will detect the following string comparison functions: *__builtin_memcmp*, *__builtin_strcasecmp*, *__builtin_strcmp*, *__builtin_strncasecmp*,

__builtin_strncmp, *_mbscmp*, *_mbscmp_l*, *_mbsicmp*, *_mbsicmp_l*, *_mbsnbcmp*, *_mbsnbcmp_l*, *_mbsnbicmp*, *_mbsnbicmp_l*, *_mbsncmp*, *_mbsncmp_l*, *_mbsnicmp*, *_mbsnicmp_l*, *_memicmp*, *_memicmp_l*, *_stricmp*, *_stricmp_l*, *_strnicmp*, *_strnicmp_l*, *_wcsicmp*, *_wcsicmp_l*, *_wcsnicmp*, *_wcsnicmp_l*, *lstrcmp*, *lstrcmpi*, *memcmp*, *memicmp*, *strcasecmp*, *strcmp*, *strcmpi*, *stricmp*, *strncasecmp*, *strncmp*, *strnicmp*, *wcscasecmp*, *wcscmp*, *wcsicmp*, *wcsncmp*, *wcsnicmp*, *wmemcmp*.

bugprone-swapped-arguments

Finds potentially swapped arguments by looking at implicit conversions.

bugprone-terminating-continue

Detects **do while** loops with a condition always evaluating to false that have a **continue** statement, as this **continue** terminates the loop effectively.

```
void f() {
do {
  // some code
  continue; // terminating continue
  // some other code
} while(false);
```

bugprone-throw-keyword-missing

Warns about a potentially missing **throw** keyword. If a temporary object is created, but the object's type derives from (or is the same as) a class that has 'EXCEPTION', 'Exception' or 'exception' in its name, we can assume that the programmer's intention was to throw that object.

Example:

```
void f(int i) {
  if (i < 0) {
    // Exception is created but is not thrown.
    std::runtime_error("Unexpected argument");
  }
}
```

bugprone-too-small-loop-variable

Detects those **for** loops that have a loop variable with a "too small" type which means this type can't represent all values which are part of the iteration range.

```
int main() {
```

```
  long size = 294967296l;
  for (short i = 0; i < size; ++i) {}
}
```

This **for** loop is an infinite loop because the **short** type can't represent all values in the **[0..size]** interval.

In a real use case size means a container's size which depends on the user input.

```
int doSomething(const std::vector& items) {
  for (short i = 0; i < items.size(); ++i) {}
}
```

This algorithm works for a small amount of objects, but will lead to freeze for a larger user input.

MagnitudeBitsUpperLimit

Upper limit for the magnitude bits of the loop variable. If it's set the check filters out those catches in which the loop variable's type has more magnitude bits as the specified upper limit. The default value is 16. For example, if the user sets this option to 31 (bits), then a 32-bit **unsigned int** is ignored by the check, however a 32-bit **int** is not (A 32-bit **signed int** has 31 magnitude bits).

```
int main() {
  long size = 294967296l;
  for (unsigned i = 0; i < size; ++i) {} // no warning with MagnitudeBitsUpperLimit = 31 on a system where unsign
  for (int i = 0; i < size; ++i) {} // warning with MagnitudeBitsUpperLimit = 31 on a system where int is 32-bit
}
```

bugprone-unchecked-optional-access

Note: This check uses a flow-sensitive static analysis to produce its results. Therefore, it may be more resource intensive (RAM, CPU) than the average clang-tidy check.

This check identifies unsafe accesses to values contained in **std::optional<T>**, **absl::optional<T>**, or **base::Optional<T>** objects. Below we will refer to all these types collectively as **optional<T>**.

An access to the value of an **optional<T>** occurs when one of its **value**, **operator***, or **operator->** member functions is invoked. To align with common misconceptions, the check considers these member functions as equivalent, even though there are subtle differences related to exceptions versus undefined behavior. See go/optional-style-recommendations for more information on that topic.

An access to the value of an **optional<T>** is considered safe if and only if code in the local scope (for example, a function body) ensures that the **optional<T>** has a value in all possible execution paths that can reach the access. That should happen either through an explicit check, using the **optional<T>::has_value** member function, or by constructing the **optional<T>** in a way that shows that it unambiguously holds a value (e.g using **std::make_optional** which always returns a populated **std::optional<T>**).

Below we list some examples, starting with unsafe optional access patterns, followed by safe access patterns.

Unsafe access patterns

Access the value without checking if it exists

The check flags accesses to the value that are not locally guarded by existence check:

```
void f(std::optional<int> opt) {
  use(*opt); // unsafe: it is unclear whether ‘opt‘ has a value.
}
```

Access the value in the wrong branch

The check is aware of the state of an optional object in different branches of the code. For example:

```
void f(std::optional<int> opt) {
  if (opt.has_value()) {
  } else {
    use(opt.value()); // unsafe: it is clear that ‘opt‘ does *not* have a value.
  }
}
```

Assume a function result to be stable

The check is aware that function results might not be stable. That is, consecutive calls to the same function might return different values. For example:

```
void f(Foo foo) {
  if (foo.opt().has_value()) {
    use(*foo.opt()); // unsafe: it is unclear whether ‘foo.opt()‘ has a value.
  }
}
```

Rely on invariants of uncommon APIs

The check is unaware of invariants of uncommon APIs. For example:

```
void f(Foo foo) {
  if (foo.HasProperty("bar")) {
    use(*foo.GetProperty("bar")); // unsafe: it is unclear whether ‘foo.GetProperty("bar")‘ has a value.
  }
}
```

Check if a value exists, then pass the optional to another function

The check relies on local reasoning. The check and value access must both happen in the same function. An access is considered unsafe even if the caller of the function performing the access ensures that the optional has a value. For example:

```
void g(std::optional<int> opt) {
  use(*opt); // unsafe: it is unclear whether ‘opt‘ has a value.
}

void f(std::optional<int> opt) {
  if (opt.has_value()) {
    g(opt);
  }
}
```

Safe access patterns

Check if a value exists, then access the value

The check recognizes all straightforward ways for checking if a value exists and accessing the value contained in an optional object. For example:

```
void f(std::optional<int> opt) {
  if (opt.has_value()) {
    use(*opt);
  }
}
```

Check if a value exists, then access the value from a copy

The criteria that the check uses is semantic, not syntactic. It recognizes when a copy of the optional object being accessed is known to have a value. For example:

```
void f(std::optional<int> opt1) {
  if (opt1.has_value()) {
    std::optional<int> opt2 = opt1;
    use(*opt2);
```

```
  }
}
```

Ensure that a value exists using common macros

The check is aware of common macros like **CHECK**, **DCHECK**, and **ASSERT_THAT**. Those can be used to ensure that an optional object has a value. For example:

```
void f(std::optional<int> opt) {
  DCHECK(opt.has_value());
  use(*opt);
}
```

Ensure that a value exists, then access the value in a correlated branch

The check is aware of correlated branches in the code and can figure out when an optional object is ensured to have a value on all execution paths that lead to an access. For example:

```
void f(std::optional<int> opt) {
  bool safe = false;
  if (opt.has_value() && SomeOtherCondition()) {
    safe = true;
  }
  // ... more code...
  if (safe) {
    use(*opt);
  }
}
```

Stabilize function results

Since function results are not assumed to be stable across calls, it is best to store the result of the function call in a local variable and use that variable to access the value. For example:

```
void f(Foo foo) {
  if (const auto& foo_opt = foo.opt(); foo_opt.has_value()) {
    use(*foo_opt);
  }
}
```

Do not rely on uncommon-API invariants

When uncommon APIs guarantee that an optional has contents, do not rely on it -- instead, check explicitly that the optional object has a value. For example:

```
void f(Foo foo) {
  if (const auto& property = foo.GetProperty("bar")) {
    use(*property);
  }
}
```

instead of the *HasProperty*, *GetProperty* pairing we saw above.

Do not rely on caller-performed checks

If you know that all of a function's callers have checked that an optional argument has a value, either change the function to take the value directly or check the optional again in the local scope of the callee. For example:

```
void g(int val) {
  use(val);
}

void f(std::optional<int> opt) {
  if (opt.has_value()) {
    g(*opt);
  }
}
```

and

```
struct S {
  std::optional<int> opt;
  int x;
};

void g(const S &s) {
  if (s.opt.has_value() && s.x > 10) {
    use(*s.opt);
}

void f(S s) {
  if (s.opt.has_value()) {
    g(s);
  }
}
```

Additional notes

Aliases created via using declarations

The check is aware of aliases of optional types that are created via **using** declarations. For example:

```
using OptionalInt = std::optional<int>;

void f(OptionalInt opt) {
  use(opt.value()); // unsafe: it is unclear whether ‘opt‘ has a value.
}
```

Lambdas

The check does not currently report unsafe optional acceses in lambdas. A future version will expand the scope to lambdas, following the rules outlined above. It is best to follow the same principles when using optionals in lambdas.

bugprone-undefined-memory-manipulation

Finds calls of memory manipulation functions **memset()**, **memcpy()** and **memmove()** on not TriviallyCopyable objects resulting in undefined behavior.

bugprone-undelegated-constructor

Finds creation of temporary objects in constructors that look like a function call to another constructor of the same class.

The user most likely meant to use a delegating constructor or base class initializer.

bugprone-unhandled-exception-at-new

Finds calls to **new** with missing exception handler for **std::bad_alloc**.

Calls to **new** may throw exceptions of type **std::bad_alloc** that should be handled. Alternatively, the nonthrowing form of **new** can be used. The check verifies that the exception is handled in the function that calls **new**.

If a nonthrowing version is used or the exception is allowed to propagate out of the function no warning is generated.

The exception handler is checked if it catches a **std::bad_alloc** or **std::exception** exception type, or all exceptions (catch-all). The check assumes that any user-defined **operator new** is either **noexcept** or may throw an exception of type **std::bad_alloc** (or one derived from it). Other exception class types are not taken into account.

```
int *f() noexcept {
  int *p = new int[1000]; // warning: missing exception handler for allocation failure at 'new'
  // ...
  return p;
}

int *f1() { // not 'noexcept'
  int *p = new int[1000]; // no warning: exception can be handled outside
                          // of this function
  // ...
  return p;
}

int *f2() noexcept {
  try {
    int *p = new int[1000]; // no warning: exception is handled
    // ...
    return p;
  } catch (std::bad_alloc &) {
    // ...
  }
  // ...
}

int *f3() noexcept {
  int *p = new (std::nothrow) int[1000]; // no warning: "nothrow" is used
  // ...
  return p;
}
```

bugprone-unhandled-self-assignment

cert-oop54-cpp redirects here as an alias for this check. For the CERT alias, the *WarnOnlyIfThisHasSuspiciousField* option is set to *false*.

Finds user-defined copy assignment operators which do not protect the code against self-assignment either by checking self-assignment explicitly or using the copy-and-swap or the copy-and-move method.

By default, this check searches only those classes which have any pointer or C array field to avoid false positives. In case of a pointer or a C array, it's likely that self-copy assignment breaks the object if the

copy assignment operator was not written with care.

See also: *OOP54-CPP. Gracefully handle self-copy assignment*

A copy assignment operator must prevent that self-copy assignment ruins the object state. A typical use case is when the class has a pointer field and the copy assignment operator first releases the pointed object and then tries to assign it:

```
class T {
int* p;

public:
  T(const T &rhs) : p(rhs.p ? new int(*rhs.p) : nullptr) {}
  ~T() { delete p; }

  // ...

  T& operator=(const T &rhs) {
    delete p;
    p = new int(*rhs.p);
    return *this;
  }
};
```

There are two common C++ patterns to avoid this problem. The first is the self-assignment check:

```
class T {
int* p;

public:
  T(const T &rhs) : p(rhs.p ? new int(*rhs.p) : nullptr) {}
  ~T() { delete p; }

  // ...

  T& operator=(const T &rhs) {
    if(this == &rhs)
      return *this;
```

```
    delete p;
    p = new int(*rhs.p);
    return *this;
  }
};
```

The second one is the copy-and-swap method when we create a temporary copy (using the copy constructor) and then swap this temporary object with **this**:

```
class T {
int* p;

public:
  T(const T &rhs) : p(rhs.p ? new int(*rhs.p) : nullptr) {}
  ~T() { delete p; }

  // ...

  void swap(T &rhs) {
    using std::swap;
    swap(p, rhs.p);
  }

  T& operator=(const T &rhs) {
    T(rhs).swap(*this);
    return *this;
  }
};
```

There is a third pattern which is less common. Let’s call it the copy-and-move method when we create a temporary copy (using the copy constructor) and then move this temporary object into **this** (needs a move assignment operator):

```
class T {
int* p;

public:
  T(const T &rhs) : p(rhs.p ? new int(*rhs.p) : nullptr) {}
  ~T() { delete p; }
```

```
  // ...

  T& operator=(const T &rhs) {
    T t = rhs;
    *this = std::move(t);
    return *this;
  }

  T& operator=(T &&rhs) {
    p = rhs.p;
    rhs.p = nullptr;
    return *this;
  }
};
```

WarnOnlyIfThisHasSuspiciousField
When *true*, the check will warn only if the container class of the copy assignment operator has any suspicious fields (pointer or C array). This option is set to *true* by default.

bugprone-unused-raii
Finds temporaries that look like RAII objects.

The canonical example for this is a scoped lock.

```
{
  scoped_lock(&global_mutex);
  critical_section();
}
```

The destructor of the scoped_lock is called before the **critical_section** is entered, leaving it unprotected.

We apply a number of heuristics to reduce the false positive count of this check:

- Ignore code expanded from macros. Testing frameworks make heavy use of this.
- Ignore types with trivial destructors. They are very unlikely to be RAII objects and there's no difference when they are deleted.
- Ignore objects at the end of a compound statement (doesn't change behavior).

- Ignore objects returned from a call.

bugprone-unused-return-value
Warns on unused function return values. The checked functions can be configured.

Options

CheckedFunctions
Semicolon-separated list of functions to check. The function is checked if the name and scope matches, with any arguments. By default the following functions are checked: **std::async, std::launder, std::remove, std::remove_if, std::unique, std::unique_ptr::release, std::basic_string::empty, std::vector::empty, std::back_inserter, std::distance, std::find, std::find_if, std::inserter, std::lower_bound, std::make_pair, std::map::count, std::map::find, std::map::lower_bound, std::multimap::equal_range, std::multimap::upper_bound, std::set::count, std::set::find, std::setfill, std::setprecision, std::setw, std::upper_bound, std::vector::at, bsearch, ferror, feof, isalnum, isalpha, isblank, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, iswalnum, iswprint, iswspace, isxdigit, memchr, memcmp, strcmp, strcoll, strncmp, strpbrk, strrchr, strspn, strstr, wcscmp, access, bind, connect, difftime, dlsym, fnmatch, getaddrinfo, getopt, htonl, htons, iconv_open, inet_addr, isascii, isatty, mmap, newlocale, openat, pathconf, pthread_equal, pthread_getspecific, pthread_mutex_trylock, readdir, readlink, recvmsg, regexec, scandir, semget, setjmp, shm_open, shmget, sigismember, strcasecmp, strsignal, ttyname**

- **std::async()**. Not using the return value makes the call synchronous.

- **std::launder()**. Not using the return value usually means that the function interface was misunderstood by the programmer. Only the returned pointer is "laundered", not the argument.

- **std::remove()**, **std::remove_if()** and **std::unique()**. The returned iterator indicates the boundary between elements to keep and elements to be removed. Not using the return value means that the information about which elements to remove is lost.

- **std::unique_ptr::release()**. Not using the return value can lead to resource leaks if the same pointer isn’t stored anywhere else. Often, ignoring the **release()** return value indicates that the programmer confused the function with **reset()**.

- **std::basic_string::empty()** and **std::vector::empty()**. Not using the return value often indicates that the programmer confused the function with **clear()**.

cert-err33-c is an alias of this check that checks a fixed and large set of standard library

functions.

bugprone-use-after-move

Warns if an object is used after it has been moved, for example:

```
std::string str = "Hello, world!\n";
std::vector<std::string> messages;
messages.emplace_back(std::move(str));
std::cout << str;
```

The last line will trigger a warning that **str** is used after it has been moved.

The check does not trigger a warning if the object is reinitialized after the move and before the use. For example, no warning will be output for this code:

```
messages.emplace_back(std::move(str));
str = "Greetings, stranger!\n";
std::cout << str;
```

Subsections below explain more precisely what exactly the check considers to be a move, use, and reinitialization.

The check takes control flow into account. A warning is only emitted if the use can be reached from the move. This means that the following code does not produce a warning:

```
if (condition) {
  messages.emplace_back(std::move(str));
} else {
  std::cout << str;
}
```

On the other hand, the following code does produce a warning:

```
for (int i = 0; i < 10; ++i) {
  std::cout << str;
  messages.emplace_back(std::move(str));
}
```

(The use-after-move happens on the second iteration of the loop.)

In some cases, the check may not be able to detect that two branches are mutually exclusive. For example (assuming that **i** is an int):

```
if (i == 1) {
  messages.emplace_back(std::move(str));
}
if (i == 2) {
  std::cout << str;
}
```

In this case, the check will erroneously produce a warning, even though it is not possible for both the move and the use to be executed. More formally, the analysis is *flow-sensitive but not path-sensitive*.

Silencing erroneous warnings

An erroneous warning can be silenced by reinitializing the object after the move:

```
if (i == 1) {
  messages.emplace_back(std::move(str));
  str = "";
}
if (i == 2) {
  std::cout << str;
}
```

If you want to avoid the overhead of actually reinitializing the object, you can create a dummy function that causes the check to assume the object was reinitialized:

```
template <class T>
void IS_INITIALIZED(T&) {}
```

You can use this as follows:

```
if (i == 1) {
  messages.emplace_back(std::move(str));
}
if (i == 2) {
  IS_INITIALIZED(str);
  std::cout << str;
}
```

The check will not output a warning in this case because passing the object to a function as a non-const pointer or reference counts as a reinitialization (see section *Reinitialization* below).

Unsequenced moves, uses, and reinitializations

In many cases, C++ does not make any guarantees about the order in which sub-expressions of a statement are evaluated. This means that in code like the following, it is not guaranteed whether the use will happen before or after the move:

```
void f(int i, std::vector<int> v);
std::vector<int> v = { 1, 2, 3 };
f(v[1], std::move(v));
```

In this kind of situation, the check will note that the use and move are unsequenced.

The check will also take sequencing rules into account when reinitializations occur in the same statement as moves or uses. A reinitialization is only considered to reinitialize a variable if it is guaranteed to be evaluated after the move and before the use.

Move

The check currently only considers calls of **std::move** on local variables or function parameters. It does not check moves of member variables or global variables.

Any call of **std::move** on a variable is considered to cause a move of that variable, even if the result of **std::move** is not passed to an rvalue reference parameter.

This means that the check will flag a use-after-move even on a type that does not define a move constructor or move assignment operator. This is intentional. Developers may use **std::move** on such a type in the expectation that the type will add move semantics in the future. If such a **std::move** has the potential to cause a use-after-move, we want to warn about it even if the type does not implement move semantics yet.

Furthermore, if the result of **std::move** *is* passed to an rvalue reference parameter, this will always be considered to cause a move, even if the function that consumes this parameter does not move from it, or if it does so only conditionally. For example, in the following situation, the check will assume that a move always takes place:

```
std::vector<std::string> messages;
void f(std::string &&str) {
  // Only remember the message if it isn't empty.
  if (!str.empty()) {
```

```
    messages.emplace_back(std::move(str));
  }
}
std::string str = "";
f(std::move(str));
```

The check will assume that the last line causes a move, even though, in this particular case, it does not. Again, this is intentional.

There is one special case: A call to **std::move** inside a **try_emplace** call is conservatively assumed not to move. This is to avoid spurious warnings, as the check has no way to reason about the **bool** returned by **try_emplace**.

When analyzing the order in which moves, uses and reinitializations happen (see section *Unsequenced moves, uses, and reinitializations*), the move is assumed to occur in whichever function the result of the **std::move** is passed to.

Use

Any occurrence of the moved variable that is not a reinitialization (see below) is considered to be a use.

An exception to this are objects of type **std::unique_ptr**, **std::shared_ptr** and **std::weak_ptr**, which have defined move behavior (objects of these classes are guaranteed to be empty after they have been moved from). Therefore, an object of these classes will only be considered to be used if it is dereferenced, i.e. if **operator***, **operator->** or **operator[]** (in the case of **std::unique_ptr<T []>**) is called on it.

If multiple uses occur after a move, only the first of these is flagged.

Reinitialization

The check considers a variable to be reinitialized in the following cases:

- The variable occurs on the left-hand side of an assignment.
- The variable is passed to a function as a non-const pointer or non-const lvalue reference. (It is assumed that the variable may be an out-parameter for the function.)
- **clear()** or **assign()** is called on the variable and the variable is of one of the standard container types **basic_string**, **vector**, **deque**, **forward_list**, **list**, **set**, **map**, **multiset**, **multimap**, **unordered_set**, **unordered_map**, **unordered_multiset**, **unordered_multimap**.
- **reset()** is called on the variable and the variable is of type **std::unique_ptr**, **std::shared_ptr** or

std::weak_ptr.

- A member function marked with the **[[clang::reinitializes]]** attribute is called on the variable.

  If the variable in question is a struct and an individual member variable of that struct is written to, the check does not consider this to be a reinitialization -- even if, eventually, all member variables of the struct are written to. For example:

```
struct S {
  std::string str;
  int i;
};
S s = { "Hello, world!\n", 42 };
S s_other = std::move(s);
s.str = "Lorem ipsum";
s.i = 99;
```

  The check will not consider **s** to be reinitialized after the last line; instead, the line that assigns to **s.str** will be flagged as a use-after-move. This is intentional as this pattern of reinitializing a struct is error-prone. For example, if an additional member variable is added to **S**, it is easy to forget to add the reinitialization for this additional member. Instead, it is safer to assign to the entire struct in one go, and this will also avoid the use-after-move warning.

bugprone-virtual-near-miss

Warn if a function is a near miss (i.e. the name is very similar and the function signature is the same) to a virtual function from a base class.

Example:

```
struct Base {
  virtual void func();
};

struct Derived : Base {
  virtual void funk();
  // warning: 'Derived::funk' has a similar name and the same signature as virtual method 'Base::func'; did you me
};
```

cert-con36-c

The cert-con36-c check is an alias, please see *bugprone-spuriously-wake-up-functions* for more

information.

cert-con54-cpp
The cert-con54-cpp check is an alias, please see *bugprone-spuriously-wake-up-functions* for more information.

cert-dcl03-c
The cert-dcl03-c check is an alias, please see *misc-static-assert* for more information.

cert-dcl16-c
The cert-dcl16-c check is an alias, please see *readability-uppercase-literal-suffix* for more information.

cert-dcl21-cpp
This check flags postfix **operator++** and **operator--** declarations if the return type is not a const object. This also warns if the return type is a reference type.

The object returned by a postfix increment or decrement operator is supposed to be a snapshot of the object's value prior to modification. With such an implementation, any modifications made to the resulting object from calling operator++(int) would be modifying a temporary object. Thus, such an implementation of a postfix increment or decrement operator should instead return a const object, prohibiting accidental mutation of a temporary object. Similarly, it is unexpected for the postfix operator to return a reference to its previous state, and any subsequent modifications would be operating on a stale object.

This check corresponds to the CERT C++ Coding Standard recommendation DCL21-CPP. Overloaded postfix increment and decrement operators should return a const object. However, all of the CERT recommendations have been removed from public view, and so their justification for the behavior of this check requires an account on their wiki to view.

cert-dcl37-c
The cert-dcl37-c check is an alias, please see *bugprone-reserved-identifier* for more information.

cert-dcl50-cpp
This check flags all function definitions (but not declarations) of C-style variadic functions.

This check corresponds to the CERT C++ Coding Standard rule *DCL50-CPP. Do not define a C-style variadic function*.

cert-dcl51-cpp
The cert-dcl51-cpp check is an alias, please see *bugprone-reserved-identifier* for more information.

cert-dcl54-cpp

The cert-dcl54-cpp check is an alias, please see *misc-new-delete-overloads* for more information.

cert-dcl58-cpp

Modification of the **std** or **posix** namespace can result in undefined behavior. This check warns for such modifications. The **std** (or **posix**) namespace is allowed to be extended with (class or function) template specializations that depend on an user-defined type (a type that is not defined in the standard system headers).

The check detects the following (user provided) declarations in namespace **std** or **posix**:

- Anything that is not a template specialization.
- Explicit specializations of any standard library function template or class template, if it does not have any user-defined type as template argument.
- Explicit specializations of any member function of a standard library class template.
- Explicit specializations of any member function template of a standard library class or class template.
- Explicit or partial specialization of any member class template of a standard library class or class template.

Examples:

```
namespace std {
  int x; // warning: modification of 'std' namespace can result in undefined behavior [cert-dcl58-cpp]
}

namespace posix::a { // warning: modification of 'posix' namespace can result in undefined behavior
}

template <>
struct ::std::hash<long> { // warning: modification of 'std' namespace can result in undefined behavior
  unsigned long operator()(const long &K) const {
    return K;
  }
};
```

```
struct MyData { long data; };

template <>
struct ::std::hash<MyData> { // no warning: specialization with user-defined type
  unsigned long operator()(const MyData &K) const {
    return K.data;
  }
};

namespace std {
  template <>
  void swap<bool>(bool &a, bool &b); // warning: modification of 'std' namespace can result in undefined behavio

  template <>
  bool less<void>::operator()<MyData &&, MyData &&>(MyData &&, MyData &&) const { // warning: modifica
    return true;
  }
}
```

This check corresponds to the CERT C++ Coding Standard rule *DCL58-CPP. Do not modify the standard namespaces*.

cert-dcl59-cpp

The cert-dcl59-cpp check is an alias, please see *google-build-namespaces* for more information.

cert-env33-c

This check flags calls to **system()**, **popen()**, and **_popen()**, which execute a command processor. It does not flag calls to **system()** with a null pointer argument, as such a call checks for the presence of a command processor but does not actually attempt to execute a command.

This check corresponds to the CERT C Coding Standard rule *ENV33-C. Do not call system()*.

cert-err09-cpp

The cert-err09-cpp check is an alias, please see *misc-throw-by-value-catch-by-reference* for more information.

This check corresponds to the CERT C++ Coding Standard recommendation ERR09-CPP. Throw anonymous temporaries. However, all of the CERT recommendations have been removed from public view, and so their justification for the behavior of this check requires an account on their wiki to view.

cert-err33-c

Warns on unused function return values. Many of the standard library functions return a value that indicates if the call was successful. Ignoring the returned value can cause unexpected behavior if an error has occured. The following functions are checked:

- aligned_alloc()
- asctime_s()
- at_quick_exit()
- atexit()
- bsearch()
- bsearch_s()
- btowc()
- c16rtomb()
- c32rtomb()
- calloc()
- clock()
- cnd_broadcast()
- cnd_init()
- cnd_signal()
- cnd_timedwait()
- cnd_wait()
- ctime_s()
- fclose()

- fflush()
- fgetc()
- fgetpos()
- fgets()
- fgetwc()
- fopen()
- fopen_s()
- fprintf()
- fprintf_s()
- fputc()
- fputs()
- fputwc()
- fputws()
- fread()
- freopen()
- freopen_s()
- fscanf()
- fscanf_s()
- fseek()
- fsetpos()

- ftell()
- fwprintf()
- fwprintf_s()
- fwrite()
- fwscanf()
- fwscanf_s()
- getc()
- getchar()
- getenv()
- getenv_s()
- gets_s()
- getwc()
- getwchar()
- gmtime()
- gmtime_s()
- localtime()
- localtime_s()
- malloc()
- mbrtoc16()
- mbrtoc32()

- mbsrtowcs()
- mbsrtowcs_s()
- mbstowcs()
- mbstowcs_s()
- memchr()
- mktime()
- mtx_init()
- mtx_lock()
- mtx_timedlock()
- mtx_trylock()
- mtx_unlock()
- printf_s()
- putc()
- putwc()
- raise()
- realloc()
- remove()
- rename()
- setlocale()
- setvbuf()

- scanf()
- scanf_s()
- signal()
- snprintf()
- snprintf_s()
- sprintf()
- sprintf_s()
- sscanf()
- sscanf_s()
- strchr()
- strerror_s()
- strftime()
- strpbrk()
- strrchr()
- strstr()
- strtod()
- strtof()
- strtoimax()
- strtok()
- strtok_s()

- strtol()
- strtold()
- strtoll()
- strtoumax()
- strtoul()
- strtoull()
- strxfrm()
- swprintf()
- swprintf_s()
- swscanf()
- swscanf_s()
- thrd_create()
- thrd_detach()
- thrd_join()
- thrd_sleep()
- time()
- timespec_get()
- tmpfile()
- tmpfile_s()
- tmpnam()

- tmpnam_s()
- tss_create()
- tss_get()
- tss_set()
- ungetc()
- ungetwc()
- vfprintf()
- vfprintf_s()
- vfscanf()
- vfscanf_s()
- vfwprintf()
- vfwprintf_s()
- vfwscanf()
- vfwscanf_s()
- vprintf_s()
- vscanf()
- vscanf_s()
- vsnprintf()
- vsnprintf_s()
- vsprintf()

- vsprintf_s()
- vsscanf()
- vsscanf_s()
- vswprintf()
- vswprintf_s()
- vswscanf()
- vswscanf_s()
- vwprintf_s()
- vwscanf()
- vwscanf_s()
- wcrtomb()
- wcschr()
- wcsftime()
- wcspbrk()
- wcsrchr()
- wcsrtombs()
- wcsrtombs_s()
- wcsstr()
- wcstod()
- wcstof()

- wcstoimax()
- wcstok()
- wcstok_s()
- wcstol()
- wcstold()
- wcstoll()
- wcstombs()
- wcstombs_s()
- wcstoumax()
- wcstoul()
- wcstoull()
- wcsxfrm()
- wctob()
- wctrans()
- wctype()
- wmemchr()
- wprintf_s()
- wscanf()
- wscanf_s()

This check is an alias of check *bugprone-unused-return-value* with a fixed set of functions.

The check corresponds to a part of CERT C Coding Standard rule *ERR33-C. Detect and handle standard library errors*. The list of checked functions is taken from the rule, with following exception:

- The check can not differentiate if a function is called with **NULL** argument. Therefore the following functions are not checked: **mblen**, **mbrlen**, **mbrtowc**, **mbtowc**, **wctomb**, **wctomb_s**

cert-err34-c

This check flags calls to string-to-number conversion functions that do not verify the validity of the conversion, such as **atoi()** or **scanf()**. It does not flag calls to **strtol()**, or other, related conversion functions that do perform better error checking.

```
#include <stdlib.h>

void func(const char *buff) {
  int si;

  if (buff) {
    si = atoi(buff); /* 'atoi' used to convert a string to an integer, but function will
                        not report conversion errors; consider using 'strtol' instead. */
  } else {
    /* Handle error */
  }
}
```

This check corresponds to the CERT C Coding Standard rule *ERR34-C. Detect errors when converting a string to a number*.

cert-err52-cpp

This check flags all call expressions involving **setjmp()** and **longjmp()**.

This check corresponds to the CERT C++ Coding Standard rule *ERR52-CPP. Do not use setjmp() or longjmp()*.

cert-err58-cpp

This check flags all **static** or **thread_local** variable declarations where the initializer for the object may throw an exception.

This check corresponds to the CERT C++ Coding Standard rule *ERR58-CPP. Handle all exceptions thrown before main() begins executing*.

cert-err60-cpp
This check flags all throw expressions where the exception object is not nothrow copy constructible.

This check corresponds to the CERT C++ Coding Standard rule *ERR60-CPP. Exception objects must be nothrow copy constructible*.

cert-err61-cpp
The cert-err61-cpp check is an alias, please see *misc-throw-by-value-catch-by-reference* for more information.

cert-exp42-c
The cert-exp42-c check is an alias, please see *bugprone-suspicious-memory-comparison* for more information.

cert-fio38-c
The cert-fio38-c check is an alias, please see *misc-non-copyable-objects* for more information.

cert-flp30-c
This check flags **for** loops where the induction expression has a floating-point type.

This check corresponds to the CERT C Coding Standard rule *FLP30-C. Do not use floating-point variables as loop counters*.

cert-flp37-c
The cert-flp37-c check is an alias, please see *bugprone-suspicious-memory-comparison* for more information.

cert-mem57-cpp
This check flags uses of default **operator new** where the type has extended alignment (an alignment greater than the fundamental alignment). (The default **operator new** is guaranteed to provide the correct alignment if the requested alignment is less or equal to the fundamental alignment). Only cases are detected (by design) where the **operator new** is not user-defined and is not a placement new (the reason is that in these cases we assume that the user provided the correct memory allocation).

This check corresponds to the CERT C++ Coding Standard rule *MEM57-CPP. Avoid using default operator new for over-aligned types*.

cert-msc30-c
The cert-msc30-c check is an alias, please see *cert-msc50-cpp* for more information.

cert-msc32-c

The cert-msc32-c check is an alias, please see *cert-msc51-cpp* for more information.

cert-msc50-cpp

Pseudorandom number generators use mathematical algorithms to produce a sequence of numbers with good statistical properties, but the numbers produced are not genuinely random. The **std::rand()** function takes a seed (number), runs a mathematical operation on it and returns the result. By manipulating the seed the result can be predictable. This check warns for the usage of **std::rand()**.

cert-msc51-cpp

This check flags all pseudo-random number engines, engine adaptor instantiations and **srand()** when initialized or seeded with default argument, constant expression or any user-configurable type. Pseudo-random number engines seeded with a predictable value may cause vulnerabilities e.g. in security protocols. This is a CERT security rule, see *MSC51-CPP. Ensure your random number generator is properly seeded* and *MSC32-C. Properly seed pseudorandom number generators*.

Examples:

```
void foo() {
  std::mt19937 engine1; // Diagnose, always generate the same sequence
  std::mt19937 engine2(1); // Diagnose
  engine1.seed(); // Diagnose
  engine2.seed(1); // Diagnose

  std::time_t t;
  engine1.seed(std::time(&t)); // Diagnose, system time might be controlled by user

  int x = atoi(argv[1]);
  std::mt19937 engine3(x);  // Will not warn
}
```

Options

DisallowedSeedTypes

A comma-separated list of the type names which are disallowed. Default values are **time_t**, **std::time_t**.

cert-oop11-cpp

The cert-oop11-cpp check is an alias, please see *performance-move-constructor-init* for more information.

This check corresponds to the CERT C++ Coding Standard recommendation OOP11-CPP. Do not copy-initialize members or base classes from a move constructor. However, all of the CERT recommendations have been removed from public view, and so their justification for the behavior of this check requires an account on their wiki to view.

cert-oop54-cpp

The cert-oop54-cpp check is an alias, please see *bugprone-unhandled-self-assignment* for more information.

cert-oop57-cpp

Flags use of the *C* standard library functions **memset**, **memcpy** and **memcmp** and similar derivatives on non-trivial types.

Options

MemSetNames

Specify extra functions to flag that act similarly to **memset**. Specify names in a semicolon delimited list. Default is an empty string. The check will detect the following functions: *memset*, *std::memset*.

MemCpyNames

Specify extra functions to flag that act similarly to **memcpy**. Specify names in a semicolon delimited list. Default is an empty string. The check will detect the following functions: *std::memcpy*, *memcpy*, *std::memmove*, *memmove*, *std::strcpy*, *strcpy*, *memccpy*, *stpncpy*, *strncpy*.

MemCmpNames

Specify extra functions to flag that act similarly to **memcmp**. Specify names in a semicolon delimited list. Default is an empty string. The check will detect the following functions: *std::memcmp*, *memcmp*, *std::strcmp*, *strcmp*, *strncmp*.

This check corresponds to the CERT C++ Coding Standard rule *OOP57-CPP. Prefer special member functions and overloaded operators to C Standard Library functions*.

cert-oop58-cpp

Finds assignments to the copied object and its direct or indirect members in copy constructors and copy assignment operators.

This check corresponds to the CERT C Coding Standard rule *OOP58-CPP. Copy operations must not mutate the source object*.

cert-pos44-c
The cert-pos44-c check is an alias, please see *bugprone-bad-signal-to-kill-thread* for more information.

cert-pos47-c
The cert-pos47-c check is an alias, please see *concurrency-thread-canceltype-asynchronous* for more information.

cert-sig30-c
The cert-sig30-c check is an alias, please see *bugprone-signal-handler* for more information.

cert-str34-c
The cert-str34-c check is an alias, please see *bugprone-signed-char-misuse* for more information.

clang-analyzer-core.CallAndMessage
The clang-analyzer-core.CallAndMessage check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-core.DivideZero
The clang-analyzer-core.DivideZero check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-core.DynamicTypePropagation
Generate dynamic type information

clang-analyzer-core.NonNullParamChecker
The clang-analyzer-core.NonNullParamChecker check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-core.NullDereference
The clang-analyzer-core.NullDereference check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-core.StackAddressEscape
The clang-analyzer-core.StackAddressEscape check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-core.UndefinedBinaryOperatorResult
The clang-analyzer-core.UndefinedBinaryOperatorResult check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-core.VLASize
The clang-analyzer-core.VLASize check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-core.uninitialized.ArraySubscript
The clang-analyzer-core.uninitialized.ArraySubscript check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-core.uninitialized.Assign
The clang-analyzer-core.uninitialized.Assign check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-core.uninitialized.Branch
The clang-analyzer-core.uninitialized.Branch check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-core.uninitialized.CapturedBlockVariable
Check for blocks that capture uninitialized values

clang-analyzer-core.uninitialized.UndefReturn
The clang-analyzer-core.uninitialized.UndefReturn check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-cplusplus.InnerPointer
Check for inner pointers of C++ containers used after re/deallocation

clang-analyzer-cplusplus.Move
The clang-analyzer-cplusplus.Move check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-cplusplus.NewDelete
The clang-analyzer-cplusplus.NewDelete check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-cplusplus.NewDeleteLeaks
The clang-analyzer-cplusplus.NewDeleteLeaks check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-deadcode.DeadStores
The clang-analyzer-deadcode.DeadStores check is an alias, please see *Clang Static Analyzer Available*

Checkers for more information.

clang-analyzer-nullability.NullPassedToNonnull
The clang-analyzer-nullability.NullPassedToNonnull check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-nullability.NullReturnedFromNonnull
The clang-analyzer-nullability.NullReturnedFromNonnull check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-nullability.NullableDereferenced
The clang-analyzer-nullability.NullableDereferenced check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-nullability.NullablePassedToNonnull
The clang-analyzer-nullability.NullablePassedToNonnull check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-nullability.NullableReturnedFromNonnull
Warns when a nullable pointer is returned from a function that has _Nonnull return type.

clang-analyzer-optin.cplusplus.UninitializedObject
The clang-analyzer-optin.cplusplus.UninitializedObject check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-optin.cplusplus.VirtualCall
The clang-analyzer-optin.cplusplus.VirtualCall check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-optin.mpi.MPI-Checker
The clang-analyzer-optin.mpi.MPI-Checker check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-optin.osx.OSObjectCStyleCast
Checker for C-style casts of OSObjects

clang-analyzer-optin.osx.cocoa.localizability.EmptyLocalizationContextChecker
The clang-analyzer-optin.osx.cocoa.localizability.EmptyLocalizationContextChecker check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-optin.osx.cocoa.localizability.NonLocalizedStringChecker
The clang-analyzer-optin.osx.cocoa.localizability.NonLocalizedStringChecker check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-optin.performance.GCDAntipattern
Check for performance anti-patterns when using Grand Central Dispatch

clang-analyzer-optin.performance.Padding
Check for excessively padded structs.

clang-analyzer-optin.portability.UnixAPI
Finds implementation-defined behavior in UNIX/Posix functions

clang-analyzer-osx.API
The clang-analyzer-osx.API check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-osx.MIG
Find violations of the Mach Interface Generator calling convention

clang-analyzer-osx.NumberObjectConversion
Check for erroneous conversions of objects representing numbers into numbers

clang-analyzer-osx.OSObjectRetainCount
Check for leaks and improper reference count management for OSObject

clang-analyzer-osx.ObjCProperty
Check for proper uses of Objective-C properties

clang-analyzer-osx.SecKeychainAPI
The clang-analyzer-osx.SecKeychainAPI check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-osx.cocoa.AtSync
The clang-analyzer-osx.cocoa.AtSync check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-osx.cocoa.AutoreleaseWrite
Warn about potentially crashing writes to autoreleasing objects from different autoreleasing pools in Objective-C

clang-analyzer-osx.cocoa.ClassRelease
The clang-analyzer-osx.cocoa.ClassRelease check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-osx.cocoa.Dealloc
The clang-analyzer-osx.cocoa.Dealloc check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-osx.cocoa.IncompatibleMethodTypes
The clang-analyzer-osx.cocoa.IncompatibleMethodTypes check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-osx.cocoa.Loops
Improved modeling of loops using Cocoa collection types

clang-analyzer-osx.cocoa.MissingSuperCall
Warn about Objective-C methods that lack a necessary call to super

clang-analyzer-osx.cocoa.NSAutoreleasePool
The clang-analyzer-osx.cocoa.NSAutoreleasePool check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-osx.cocoa.NSError
The clang-analyzer-osx.cocoa.NSError check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-osx.cocoa.NilArg
The clang-analyzer-osx.cocoa.NilArg check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-osx.cocoa.NonNilReturnValue
Model the APIs that are guaranteed to return a non-nil value

clang-analyzer-osx.cocoa.ObjCGenerics
The clang-analyzer-osx.cocoa.ObjCGenerics check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-osx.cocoa.RetainCount
The clang-analyzer-osx.cocoa.RetainCount check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-osx.cocoa.RunLoopAutoreleaseLeak
Check for leaked memory in autorelease pools that will never be drained

clang-analyzer-osx.cocoa.SelfInit
The clang-analyzer-osx.cocoa.SelfInit check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-osx.cocoa.SuperDealloc
The clang-analyzer-osx.cocoa.SuperDealloc check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-osx.cocoa.UnusedIvars
The clang-analyzer-osx.cocoa.UnusedIvars check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-osx.cocoa.VariadicMethodTypes
The clang-analyzer-osx.cocoa.VariadicMethodTypes check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-osx.coreFoundation.CFError
The clang-analyzer-osx.coreFoundation.CFError check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-osx.coreFoundation.CFNumber
The clang-analyzer-osx.coreFoundation.CFNumber check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-osx.coreFoundation.CFRetainRelease
The clang-analyzer-osx.coreFoundation.CFRetainRelease check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-osx.coreFoundation.containers.OutOfBounds
The clang-analyzer-osx.coreFoundation.containers.OutOfBounds check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-osx.coreFoundation.containers.PointerSizedValues
The clang-analyzer-osx.coreFoundation.containers.PointerSizedValues check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-security.FloatLoopCounter

The clang-analyzer-security.FloatLoopCounter check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-security.insecureAPI.DeprecatedOrUnsafeBufferHandling
The clang-analyzer-security.insecureAPI.DeprecatedOrUnsafeBufferHandling check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-security.insecureAPI.UncheckedReturn
The clang-analyzer-security.insecureAPI.UncheckedReturn check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-security.insecureAPI.bcmp
The clang-analyzer-security.insecureAPI.bcmp check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-security.insecureAPI.bcopy
The clang-analyzer-security.insecureAPI.bcopy check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-security.insecureAPI.bzero
The clang-analyzer-security.insecureAPI.bzero check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-security.insecureAPI.getpw
The clang-analyzer-security.insecureAPI.getpw check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-security.insecureAPI.gets
The clang-analyzer-security.insecureAPI.gets check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-security.insecureAPI.mkstemp
The clang-analyzer-security.insecureAPI.mkstemp check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-security.insecureAPI.mktemp
The clang-analyzer-security.insecureAPI.mktemp check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-security.insecureAPI.rand

The clang-analyzer-security.insecureAPI.rand check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-security.insecureAPI.strcpy
The clang-analyzer-security.insecureAPI.strcpy check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-security.insecureAPI.vfork
The clang-analyzer-security.insecureAPI.vfork check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-unix.API
The clang-analyzer-unix.API check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-unix.Malloc
The clang-analyzer-unix.Malloc check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-unix.MallocSizeof
The clang-analyzer-unix.MallocSizeof check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-unix.MismatchedDeallocator
The clang-analyzer-unix.MismatchedDeallocator check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-unix.Vfork
The clang-analyzer-unix.Vfork check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-unix.cstring.BadSizeArg
The clang-analyzer-unix.cstring.BadSizeArg check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-unix.cstring.NullArg
The clang-analyzer-unix.cstring.NullArg check is an alias, please see *Clang Static Analyzer Available Checkers* for more information.

clang-analyzer-valist.CopyToSelf

Check for va_lists which are copied onto itself.

clang-analyzer-valist.Uninitialized
Check for usages of uninitialized (or already released) va_lists.

clang-analyzer-valist.Unterminated
Check for va_lists which are not released by a va_end call.

concurrency-mt-unsafe
Checks for some thread-unsafe functions against a black list of known-to-be-unsafe functions. Usually they access static variables without synchronization (e.g. gmtime(3)) or utilize signals in a racy way. The set of functions to check is specified with the *FunctionSet* option.

Note that using some thread-unsafe functions may be still valid in concurrent programming if only a single thread is used (e.g. setenv(3)), however, some functions may track a state in global variables which would be clobbered by subsequent (non-parallel, but concurrent) calls to a related function. E.g. the following code suffers from unprotected accesses to a global state:

```
// getnetent(3) maintains global state with DB connection, etc.
// If a concurrent green thread calls getnetent(3), the global state is corrupted.
netent = getnetent();
yield();
netent = getnetent();
```

Examples:

```
tm = gmtime(timep); // uses a global buffer

sleep(1); // implementation may use SIGALRM
```

FunctionSet
Specifies which functions in libc should be considered thread-safe, possible values are *posix*, *glibc*, or *any*.

posix means POSIX defined thread-unsafe functions. POSIX.1-2001 in "2.9.1 Thread-Safety" defines that all functions specified in the standard are thread-safe except a predefined list of thread-unsafe functions.

Glibc defines some of them as thread-safe (e.g. dirname(3)), but adds non-POSIX thread-unsafe ones (e.g. getopt_long(3)). Glibc’s list is compiled from GNU web documentation with a search

for MT-Safe tag:
https://www.gnu.org/software/libc/manual/html_node/POSIX-Safety-Concepts.html

If you want to identify thread-unsafe API for at least one libc or unsure which libc will be used, use *any* (default).

concurrency-thread-canceltype-asynchronous

Finds **pthread_setcanceltype** function calls where a thread's cancellation type is set to asynchronous. Asynchronous cancellation type (**PTHREAD_CANCEL_ASYNCHRONOUS**) is generally unsafe, use type **PTHREAD_CANCEL_DEFERRED** instead which is the default. Even with deferred cancellation, a cancellation point in an asynchronous signal handler may still be acted upon and the effect is as if it was an asynchronous cancellation.

This check corresponds to the CERT C Coding Standard rule *POS47-C. Do not use threads that can be canceled asynchronously*.

cppcoreguidelines-avoid-c-arrays

The cppcoreguidelines-avoid-c-arrays check is an alias, please see *modernize-avoid-c-arrays* for more information.

cppcoreguidelines-avoid-goto

The usage of **goto** for control flow is error prone and should be replaced with looping constructs. Only forward jumps in nested loops are accepted.

This check implements *ES.76* from the CppCoreGuidelines and *6.3.1 from High Integrity C++*.

For more information on why to avoid programming with **goto** you can read the famous paper *A Case against the GO TO Statement.*.

The check diagnoses **goto** for backward jumps in every language mode. These should be replaced with *C/C++* looping constructs.

```
// Bad, handwritten for loop.
int i = 0;
// Jump label for the loop
loop_start:
do_some_operation();

if (i < 100) {
  ++i;
```

```
  goto loop_start;
}

// Better
for(int i = 0; i < 100; ++i)
  do_some_operation();
```

Modern C++ needs **goto** only to jump out of nested loops.

```
for(int i = 0; i < 100; ++i) {
  for(int j = 0; j < 100; ++j) {
    if (i * j > 500)
      goto early_exit;
  }
}

early_exit:
some_operation();
```

All other uses of **goto** are diagnosed in *C++*.

cppcoreguidelines-avoid-magic-numbers

The cppcoreguidelines-avoid-magic-numbers check is an alias, please see *readability-magic-numbers* for more information.

cppcoreguidelines-avoid-non-const-global-variables

Finds non-const global variables as described in *I.2 of C++ Core Guidelines* . As *R.6 of C++ Core Guidelines* is a duplicate of rule I.2 it also covers that rule.

```
char a;  // Warns!
const char b =  0;

namespace some_namespace
{
   char c;  // Warns!
   const char d = 0;
}

char * c_ptr1 = &some_namespace::c;  // Warns!
char *const c_const_ptr = &some_namespace::c;  // Warns!
```

```
char & c_reference = some_namespace::c;  // Warns!

class Foo  // No Warnings inside Foo, only namespace scope is covered
{
public:
   char e = 0;
   const char f = 0;
protected:
   char g = 0;
private:
   char h = 0;
};
```

Variables: **a**, **c**, **c_ptr1**, **c_ptr2**, **c_const_ptr** and **c_reference**, will all generate warnings since they are either: a globally accessible variable and non-const, a pointer or reference providing global access to non-const data or both.

cppcoreguidelines-c-copy-assignment-signature

The cppcoreguidelines-c-copy-assignment-signature check is an alias, please see *misc-unconventional-assign-operator* for more information.

cppcoreguidelines-explicit-virtual-functions

The cppcoreguidelines-explicit-virtual-functions check is an alias, please see *modernize-use-override* for more information.

cppcoreguidelines-init-variables

Checks whether there are local variables that are declared without an initial value. These may lead to unexpected behavior if there is a code path that reads the variable before assigning to it.

Only integers, booleans, floats, doubles and pointers are checked. The fix option initializes all detected values with the value of zero. An exception is float and double types, which are initialized to NaN.

As an example a function that looks like this:

```
void function() {
  int x;
  char *txt;
  double d;

  // Rest of the function.
```

```
}
```

Would be rewritten to look like this:

```
#include <math.h>

void function() {
  int x = 0;
  char *txt = nullptr;
  double d = NAN;

  // Rest of the function.
}
```

It warns for the uninitialized enum case, but without a FixIt:

```
enum A {A1, A2, A3};
enum A_c : char { A_c1, A_c2, A_c3 };
enum class B { B1, B2, B3 };
enum class B_i : int { B_i1, B_i2, B_i3 };
void function() {
  A a;     // Warning: variable 'a' is not initialized
  A_c a_c; // Warning: variable 'a_c' is not initialized
  B b;     // Warning: variable 'b' is not initialized
  B_i b_i; // Warning: variable 'b_i' is not initialized
}
```

Options

IncludeStyle
A string specifying which include-style is used, *llvm* or *google*. Default is *llvm*.

MathHeader
A string specifying the header to include to get the definition of *NAN*. Default is *<math.h>*.

cppcoreguidelines-interfaces-global-init
This check flags initializers of globals that access extern objects, and therefore can lead to order-of-initialization problems.

This rule is part of the "Interfaces" profile of the C++ Core Guidelines, see

https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Ri-global-init

Note that currently this does not flag calls to non-constexpr functions, and therefore globals could still be accessed from functions themselves.

cppcoreguidelines-macro-to-enum

The cppcoreguidelines-macro-to-enum check is an alias, please see *modernize-macro-to-enum* for more information.

cppcoreguidelines-macro-usage

Finds macro usage that is considered problematic because better language constructs exist for the task.

The relevant sections in the C++ Core Guidelines are *ES.31*, and *ES.32*.

Examples:

```
#define C 0
#define F1(x, y) ((a) > (b) ? (a) : (b))
#define F2(...) (__VA_ARGS__)
#define COMMA ,
#define NORETURN [[noreturn]]
#define DEPRECATED attribute((deprecated))
#if LIB_EXPORTS
#define DLLEXPORTS __declspec(dllexport)
#else
#define DLLEXPORTS __declspec(dllimport)
#endif
```

results in the following warnings:

```
4 warnings generated.
test.cpp:1:9: warning: macro 'C' used to declare a constant; consider using a 'constexpr' constant [cppcoreguidelin
#define C 0
        ^
test.cpp:2:9: warning: function-like macro 'F1' used; consider a 'constexpr' template function [cppcoreguidelines-
#define F1(x, y) ((a) > (b) ? (a) : (b))
        ^
test.cpp:3:9: warning: variadic macro 'F2' used; consider using a 'constexpr' variadic template function [cppcoregu
#define F2(...) (__VA_ARGS__)
        ^
```

Options

AllowedRegexp
A regular expression to filter allowed macros. For example *DEBUG*|LIBTORRENT*|TORRENT*|UNI** could be applied to filter *libtorrent*. Default value is *^DEBUG_**.

CheckCapsOnly
Boolean flag to warn on all macros except those with CAPS_ONLY names. This option is intended to ease introduction of this check into older code bases. Default value is *false*.

IgnoreCommandLineMacros
Boolean flag to toggle ignoring command-line-defined macros. Default value is *true*.

cppcoreguidelines-narrowing-conversions
Checks for silent narrowing conversions, e.g: **int i = 0; i += 0.1;**. While the issue is obvious in this former example, it might not be so in the following: **void MyClass::f(double d) { int_member_ += d; }**.

This rule is part of the "Expressions and statements" profile of the C++ Core Guidelines, corresponding to rule ES.46. See

https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#es46-avoid-lossy-narrowing-trunc

We enforce only part of the guideline, more specifically, we flag narrowing conversions from:

- an integer to a narrower integer (e.g. **char** to **unsigned char**) if WarnOnIntegerNarrowingConversion Option is set,
- an integer to a narrower floating-point (e.g. **uint64_t** to **float**) if WarnOnIntegerToFloatingPointNarrowingConversion Option is set,
- a floating-point to an integer (e.g. **double** to **int**),
- a floating-point to a narrower floating-point (e.g. **double** to **float**) if WarnOnFloatingPointNarrowingConversion Option is set.

This check will flag:

- All narrowing conversions that are not marked by an explicit cast (c-style or **static_cast**). For example: **int i = 0; i += 0.1;**, **void f(int); f(0.1);**,

- All applications of binary operators with a narrowing conversions. For example: **int i; i+= 0.1;**.

Options

WarnOnIntegerNarrowingConversion
When *true*, the check will warn on narrowing integer conversion (e.g. **int** to **size_t**). *true* by default.

WarnOnIntegerToFloatingPointNarrowingConversion
When *true*, the check will warn on narrowing integer to floating-point conversion (e.g. **size_t** to **double**). *true* by default.

WarnOnFloatingPointNarrowingConversion
When *true*, the check will warn on narrowing floating point conversion (e.g. **double** to **float**). *true* by default.

WarnWithinTemplateInstantiation
When *true*, the check will warn on narrowing conversions within template instantiations. *false* by default.

WarnOnEquivalentBitWidth
When *true*, the check will warn on narrowing conversions that arise from casting between types of equivalent bit width. (e.g. *int n = uint(0);* or *long long n = double(0);*) *true* by default.

IgnoreConversionFromTypes
Narrowing conversions from any type in this semicolon-separated list will be ignored. This may be useful to weed out commonly occurring, but less commonly problematic assignments such as *int n = std::vector<char>().size();* or *int n = std::difference(it1, it2);*. The default list is empty, but one suggested list for a legacy codebase would be *size_t;ptrdiff_t;size_type;difference_type*.

PedanticMode
When *true*, the check will warn on assigning a floating point constant to an integer value even if the floating point value is exactly representable in the destination type (e.g. **int i = 1.0;**). *false* by default.

FAQ

- What does "narrowing conversion from ’int’ to ’float’" mean?

  An IEEE754 Floating Point number can represent all integer values in the range

[-2^PrecisionBits, 2^PrecisionBits] where PrecisionBits is the number of bits in the mantissa.

For **float** this would be [-2^23, 2^23], where **int** can represent values in the range [-2^31, 2^31-1].

- What does "implementation-defined" mean?

  You may have encountered messages like "narrowing conversion from ’unsigned int’ to signed type ’int’ is implementation-defined". The C/C++ standard does not mandate two’s complement for signed integers, and so the compiler is free to define what the semantics are for converting an unsigned integer to signed integer. Clang’s implementation uses the two’s complement format.

cppcoreguidelines-no-malloc

This check handles C-Style memory management using **malloc()**, **realloc()**, **calloc()** and **free()**. It warns about its use and tries to suggest the use of an appropriate RAII object. Furthermore, it can be configured to check against a user-specified list of functions that are used for memory management (e.g. **posix_memalign()**). See *C++ Core Guidelines*.

There is no attempt made to provide fix-it hints, since manual resource management isn’t easily transformed automatically into RAII.

```
// Warns each of the following lines.
// Containers like std::vector or std::string should be used.
char* some_string = (char*) malloc(sizeof(char) * 20);
char* some_string = (char*) realloc(sizeof(char) * 30);
free(some_string);

int* int_array = (int*) calloc(30, sizeof(int));

// Rather use a smartpointer or stack variable.
struct some_struct* s = (struct some_struct*) malloc(sizeof(struct some_struct));
```

Options

Allocations

Semicolon-separated list of fully qualified names of memory allocation functions. Defaults to **::malloc;::calloc**.

Deallocations

Semicolon-separated list of fully qualified names of memory allocation functions. Defaults to **::free**.

Reallocations
Semicolon-separated list of fully qualified names of memory allocation functions. Defaults to **::realloc**.

cppcoreguidelines-non-private-member-variables-in-classes
The cppcoreguidelines-non-private-member-variables-in-classes check is an alias, please see *misc-non-private-member-variables-in-classes* for more information.

cppcoreguidelines-owning-memory
This check implements the type-based semantics of **gsl::owner<T*>**, which allows static analysis on code, that uses raw pointers to handle resources like dynamic memory, but won't introduce RAII concepts.

The relevant sections in the *C++ Core Guidelines* are I.11, C.33, R.3 and GSL.Views The definition of a **gsl::owner<T*>** is straight forward

```
namespace gsl { template <typename T> owner = T; }
```

It is therefore simple to introduce the owner even without using an implementation of the *Guideline Support Library*.

All checks are purely type based and not (yet) flow sensitive.

The following examples will demonstrate the correct and incorrect initializations of owners, assignment is handled the same way. Note that both **new** and **malloc()**-like resource functions are considered to produce resources.

```
// Creating an owner with factory functions is checked.
gsl::owner<int*> function_that_returns_owner() { return gsl::owner<int*>(new int(42)); }

// Dynamic memory must be assigned to an owner
int* Something = new int(42); // BAD, will be caught
gsl::owner<int*> Owner = new int(42); // Good
gsl::owner<int*> Owner = new int[42]; // Good as well

// Returned owner must be assigned to an owner
int* Something = function_that_returns_owner(); // Bad, factory function
```

```
gsl::owner<int*> Owner = function_that_returns_owner(); // Good, result lands in owner

// Something not a resource or owner should not be assigned to owners
int Stack = 42;
gsl::owner<int*> Owned = &Stack; // Bad, not a resource assigned
```

In the case of dynamic memory as resource, only **gsl::owner<T*>** variables are allowed to be deleted.

```
// Example Bad, non-owner as resource handle, will be caught.
int* NonOwner = new int(42); // First warning here, since new must land in an owner
delete NonOwner; // Second warning here, since only owners are allowed to be deleted

// Example Good, Ownership correctly stated
gsl::owner<int*> Owner = new int(42); // Good
delete Owner; // Good as well, statically enforced, that only owners get deleted
```

The check will furthermore ensure, that functions, that expect a **gsl::owner<T*>** as argument get called with either a **gsl::owner<T*>** or a newly created resource.

```
void expects_owner(gsl::owner<int*> o) { delete o; }

// Bad Code
int NonOwner = 42;
expects_owner(&NonOwner); // Bad, will get caught

// Good Code
gsl::owner<int*> Owner = new int(42);
expects_owner(Owner); // Good
expects_owner(new int(42)); // Good as well, recognized created resource

// Port legacy code for better resource-safety
gsl::owner<FILE*> File = fopen("my_file.txt", "rw+");
FILE* BadFile = fopen("another_file.txt", "w"); // Bad, warned

// ... use the file

fclose(File); // Ok, File is annotated as 'owner<>'
fclose(BadFile); // BadFile is not an 'owner<>', will be warned
```

Options

LegacyResourceProducers
Semicolon-separated list of fully qualified names of legacy functions that create resources but cannot introduce **gsl::owner<>**. Defaults to **::malloc;::aligned_alloc;::realloc;::calloc;::fopen;::freopen;::tmpfile**.

LegacyResourceConsumers
Semicolon-separated list of fully qualified names of legacy functions expecting resource owners as pointer arguments but cannot introduce **gsl::owner<>**. Defaults to **::free;::realloc;::freopen;::fclose**.

Limitations

Using **gsl::owner<T*>** in a typedef or alias is not handled correctly.

```
using heap_int = gsl::owner<int*>;
heap_int allocated = new int(42); // False positive!
```

The **gsl::owner<T*>** is declared as a templated type alias. In template functions and classes, like in the example below, the information of the type aliases gets lost. Therefore using **gsl::owner<T*>** in a heavy templated code base might lead to false positives.

Known code constructs that do not get diagnosed correctly are:

- **std::exchange**

- **std::vector<gsl::owner<T*>>**

```
// This template function works as expected. Type information doesn't get lost.
template <typename T>
void delete_owner(gsl::owner<T*> owned_object) {
  delete owned_object; // Everything alright
}

gsl::owner<int*> function_that_returns_owner() { return gsl::owner<int*>(new int(42)); }

// Type deduction does not work for auto variables.
// This is caught by the check and will be noted accordingly.
auto OwnedObject = function_that_returns_owner(); // Type of OwnedObject will be int*

// Problematic function template that looses the typeinformation on owner
```

```
template <typename T>
void bad_template_function(T some_object) {
  // This line will trigger the warning, that a non-owner is assigned to an owner
  gsl::owner<T*> new_owner = some_object;
}

// Calling the function with an owner still yields a false positive.
bad_template_function(gsl::owner<int*>(new int(42)));


// The same issue occurs with templated classes like the following.
template <typename T>
class OwnedValue {
public:
  const T getValue() const { return _val; }
private:
  T _val;
};

// Code, that yields a false positive.
OwnedValue<gsl::owner<int*>> Owner(new int(42)); // Type deduction yield T -> int *
// False positive, getValue returns int* and not gsl::owner<int*>
gsl::owner<int*> OwnedInt = Owner.getValue();
```

Another limitation of the current implementation is only the type based checking. Suppose you have code like the following:

```
// Two owners with assigned resources
gsl::owner<int*> Owner1 = new int(42);
gsl::owner<int*> Owner2 = new int(42);

Owner2 = Owner1; // Conceptual Leak of initial resource of Owner2!
Owner1 = nullptr;
```

The semantic of a **gsl::owner<T*>** is mostly like a **std::unique_ptr<T>**, therefore assignment of two **gsl::owner<T*>** is considered a move, which requires that the resource **Owner2** must have been released before the assignment. This kind of condition could be caught in later improvements of this check with flowsensitive analysis. Currently, the *Clang Static Analyzer* catches this bug for dynamic memory, but not for general types of resources.

cppcoreguidelines-prefer-member-initializer

Finds member initializations in the constructor body which can be converted into member initializers of the constructor instead. This not only improves the readability of the code but also positively affects its performance. Class-member assignments inside a control statement or following the first control statement are ignored.

This check implements *C.49* from the CppCoreGuidelines.

If the language version is *C++ 11* or above, the constructor is the default constructor of the class, the field is not a bitfield (only in case of earlier language version than *C++ 20*), furthermore the assigned value is a literal, negated literal or **enum** constant then the preferred place of the initialization is at the class member declaration.

This latter rule is *C.48* from CppCoreGuidelines.

Please note, that this check does not enforce this latter rule for initializations already implemented as member initializers. For that purpose see check *modernize-use-default-member-init*.

Example 1

```
class C {
  int n;
  int m;
public:
  C() {
    n = 1; // Literal in default constructor
    if (dice())
      return;
    m = 1;
  }
};
```

Here **n** can be initialized using a default member initializer, unlike **m**, as **m**'s initialization follows a control statement (**if**):

```
class C {
  int n{1};
  int m;
public:
  C() {
```

```
  if (dice())
    return;
  m = 1;
 }
```

Example 2

```
class C {
 int n;
 int m;
public:
 C(int nn, int mm) {
  n = nn; // Neither default constructor nor literal
  if (dice())
    return;
  m = mm;
 }
};
```

Here **n** can be initialized in the constructor initialization list, unlike **m**, as **m**'s initialization follows a control statement (**if**):

```
C(int nn, int mm) : n(nn) {
 if (dice())
  return;
 m = mm;
}
```

UseAssignment

If this option is set to *true* (default is *false*), the check will initialize members with an assignment. In this case the fix of the first example looks like this:

```
class C {
 int n = 1;
 int m;
public:
 C() {
  if (dice())
    return;
  m = 1;
```

```
  }
};
```

cppcoreguidelines-pro-bounds-array-to-pointer-decay
This check flags all array to pointer decays.

Pointers should not be used as arrays. **span<T>** is a bounds-checked, safe alternative to using pointers to access arrays.

This rule is part of the "Bounds safety" profile of the C++ Core Guidelines, see *https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Pro-bounds-decay*.

cppcoreguidelines-pro-bounds-constant-array-index
This check flags all array subscript expressions on static arrays and **std::arrays** that either do not have a constant integer expression index or are out of bounds (for **std::array**). For out-of-bounds checking of static arrays, see the *-Warray-bounds* Clang diagnostic.

This rule is part of the "Bounds safety" profile of the C++ Core Guidelines, see *https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Pro-bounds-arrayindex*.

Optionally, this check can generate fixes using **gsl::at** for indexing.

Options

GslHeader
The check can generate fixes after this option has been set to the name of the include file that contains **gsl::at()**, e.g. *"gsl/gsl.h"*.

IncludeStyle
A string specifying which include-style is used, *llvm* or *google*. Default is *llvm*.

cppcoreguidelines-pro-bounds-pointer-arithmetic
This check flags all usage of pointer arithmetic, because it could lead to an invalid pointer. Subtraction of two pointers is not flagged by this check.

Pointers should only refer to single objects, and pointer arithmetic is fragile and easy to get wrong. **span<T>** is a bounds-checked, safe type for accessing arrays of data.

This rule is part of the "Bounds safety" profile of the C++ Core Guidelines, see *https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Pro-bounds-arithmetic*.

cppcoreguidelines-pro-type-const-cast

This check flags all uses of **const_cast** in C++ code.

Modifying a variable that was declared const is undefined behavior, even with **const_cast**.

This rule is part of the "Type safety" profile of the C++ Core Guidelines, see *https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Pro-type-constcast*.

cppcoreguidelines-pro-type-cstyle-cast

This check flags all use of C-style casts that perform a **static_cast** downcast, **const_cast**, or **reinterpret_cast**.

Use of these casts can violate type safety and cause the program to access a variable that is actually of type X to be accessed as if it were of an unrelated type Z. Note that a C-style **(T)expression** cast means to perform the first of the following that is possible: a **const_cast**, a **static_cast**, a **static_cast** followed by a **const_cast**, a **reinterpret_cast**, or a **reinterpret_cast** followed by a **const_cast**. This rule bans **(T)expression** only when used to perform an unsafe cast.

This rule is part of the "Type safety" profile of the C++ Core Guidelines, see *https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Pro-type-cstylecast*.

cppcoreguidelines-pro-type-member-init

The check flags user-defined constructor definitions that do not initialize all fields that would be left in an undefined state by default construction, e.g. builtins, pointers and record types without user-provided default constructors containing at least one such type. If these fields aren’t initialized, the constructor will leave some of the memory in an undefined state.

For C++11 it suggests fixes to add in-class field initializers. For older versions it inserts the field initializers into the constructor initializer list. It will also initialize any direct base classes that need to be zeroed in the constructor initializer list.

The check takes assignment of fields in the constructor body into account but generates false positives for fields initialized in methods invoked in the constructor body.

The check also flags variables with automatic storage duration that have record types without a user-provided constructor and are not initialized. The suggested fix is to zero initialize the variable via **{}** for C++11 and beyond or **= {}** for older language versions.

Options

IgnoreArrays
If set to *true*, the check will not warn about array members that are not zero-initialized during construction. For performance critical code, it may be important to not initialize fixed-size array members. Default is *false*.

UseAssignment
If set to *true*, the check will provide fix-its with literal initializers (**int i = 0;**) instead of curly braces (**int i{};**).

This rule is part of the "Type safety" profile of the C++ Core Guidelines, corresponding to rule Type.6. See *https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Pro-type-memberinit*.

cppcoreguidelines-pro-type-reinterpret-cast
This check flags all uses of **reinterpret_cast** in C++ code.

Use of these casts can violate type safety and cause the program to access a variable that is actually of type **X** to be accessed as if it were of an unrelated type **Z**.

This rule is part of the "Type safety" profile of the C++ Core Guidelines, see *https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Pro-type-reinterpretcast*.

cppcoreguidelines-pro-type-static-cast-downcast
This check flags all usages of **static_cast**, where a base class is casted to a derived class. In those cases, a fix-it is provided to convert the cast to a **dynamic_cast**.

Use of these casts can violate type safety and cause the program to access a variable that is actually of type **X** to be accessed as if it were of an unrelated type **Z**.

This rule is part of the "Type safety" profile of the C++ Core Guidelines, see *https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Pro-type-downcast*.

cppcoreguidelines-pro-type-union-access
This check flags all access to members of unions. Passing unions as a whole is not flagged.

Reading from a union member assumes that member was the last one written, and writing to a union member assumes another member with a nontrivial destructor had its destructor called. This is fragile because it cannot generally be enforced to be safe in the language and so relies on programmer discipline to get it right.

This rule is part of the "Type safety" profile of the C++ Core Guidelines, see *https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Pro-type-unions*.

cppcoreguidelines-pro-type-vararg

This check flags all calls to c-style vararg functions and all use of **va_arg**.

To allow for SFINAE use of vararg functions, a call is not flagged if a literal 0 is passed as the only vararg argument.

Passing to varargs assumes the correct type will be read. This is fragile because it cannot generally be enforced to be safe in the language and so relies on programmer discipline to get it right.

This rule is part of the "Type safety" profile of the C++ Core Guidelines, see *https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Pro-type-varargs*.

cppcoreguidelines-slicing

Flags slicing of member variables or vtable. Slicing happens when copying a derived object into a base object: the members of the derived object (both member variables and virtual member functions) will be discarded. This can be misleading especially for member function slicing, for example:

```
struct B { int a; virtual int f(); };
struct D : B { int b; int f() override; };

void use(B b) {  // Missing reference, intended?
  b.f();  // Calls B::f.
}

D d;
use(d);  // Slice.
```

See the relevant C++ Core Guidelines sections for details:
https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#es63-dont-slice
https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#c145-access-polymorphic-o

cppcoreguidelines-special-member-functions

The check finds classes where some but not all of the special member functions are defined.

By default the compiler defines a copy constructor, copy assignment operator, move constructor, move assignment operator and destructor. The default can be suppressed by explicit user-definitions. The relationship between which functions will be suppressed by definitions of other functions is

complicated and it is advised that all five are defaulted or explicitly defined.

Note that defining a function with **= delete** is considered to be a definition.

This rule is part of the "Constructors, assignments, and destructors" profile of the C++ Core Guidelines, corresponding to rule C.21. See

https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#c21-if-you-define-or-delete-any-de

Options

AllowSoleDefaultDtor
When set to *true* (default is *false*), this check doesn't flag classes with a sole, explicitly defaulted destructor. An example for such a class is:

```
struct A {
  virtual ~A() = default;
};
```

AllowMissingMoveFunctions
When set to *true* (default is *false*), this check doesn't flag classes which define no move operations at all. It still flags classes which define only one of either move constructor or move assignment operator. With this option enabled, the following class won't be flagged:

```
struct A {
  A(const A&);
  A& operator=(const A&);
  ~A();
};
```

AllowMissingMoveFunctionsWhenCopyIsDeleted
When set to *true* (default is *false*), this check doesn't flag classes which define deleted copy operations but don't define move operations. This flag is related to Google C++ Style Guide *https://google.github.io/styleguide/cppguide.html#Copyable_Movable_Types*. With this option enabled, the following class won't be flagged:

```
struct A {
  A(const A&) = delete;
  A& operator=(const A&) = delete;
  ~A();
```

```
};
```

cppcoreguidelines-virtual-class-destructor

Finds virtual classes whose destructor is neither public and virtual nor protected and non-virtual. A virtual class's destructor should be specified in one of these ways to prevent undefined behavior.

This check implements *C.35* from the CppCoreGuidelines.

Note that this check will diagnose a class with a virtual method regardless of whether the class is used as a base class or not.

Fixes are available for user-declared and implicit destructors that are either public and non-virtual or protected and virtual. No fixes are offered for private destructors. There, the decision whether to make them private and virtual or protected and non-virtual depends on the use case and is thus left to the user.

Example

For example, the following classes/structs get flagged by the check since they violate guideline **C.35**:

```
struct Foo {        // NOK, protected destructor should not be virtual
  virtual void f();
protected:
  virtual ~Foo(){}
};

class Bar {         // NOK, public destructor should be virtual
  virtual void f();
public:
  ~Bar(){}
};
```

This would be rewritten to look like this:

```
struct Foo {        // OK, destructor is not virtual anymore
  virtual void f();
protected:
  ~Foo(){}
};

class Bar {         // OK, destructor is now virtual
```

```
  virtual void f();
public:
  virtual ~Bar(){}
};
```

darwin-avoid-spinlock
Finds usages of **OSSpinlock**, which is deprecated due to potential livelock problems.

This check will detect following function invocations:

- **OSSpinlockLock**
- **OSSpinlockTry**
- **OSSpinlockUnlock**

The corresponding information about the problem of **OSSpinlock**:
https://blog.postmates.com/why-spinlocks-are-bad-on-ios-b69fc5221058

darwin-dispatch-once-nonstatic
Finds declarations of **dispatch_once_t** variables without static or global storage. The behavior of using **dispatch_once_t** predicates with automatic or dynamic storage is undefined by libdispatch, and should be avoided.

It is a common pattern to have functions initialize internal static or global data once when the function runs, but programmers have been known to miss the static on the **dispatch_once_t** predicate, leading to an uninitialized flag value at the mercy of the stack.

Programmers have also been known to make **dispatch_once_t** variables be members of structs or classes, with the intent to lazily perform some expensive struct or class member initialization only once; however, this violates the libdispatch requirements.

See the discussion section of *Apple's dispatch_once documentation* for more information.

fuchsia-default-arguments-calls
Warns if a function or method is called with default arguments.

For example, given the declaration:

```
int foo(int value = 5) { return value; }
```

A function call expression that uses a default argument will be diagnosed. Calling it without defaults will not cause a warning:

```
foo();  // warning
foo(0); // no warning
```

See the features disallowed in Fuchsia at *https://fuchsia.googlesource.com/zircon/+/master/docs/cxx.md*

fuchsia-default-arguments-declarations

Warns if a function or method is declared with default parameters.

For example, the declaration:

```
int foo(int value = 5) { return value; }
```

will cause a warning.

See the features disallowed in Fuchsia at *https://fuchsia.googlesource.com/zircon/+/master/docs/cxx.md*

fuchsia-header-anon-namespaces

The fuchsia-header-anon-namespaces check is an alias, please see *google-build-namespace* for more information.

fuchsia-multiple-inheritance

Warns if a class inherits from multiple classes that are not pure virtual.

For example, declaring a class that inherits from multiple concrete classes is disallowed:

```
class Base_A {
public:
  virtual int foo() { return 0; }
};

class Base_B {
public:
  virtual int bar() { return 0; }
};
```

```
// Warning
class Bad_Child1 : public Base_A, Base_B {};
```

A class that inherits from a pure virtual is allowed:

```
class Interface_A {
public:
  virtual int foo() = 0;
};

class Interface_B {
public:
  virtual int bar() = 0;
};

// No warning
class Good_Child1 : public Interface_A, Interface_B {
  virtual int foo() override { return 0; }
  virtual int bar() override { return 0; }
};
```

See the features disallowed in Fuchsia at *https://fuchsia.googlesource.com/zircon/+/master/docs/cxx.md*

fuchsia-overloaded-operator

Warns if an operator is overloaded, except for the assignment (copy and move) operators.

For example:

```
int operator+(int);     // Warning

B &operator=(const B &Other);  // No warning
B &operator=(B &&Other) // No warning
```

See the features disallowed in Fuchsia at *https://fuchsia.googlesource.com/zircon/+/master/docs/cxx.md*

fuchsia-statically-constructed-objects

Warns if global, non-trivial objects with static storage are constructed, unless the object is statically initialized with a **constexpr** constructor or has no explicit constructor.

For example:

```
class A {};

class B {
public:
  B(int Val) : Val(Val) {}
private:
  int Val;
};

class C {
public:
  C(int Val) : Val(Val) {}
  constexpr C() : Val(0) {}

private:
  int Val;
};

static A a;         // No warning, as there is no explicit constructor
static C c(0);      // No warning, as constructor is constexpr

static B b(0);      // Warning, as constructor is not constexpr
static C c2(0, 1);  // Warning, as constructor is not constexpr

static int i;       // No warning, as it is trivial

extern int get_i();
static C(get_i())   // Warning, as the constructor is dynamically initialized
```

See the features disallowed in Fuchsia at *https://fuchsia.googlesource.com/zircon/+/master/docs/cxx.md*

fuchsia-trailing-return

Functions that have trailing returns are disallowed, except for those using **decltype** specifiers and lambda with otherwise unutterable return types.

For example:

```
// No warning
int add_one(const int arg) { return arg; }

// Warning
auto get_add_one() -> int (*)(const int) {
  return add_one;
}
```

Exceptions are made for lambdas and **decltype** specifiers:

```
// No warning
auto lambda = [](double x, double y) -> double {return x + y;};

// No warning
template <typename T1, typename T2>
auto fn(const T1 &lhs, const T2 &rhs) -> decltype(lhs + rhs) {
  return lhs + rhs;
}
```

See the features disallowed in Fuchsia at *https://fuchsia.googlesource.com/zircon/+/master/docs/cxx.md*

fuchsia-virtual-inheritance

Warns if classes are defined with virtual inheritance.

For example, classes should not be defined with virtual inheritance:

```
class B : public virtual A {};   // warning
```

See the features disallowed in Fuchsia at *https://fuchsia.googlesource.com/zircon/+/master/docs/cxx.md*

google-build-explicit-make-pair

Check that **make_pair**'s template arguments are deduced.

G++ 4.6 in C++11 mode fails badly if **make_pair**'s template arguments are specified explicitly, and such use isn't intended in any case.

Corresponding cpplint.py check name: *build/explicit_make_pair*.

google-build-namespaces

cert-dcl59-cpp redirects here as an alias for this check. *fuchsia-header-anon-namespaces* redirects here as an alias for this check.

Finds anonymous namespaces in headers.

https://google.github.io/styleguide/cppguide.html#Namespaces

Corresponding cpplint.py check name: *build/namespaces*.

Options

HeaderFileExtensions

A comma-separated list of filename extensions of header files (the filename extensions should not include "." prefix). Default is "h,hh,hpp,hxx". For header files without an extension, use an empty string (if there are no other desired extensions) or leave an empty element in the list. E.g., "h,hh,hpp,hxx," (note the trailing comma).

google-build-using-namespace

Finds **using namespace** directives.

The check implements the following rule of the *Google C++ Style Guide*:

You may not use a using-directive to make all names from a namespace available.

```
// Forbidden -- This pollutes the namespace.
using namespace foo;
```

Corresponding cpplint.py check name: *build/namespaces*.

google-default-arguments

Checks that default arguments are not given for virtual methods.

See *https://google.github.io/styleguide/cppguide.html#Default_Arguments*

google-explicit-constructor

Checks that constructors callable with a single argument and conversion operators are marked explicit to avoid the risk of unintentional implicit conversions.

Consider this example:

```
struct S {
  int x;
  operator bool() const { return true; }
};

bool f() {
  S a{1};
  S b{2};
  return a == b;
}
```

The function will return **true**, since the objects are implicitly converted to **bool** before comparison, which is unlikely to be the intent.

The check will suggest inserting **explicit** before the constructor or conversion operator declaration. However, copy and move constructors should not be explicit, as well as constructors taking a single **initializer_list** argument.

This code:

```
struct S {
  S(int a);
  explicit S(const S&);
  operator bool() const;
  ...
```

will become

```
struct S {
  explicit S(int a);
  S(const S&);
  explicit operator bool() const;
  ...
```

See *https://google.github.io/styleguide/cppguide.html#Explicit_Constructors*

google-global-names-in-headers

Flag global namespace pollution in header files. Right now it only triggers on **using** declarations and directives.

The relevant style guide section is *https://google.github.io/styleguide/cppguide.html#Namespaces*.

Options

HeaderFileExtensions
A comma-separated list of filename extensions of header files (the filename extensions should not contain "." prefix). Default is "h". For header files without an extension, use an empty string (if there are no other desired extensions) or leave an empty element in the list. E.g., "h,hh,hpp,hxx," (note the trailing comma).

google-objc-avoid-nsobject-new
Finds calls to **+new** or overrides of it, which are prohibited by the Google Objective-C style guide.

The Google Objective-C style guide forbids calling **+new** or overriding it in class implementations, preferring **+alloc** and **-init** methods to instantiate objects.

An example:

```
NSDate *now = [NSDate new];
Foo *bar = [Foo new];
```

Instead, code should use **+alloc**/**-init** or class factory methods.

```
NSDate *now = [NSDate date];
Foo *bar = [[Foo alloc] init];
```

This check corresponds to the Google Objective-C Style Guide rule *Do Not Use +new*.

google-objc-avoid-throwing-exception
Finds uses of throwing exceptions usages in Objective-C files.

For the same reason as the Google C++ style guide, we prefer not throwing exceptions from Objective-C code.

The corresponding C++ style guide rule: *https://google.github.io/styleguide/cppguide.html#Exceptions*

Instead, prefer passing in **NSError **** and return **BOOL** to indicate success or failure.

A counterexample:

```
- (void)readFile {
  if ([self isError]) {
    @throw [NSException exceptionWithName:...];
  }
}
```

Instead, returning an error via **NSError **** is preferred:

```
- (BOOL)readFileWithError:(NSError **)error {
  if ([self isError]) {
    *error = [NSError errorWithDomain:...];
    return NO;
  }
  return YES;
}
```

The corresponding style guide rule:
https://google.github.io/styleguide/objcguide.html#avoid-throwing-exceptions

google-objc-function-naming

Finds function declarations in Objective-C files that do not follow the pattern described in the Google Objective-C Style Guide.

The corresponding style guide rule can be found here:
https://google.github.io/styleguide/objcguide.html#function-names

All function names should be in Pascal case. Functions whose storage class is not static should have an appropriate prefix.

The following code sample does not follow this pattern:

```
static bool is_positive(int i) { return i > 0; }
bool IsNegative(int i) { return i < 0; }
```

The sample above might be corrected to the following code:

```
static bool IsPositive(int i) { return i > 0; }
bool *ABCIsNegative(int i) { return i < 0; }
```

google-objc-global-variable-declaration

Finds global variable declarations in Objective-C files that do not follow the pattern of variable names in Google's Objective-C Style Guide.

The corresponding style guide rule: *https://google.github.io/styleguide/objcguide.html#variable-names*

All the global variables should follow the pattern of **g[A-Z].*** (variables) or **k[A-Z].*** (constants). The check will suggest a variable name that follows the pattern if it can be inferred from the original name.

For code:

static NSString* myString = @"hello";

The fix will be:

static NSString* gMyString = @"hello";

Another example of constant:

static NSString* const myConstString = @"hello";

The fix will be:

static NSString* const kMyConstString = @"hello";

However for code that prefixed with non-alphabetical characters like:

static NSString* __anotherString = @"world";

The check will give a warning message but will not be able to suggest a fix. The user needs to fix it on their own.

google-readability-avoid-underscore-in-googletest-name

Checks whether there are underscores in googletest test and test case names in test macros:

- **TEST**
- **TEST_F**
- **TEST_P**

- **TYPED_TEST**

- **TYPED_TEST_P**

  The **FRIEND_TEST** macro is not included.

  For example:

```
TEST(TestCaseName, Illegal_TestName) {}
TEST(Illegal_TestCaseName, TestName) {}
```

  would trigger the check. *Underscores are not allowed* in test names nor test case names.

  The **DISABLED_** prefix, which may be used to *disable individual tests*, is ignored when checking test names, but the rest of the rest of the test name is still checked.

  This check does not propose any fixes.

google-readability-braces-around-statements
The google-readability-braces-around-statements check is an alias, please see *readability-braces-around-statements* for more information.

google-readability-casting
Finds usages of C-style casts.

https://google.github.io/styleguide/cppguide.html#Casting

Corresponding cpplint.py check name: *readability/casting*.

This check is similar to **-Wold-style-cast**, but it suggests automated fixes in some cases. The reported locations should not be different from the ones generated by **-Wold-style-cast**.

google-readability-function-size
The google-readability-function-size check is an alias, please see *readability-function-size* for more information.

google-readability-namespace-comments
The google-readability-namespace-comments check is an alias, please see *llvm-namespace-comment* for more information.

google-readability-todo
Finds TODO comments without a username or bug number.

The relevant style guide section is *https://google.github.io/styleguide/cppguide.html#TODO_Comments*.

Corresponding cpplint.py check: *readability/todo*

google-runtime-int
Finds uses of **short**, **long** and **long long** and suggest replacing them with **u?intXX(_t)?**.

The corresponding style guide rule: *https://google.github.io/styleguide/cppguide.html#Integer_Types*.

Corresponding cpplint.py check: *runtime/int*.

Options

UnsignedTypePrefix
A string specifying the unsigned type prefix. Default is *uint*.

SignedTypePrefix
A string specifying the signed type prefix. Default is *int*.

TypeSuffix
A string specifying the type suffix. Default is an empty string.

google-runtime-operator
Finds overloads of unary **operator &**.

https://google.github.io/styleguide/cppguide.html#Operator_Overloading

Corresponding cpplint.py check name: *runtime/operator*.

google-upgrade-googletest-case
Finds uses of deprecated Google Test version 1.9 APIs with names containing **case** and replaces them with equivalent APIs with **suite**.

All names containing **case** are being replaced to be consistent with the meanings of "test case" and "test suite" as used by the International Software Testing Qualifications Board and ISO 29119.

The new names are a part of Google Test version 1.9 (release pending). It is recommended that users update their dependency to version 1.9 and then use this check to remove deprecated names.

The affected APIs are:

- Member functions of **testing::Test**, **testing::TestInfo**, **testing::TestEventListener**, **testing::UnitTest**, and any type inheriting from these types
- The macros **TYPED_TEST_CASE**, **TYPED_TEST_CASE_P**, **REGISTER_TYPED_TEST_CASE_P**, and **INSTANTIATE_TYPED_TEST_CASE_P**
- The type alias **testing::TestCase**

Examples of fixes created by this check:

```
class FooTest : public testing::Test {
public:
  static void SetUpTestCase();
  static void TearDownTestCase();
};

TYPED_TEST_CASE(BarTest, BarTypes);
```

becomes

```
class FooTest : public testing::Test {
public:
  static void SetUpTestSuite();
  static void TearDownTestSuite();
};

TYPED_TEST_SUITE(BarTest, BarTypes);
```

For better consistency of user code, the check renames both virtual and non-virtual member functions with matching names in derived types. The check tries to provide only a warning when a fix cannot be made safely, as is the case with some template and macro uses.

hicpp-avoid-c-arrays

The hicpp-avoid-c-arrays check is an alias, please see *modernize-avoid-c-arrays* for more information.

hicpp-avoid-goto

The *hicpp-avoid-goto* check is an alias to *cppcoreguidelines-avoid-goto*. Rule *6.3.1 High Integrity C++* requires that **goto** only skips parts of a block and is not used for other reasons.

Both coding guidelines implement the same exception to the usage of **goto**.

hicpp-braces-around-statements

The *hicpp-braces-around-statements* check is an alias, please see *readability-braces-around-statements* for more information. It enforces the *rule 6.1.1*.

hicpp-deprecated-headers

The *hicpp-deprecated-headers* check is an alias, please see *modernize-deprecated-headers* for more information. It enforces the *rule 1.3.3*.

hicpp-exception-baseclass

Ensure that every value that in a **throw** expression is an instance of **std::exception**.

This enforces *rule 15.1* of the High Integrity C++ Coding Standard.

```
class custom_exception {};

void throwing() noexcept(false) {
  // Problematic throw expressions.
  throw int(42);
  throw custom_exception();
}

class mathematical_error : public std::exception {};

void throwing2() noexcept(false) {
  // These kind of throws are ok.
  throw mathematical_error();
  throw std::runtime_error();
  throw std::exception();
}
```

hicpp-explicit-conversions

This check is an alias for *google-explicit-constructor*. Used to enforce parts of *rule 5.4.1*. This check will enforce that constructors and conversion operators are marked *explicit*. Other forms of casting checks are implemented in other places. The following checks can be used to check for more forms of

casting:

- *cppcoreguidelines-pro-type-static-cast-downcast*
- *cppcoreguidelines-pro-type-reinterpret-cast*
- *cppcoreguidelines-pro-type-const-cast*
- *cppcoreguidelines-pro-type-cstyle-cast*

hicpp-function-size
This check is an alias for *readability-function-size*. Useful to enforce multiple sections on function complexity.

- *rule 8.2.2*
- *rule 8.3.1*
- *rule 8.3.2*

hicpp-invalid-access-moved
This check is an alias for *bugprone-use-after-move*.

Implements parts of the *rule 8.4.1* to check if moved-from objects are accessed.

hicpp-member-init
This check is an alias for *cppcoreguidelines-pro-type-member-init*. Implements the check for *rule 12.4.2* to initialize class members in the right order.

hicpp-move-const-arg
The *hicpp-move-const-arg* check is an alias, please see *performance-move-const-arg* for more information. It enforces the *rule 17.3.1*.

hicpp-multiway-paths-covered
This check discovers situations where code paths are not fully-covered. It furthermore suggests using **if** instead of **switch** if the code will be more clear. The *rule 6.1.2* and *rule 6.1.4* of the High Integrity C++ Coding Standard are enforced.

if-else if chains that miss a final **else** branch might lead to unexpected program execution and be the result of a logical error. If the missing **else** branch is intended you can leave it empty with a clarifying

comment. This warning can be noisy on some code bases, so it is disabled by default.

```
void f1() {
  int i = determineTheNumber();

   if(i > 0) {
     // Some Calculation
   } else if (i < 0) {
     // Precondition violated or something else.
   }
   // ...
}
```

Similar arguments hold for **switch** statements which do not cover all possible code paths.

```
// The missing default branch might be a logical error. It can be kept empty
// if there is nothing to do, making it explicit.
void f2(int i) {
  switch (i) {
  case 0: // something
    break;
  case 1: // something else
    break;
  }
  // All other numbers?
}

// Violates this rule as well, but already emits a compiler warning (-Wswitch).
enum Color { Red, Green, Blue, Yellow };
void f3(enum Color c) {
  switch (c) {
  case Red: // We can't drive for now.
    break;
  case Green:  // We are allowed to drive.
    break;
  }
  // Other cases missing
}
```

The *rule 6.1.4* requires every **switch** statement to have at least two **case** labels other than a

default label. Otherwise, the **switch** could be better expressed with an **if** statement. Degenerated **switch** statements without any labels are caught as well.

```
// Degenerated switch that could be better written as ‘if‘
int i = 42;
switch(i) {
  case 1: // do something here
  default: // do something else here
}

// Should rather be the following:
if (i == 1) {
  // do something here
}
else {
  // do something here
}

// A completely degenerated switch will be diagnosed.
int i = 42;
switch(i) {}
```

Options

WarnOnMissingElse

Boolean flag that activates a warning for missing **else** branches. Default is *false*.

hicpp-named-parameter

This check is an alias for *readability-named-parameter*.

Implements *rule 8.2.1*.

hicpp-new-delete-operators

This check is an alias for *misc-new-delete-overloads*. Implements *rule 12.3.1* to ensure the *new* and *delete* operators have the correct signature.

hicpp-no-array-decay

The *hicpp-no-array-decay* check is an alias, please see *cppcoreguidelines-pro-bounds-array-to-pointer-decay* for more information. It enforces the *rule 4.1.1*.

hicpp-no-assembler
Check for assembler statements. No fix is offered.

Inline assembler is forbidden by the *High Integrity C++ Coding Standard* as it restricts the portability of code.

hicpp-no-malloc
The *hicpp-no-malloc* check is an alias, please see *cppcoreguidelines-no-malloc* for more information. It enforces the *rule 5.3.2*.

hicpp-noexcept-move
This check is an alias for *performance-noexcept-move-constructor*. Checks *rule 12.5.4* to mark move assignment and move construction *noexcept*.

hicpp-signed-bitwise
Finds uses of bitwise operations on signed integer types, which may lead to undefined or implementation defined behavior.

The according rule is defined in the *High Integrity C++ Standard, Section 5.6.1*.

Options

IgnorePositiveIntegerLiterals
If this option is set to *true*, the check will not warn on bitwise operations with positive integer literals, e.g. *~0*, *2 << 1*, etc. Default value is *false*.

hicpp-special-member-functions
This check is an alias for *cppcoreguidelines-special-member-functions*. Checks that special member functions have the correct signature, according to *rule 12.5.7*.

hicpp-static-assert
The *hicpp-static-assert* check is an alias, please see *misc-static-assert* for more information. It enforces the *rule 7.1.10*.

hicpp-undelegated-constructor
This check is an alias for *bugprone-undelegated-constructor*. Partially implements *rule 12.4.5* to find misplaced constructor calls inside a constructor.

```
struct Ctor {
  Ctor();
```

```
  Ctor(int);
  Ctor(int, int);
  Ctor(Ctor *i) {
    // All Ctor() calls result in a temporary object
    Ctor(); // did you intend to call a delegated constructor?
    Ctor(0); // did you intend to call a delegated constructor?
    Ctor(1, 2); // did you intend to call a delegated constructor?
    foo();
  }
};
```

hicpp-uppercase-literal-suffix

The hicpp-uppercase-literal-suffix check is an alias, please see *readability-uppercase-literal-suffix* for more information.

hicpp-use-auto

The *hicpp-use-auto* check is an alias, please see *modernize-use-auto* for more information. It enforces the *rule 7.1.8*.

hicpp-use-emplace

The *hicpp-use-emplace* check is an alias, please see *modernize-use-emplace* for more information. It enforces the *rule 17.4.2*.

hicpp-use-equals-default

This check is an alias for *modernize-use-equals-default*. Implements *rule 12.5.1* to explicitly default special member functions.

hicpp-use-equals-delete

This check is an alias for *modernize-use-equals-delete*. Implements *rule 12.5.1* to explicitly default or delete special member functions.

hicpp-use-noexcept

The *hicpp-use-noexcept* check is an alias, please see *modernize-use-noexcept* for more information. It enforces the *rule 1.3.5*.

hicpp-use-nullptr

The *hicpp-use-nullptr* check is an alias, please see *modernize-use-nullptr* for more information. It enforces the *rule 2.5.3*.

hicpp-use-override

This check is an alias for *modernize-use-override*. Implements *rule 10.2.1* to declare a virtual function *override* when overriding.

hicpp-vararg

The *hicpp-vararg* check is an alias, please see *cppcoreguidelines-pro-type-vararg* for more information. It enforces the *rule 14.1.1*.

linuxkernel-must-use-errs

Checks Linux kernel code to see if it uses the results from the functions in **linux/err.h**. Also checks to see if code uses the results from functions that directly return a value from one of these error functions.

This is important in the Linux kernel because **ERR_PTR**, **PTR_ERR**, **IS_ERR**, **IS_ERR_OR_NULL**, **ERR_CAST**, and **PTR_ERR_OR_ZERO** return values must be checked, since positive pointers and negative error codes are being used in the same context. These functions are marked with **__attribute__((warn_unused_result))**, but some kernel versions do not have this warning enabled for clang.

Examples:

```
/* Trivial unused call to an ERR function */
PTR_ERR_OR_ZERO(some_function_call());

/* A function that returns ERR_PTR. */
void *fn() { ERR_PTR(-EINVAL); }

/* An invalid use of fn. */
fn();
```

llvm-else-after-return

The llvm-else-after-return check is an alias, please see *readability-else-after-return* for more information.

llvm-header-guard

Finds and fixes header guards that do not adhere to LLVM style.

Options

HeaderFileExtensions

A comma-separated list of filename extensions of header files (the filename extensions should not include "." prefix). Default is "h,hh,hpp,hxx". For header files without an extension, use an empty

string (if there are no other desired extensions) or leave an empty element in the list. E.g., "h,hh,hpp,hxx," (note the trailing comma).

llvm-include-order

Checks the correct order of **#includes**.

See *https://llvm.org/docs/CodingStandards.html#include-style*

llvm-namespace-comment

google-readability-namespace-comments redirects here as an alias for this check.

Checks that long namespaces have a closing comment.

https://llvm.org/docs/CodingStandards.html#namespace-indentation

https://google.github.io/styleguide/cppguide.html#Namespaces

```
namespace n1 {
void f();
}

// becomes

namespace n1 {
void f();
}  // namespace n1
```

Options

ShortNamespaceLines

Requires the closing brace of the namespace definition to be followed by a closing comment if the body of the namespace has more than *ShortNamespaceLines* lines of code. The value is an unsigned integer that defaults to *1U*.

SpacesBeforeComments

An unsigned integer specifying the number of spaces before the comment closing a namespace definition. Default is *1U*.

llvm-prefer-isa-or-dyn-cast-in-conditionals

Looks at conditionals and finds and replaces cases of **cast<>**, which will assert rather than return a null

pointer, and **dyn_cast<>** where the return value is not captured. Additionally, finds and replaces cases that match the pattern **var && isa<X>(var)**, where **var** is evaluated twice.

```
// Finds these:
if (auto x = cast<X>(y)) {}
// is replaced by:
if (auto x = dyn_cast<X>(y)) {}

if (cast<X>(y)) {}
// is replaced by:
if (isa<X>(y)) {}

if (dyn_cast<X>(y)) {}
// is replaced by:
if (isa<X>(y)) {}

if (var && isa<T>(var)) {}
// is replaced by:
if (isa_and_nonnull<T>(var.foo())) {}

// Other cases are ignored, e.g.:
if (auto f = cast<Z>(y)->foo()) {}
if (cast<Z>(y)->foo()) {}
if (X.cast(y)) {}
```

llvm-prefer-register-over-unsigned

Finds historical use of **unsigned** to hold vregs and physregs and rewrites them to use **Register**.

Currently this works by finding all variables of unsigned integer type whose initializer begins with an implicit cast from **Register** to **unsigned**.

```
void example(MachineOperand &MO) {
  unsigned Reg = MO.getReg();
  ...
}
```

becomes:

```
void example(MachineOperand &MO) {
  Register Reg = MO.getReg();
```

```
  ...
}
```

llvm-qualified-auto

The llvm-qualified-auto check is an alias, please see *readability-qualified-auto* for more information.

llvm-twine-local

Looks for local **Twine** variables which are prone to use after frees and should be generally avoided.

```
static Twine Moo = Twine("bark") + "bah";

// becomes

static std::string Moo = (Twine("bark") + "bah").str();
```

llvmlibc-callee-namespace

Checks all calls resolve to functions within **__llvm_libc** namespace.

```
namespace __llvm_libc {

// Allow calls with the fully qualified name.
__llvm_libc::strlen("hello");

// Allow calls to compiler provided functions.
(void)__builtin_abs(-1);

// Bare calls are allowed as long as they resolve to the correct namespace.
strlen("world");

// Disallow calling into functions in the global namespace.
::strlen("!");

} // namespace __llvm_libc
```

llvmlibc-implementation-in-namespace

Checks that all declarations in the llvm-libc implementation are within the correct namespace.

```
// Correct: implementation inside the correct namespace.
namespace __llvm_libc {
  void LLVM_LIBC_ENTRYPOINT(strcpy)(char *dest, const char *src) {}
```

```
  // Namespaces within __llvm_libc namespace are allowed.
  namespace inner{
    int localVar = 0;
  }
  // Functions with C linkage are allowed.
  extern "C" void str_fuzz(){}
}

// Incorrect: implementation not in a namespace.
void LLVM_LIBC_ENTRYPOINT(strcpy)(char *dest, const char *src) {}

// Incorrect: outer most namespace is not correct.
namespace something_else {
  void LLVM_LIBC_ENTRYPOINT(strcpy)(char *dest, const char *src) {}
}
```

llvmlibc-restrict-system-libc-headers

Finds includes of system libc headers not provided by the compiler within llvm-libc implementations.

```
#include <stdio.h>           // Not allowed because it is part of system libc.
#include <stddef.h>          // Allowed because it is provided by the compiler.
#include "internal/stdio.h"  // Allowed because it is NOT part of system libc.
```

This check is necessary because accidentally including system libc headers can lead to subtle and hard to detect bugs. For example consider a system libc whose **dirent** struct has slightly different field ordering than llvm-libc. While this will compile successfully, this can cause issues during runtime because they are ABI incompatible.

Options

Includes

A string containing a comma separated glob list of allowed include filenames. Similar to the -checks glob list for running clang-tidy itself, the two wildcard characters are * and -, to include and exclude globs, respectively. The default is -*, which disallows all includes.

This can be used to allow known safe includes such as Linux development headers. See *portability-restrict-system-includes* for more details.

misc-confusable-identifiers

Warn about confusable identifiers, i.e. identifiers that are visually close to each other, but use different

Unicode characters. This detects a potential attack described in *CVE-2021-42574*.

Example:

```
int fo; // Initial character is U+0066 (LATIN SMALL LETTER F).
int <?>o; // Initial character is U+1234 (SUPER COOL AWESOME UPPERCASE NOT LATIN F) not U+0066 (I
```

misc-const-correctness

This check implements detection of local variables which could be declared as **const** but are not. Declaring variables as **const** is required or recommended by many coding guidelines, such as: *CppCoreGuidelines ES.25* and *AUTOSAR C++14 Rule A7-1-1 (6.7.1 Specifiers)*.

Please note that this check's analysis is type-based only. Variables that are not modified but used to create a non-const handle that might escape the scope are not diagnosed as potential **const**.

```
// Declare a variable, which is not ‘‘const‘‘ ...
int i = 42;
// but use it as read-only. This means that ‘i‘ can be declared ‘‘const‘‘.
int result = i * i;       // Before transformation
int const result = i * i; // After transformation
```

The check can analyze values, pointers and references but not (yet) pointees:

```
// Normal values like built-ins or objects.
int potential_const_int = 42;       // Before transformation
int const potential_const_int = 42; // After transformation
int copy_of_value = potential_const_int;

MyClass could_be_const;       // Before transformation
MyClass const could_be_const; // After transformation
could_be_const.const_qualified_method();

// References can be declared const as well.
int &reference_value = potential_const_int;       // Before transformation
int const& reference_value = potential_const_int; // After transformation
int another_copy = reference_value;

// The similar semantics of pointers are not (yet) analyzed.
int *pointer_variable = &potential_const_int; // _NO_ ’const int *pointer_variable’ suggestion.
int last_copy = *pointer_variable;
```

The automatic code transformation is only applied to variables that are declared in single declarations. You may want to prepare your code base with *readability-isolate-declaration* first.

Note that there is the check *cppcoreguidelines-avoid-non-const-global-variables* to enforce **const** correctness on all globals.

Known Limitations

The check does not run on *C* code.

The check will not analyze templated variables or variables that are instantiation dependent. Different instantiations can result in different **const** correctness properties and in general it is not possible to find all instantiations of a template. The template might be used differently in an independent translation unit.

Pointees can not be analyzed for constness yet. The following code shows this limitation.

```
// Declare a variable that will not be modified.
int constant_value = 42;

// Declare a pointer to that variable, that does not modify either, but misses 'const'.
// Could be 'const int *pointer_to_constant = &constant_value;'
int *pointer_to_constant = &constant_value;

// Usage:
int result = 520 * 120 * (*pointer_to_constant);
```

This limitation affects the capability to add **const** to methods which is not possible, too.

Options

AnalyzeValues (default = true)

Enable or disable the analysis of ordinary value variables, like **int i = 42;**

```
// Warning
int i = 42;
// No warning
int const i = 42;

// Warning
int a[] = {42, 42, 42};
```

```
// No warning
int const a[] = {42, 42, 42};
```

AnalyzeReferences (default = true)

Enable or disable the analysis of reference variables, like **int &ref = i;**

```
int i = 42;
// Warning
int& ref = i;
// No warning
int const& ref = i;
```

WarnPointersAsValues (default = false)

This option enables the suggestion for **const** of the pointer itself. Pointer values have two possibilities to be **const**, the pointer and the value pointing to.

```
int value = 42;

// Warning
const int * pointer_variable = &value;
// No warning
const int *const pointer_variable = &value;
```

TransformValues (default = true)

Provides fixit-hints for value types that automatically add **const** if its a single declaration.

```
// Before
int value = 42;
// After
int const value = 42;

// Before
int a[] = {42, 42, 42};
// After
int const a[] = {42, 42, 42};

// Result is modified later in its life-time. No diagnostic and fixit hint will be emitted.
int result = value * 3;
result -= 10;
```

TransformReferences (default = true)

Provides fixit-hints for reference types that automatically add **const** if its a single declaration.

```
// This variable could still be a constant. But because there is a non-const reference to
// it, it can not be transformed (yet).
int value = 42;
// The reference 'ref_value' is not modified and can be made 'const int &ref_value = value;'
// Before
int &ref_value = value;
// After
int const &ref_value = value;

// Result is modified later in its life-time. No diagnostic and fixit hint will be emitted.
int result = ref_value * 3;
result -= 10;
```

TransformPointersAsValues (default = false)

Provides fixit-hints for pointers if their pointee is not changed. This does not analyze if the value-pointed-to is unchanged!

Requires 'WarnPointersAsValues' to be 'true'.

```
int value = 42;

// Before
const int * pointer_variable = &value;
// After
const int *const pointer_variable = &value;

// Before
const int * a[] = {&value, &value};
// After
const int *const a[] = {&value, &value};

// Before
int *ptr_value = &value;
// After
int *const ptr_value = &value;

int result = 100 * (*ptr_value); // Does not modify the pointer itself.
```

```
// This modification of the pointee is still allowed and not diagnosed.
*ptr_value = 0;

// The following pointer may not become a ’int *const’.
int *changing_pointee = &value;
changing_pointee = &result;
```

misc-definitions-in-headers

Finds non-extern non-inline function and variable definitions in header files, which can lead to potential ODR violations in case these headers are included from multiple translation units.

```
// Foo.h
int a = 1; // Warning: variable definition.
extern int d; // OK: extern variable.

namespace N {
  int e = 2; // Warning: variable definition.
}

// Warning: variable definition.
const char* str = "foo";

// OK: internal linkage variable definitions are ignored for now.
// Although these might also cause ODR violations, we can be less certain and
// should try to keep the false-positive rate down.
static int b = 1;
const int c = 1;
const char* const str2 = "foo";
constexpr int k = 1;

// Warning: function definition.
int g() {
  return 1;
}

// OK: inline function definition is allowed to be defined multiple times.
inline int e() {
  return 1;
}
```

```
class A {
public:
  int f1() { return 1; } // OK: implicitly inline member function definition is allowed.
  int f2();

  static int d;
};

// Warning: not an inline member function definition.
int A::f2() { return 1; }

// OK: class static data member declaration is allowed.
int A::d = 1;

// OK: function template is allowed.
template<typename T>
T f3() {
  T a = 1;
  return a;
}

// Warning: full specialization of a function template is not allowed.
template <>
int f3() {
  int a = 1;
  return a;
}

template <typename T>
struct B {
  void f1();
};

// OK: member function definition of a class template is allowed.
template <typename T>
void B<T>::f1() {}

class CE {
  constexpr static int i = 5; // OK: inline variable definition.
};
```

```
inline int i = 5; // OK: inline variable definition.

constexpr int f10() { return 0; } // OK: constexpr function implies inline.

// OK: C++14 variable templates are inline.
template <class T>
constexpr T pi = T(3.1415926L);
```

Options

HeaderFileExtensions
A comma-separated list of filename extensions of header files (the filename extensions should not include "." prefix). Default is "h,hh,hpp,hxx". For header files without an extension, use an empty string (if there are no other desired extensions) or leave an empty element in the list. E.g., "h,hh,hpp,hxx," (note the trailing comma).

UseHeaderFileExtension
When *true*, the check will use the file extension to distinguish header files. Default is *true*.

misc-misleading-bidirectional
Warn about unterminated bidirectional unicode sequence, detecting potential attack as described in the *Trojan Source* attack.

Example:

```
#include <iostream>

int main() {
   bool isAdmin = false;
   /*<?> } <?>if (isAdmin)<?> <?> begin admins only */
      std::cout << "You are an admin.\n";
   /* end admins only <?> { <?>*/
   return 0;
}
```

misc-misleading-identifier
Finds identifiers that contain Unicode characters with right-to-left direction, which can be confusing as they may change the understanding of a whole statement line, as described in *Trojan Source*.

An example of such misleading code follows:

```
#include <stdio.h>

short int <?> = (short int)0;
short int <?> = (short int)12345;

int main() {
  int <?> = <?>; // a local variable, set to zero?
  printf("<?> is %d\n", <?>);
  printf("<?> is %d\n", <?>);
}
```

misc-misplaced-const

This check diagnoses when a **const** qualifier is applied to a **typedef**/ **using** to a pointer type rather than to the pointee, because such constructs are often misleading to developers because the **const** applies to the pointer rather than the pointee.

For instance, in the following code, the resulting type is **int * const** rather than **const int ***:

```
typedef int *int_ptr;
void f(const int_ptr ptr) {
  *ptr = 0; // potentially quite unexpectedly the int can be modified here
  ptr = 0; // does not compile
}
```

The check does not diagnose when the underlying **typedef**/**using** type is a pointer to a **const** type or a function pointer type. This is because the **const** qualifier is less likely to be mistaken because it would be redundant (or disallowed) on the underlying pointee type.

misc-new-delete-overloads

cert-dcl54-cpp redirects here as an alias for this check.

The check flags overloaded operator **new()** and operator **delete()** functions that do not have a corresponding free store function defined within the same scope. For instance, the check will flag a class implementation of a non-placement operator **new()** when the class does not also define a non-placement operator **delete()** function as well.

The check does not flag implicitly-defined operators, deleted or private operators, or placement operators.

This check corresponds to CERT C++ Coding Standard rule *DCL54-CPP. Overload allocation and*

deallocation functions as a pair in the same scope.

misc-no-recursion

Finds strongly connected functions (by analyzing the call graph for SCC's (Strongly Connected Components) that are loops), diagnoses each function in the cycle, and displays one example of a possible call graph loop (recursion).

References:

- CERT C++ Coding Standard rule *DCL56-CPP. Avoid cycles during initialization of static objects*.
- JPL Institutional Coding Standard for the C Programming Language (JPL DOCID D-60411) rule *2.4 Do not use direct or indirect recursion*.
- OpenCL Specification, Version 1.2 rule *6.9 Restrictions: i. Recursion is not supported.*.

Limitations:

- The check does not handle calls done through function pointers
- The check does not handle C++ destructors

misc-non-copyable-objects

cert-fio38-c redirects here as an alias for this check.

The check flags dereferences and non-pointer declarations of objects that are not meant to be passed by value, such as C FILE objects or POSIX **pthread_mutex_t** objects.

This check corresponds to CERT C++ Coding Standard rule *FIO38-C. Do not copy a FILE object*.

misc-non-private-member-variables-in-classes

cppcoreguidelines-non-private-member-variables-in-classes redirects here as an alias for this check.

Finds classes that contain non-static data members in addition to user-declared non-static member functions and diagnose all data members declared with a non-**public** access specifier. The data members should be declared as **private** and accessed through member functions instead of exposed to derived classes or class consumers.

Options

IgnoreClassesWithAllMemberVariablesBeingPublic
Allows to completely ignore classes if **all** the member variables in that class a declared with a **public** access specifier.

IgnorePublicMemberVariables
Allows to ignore (not diagnose) **all** the member variables declared with a **public** access specifier.

misc-redundant-expression
Detect redundant expressions which are typically errors due to copy-paste.

Depending on the operator expressions may be

- redundant,
- always **true**,
- always **false**,
- always a constant (zero or one).

Examples:

```
((x+1) | (x+1))            // (x+1) is redundant
(p->x == p->x)               // always true
(p->x < p->x)              // always false
(speed - speed + 1 == 12)   // speed - speed is always zero
```

misc-static-assert
cert-dcl03-c redirects here as an alias for this check.

Replaces **assert()** with **static_assert()** if the condition is evaluable at compile time.

The condition of **static_assert()** is evaluated at compile time which is safer and more efficient.

misc-throw-by-value-catch-by-reference
cert-err09-cpp redirects here as an alias for this check. *cert-err61-cpp* redirects here as an alias for this check.

Finds violations of the rule "Throw by value, catch by reference" presented for example in "C++ Coding Standards" by H. Sutter and A. Alexandrescu, as well as the CERT C++ Coding Standard rule

ERR61-CPP. Catch exceptions by lvalue reference.

Exceptions:

- Throwing string literals will not be flagged despite being a pointer. They are not susceptible to slicing and the usage of string literals is idiomatic.
- Catching character pointers (**char**, **wchar_t**, unicode character types) will not be flagged to allow catching sting literals.
- Moved named values will not be flagged as not throwing an anonymous temporary. In this case we can be sure that the user knows that the object can't be accessed outside catch blocks handling the error.
- Throwing function parameters will not be flagged as not throwing an anonymous temporary. This allows helper functions for throwing.
- Re-throwing caught exception variables will not be flagged as not throwing an anonymous temporary. Although this can usually be done by just writing **throw;** it happens often enough in real code.

Options

CheckThrowTemporaries
Triggers detection of violations of the CERT recommendation ERR09-CPP. Throw anonymous temporaries. Default is *true*.

WarnOnLargeObject
Also warns for any large, trivial object caught by value. Catching a large object by value is not dangerous but affects the performance negatively. The maximum size of an object allowed to be caught without warning can be set using the *MaxSize* option. Default is *false*.

MaxSize
Determines the maximum size of an object allowed to be caught without warning. Only applicable if *WarnOnLargeObject* is set to *true*. If the option is set by the user to *std::numeric_limits<uint64_t>::max()* then it reverts to the default value. Default is the size of *size_t*.

misc-unconventional-assign-operator
Finds declarations of assign operators with the wrong return and/or argument types and definitions with

good return type but wrong **return** statements.

- The return type must be **Class&**.
- The assignment may be from the class type by value, const lvalue reference, non-const rvalue reference, or from a completely different type (e.g. **int**).
- Private and deleted operators are ignored.
- The operator must always return ***this**.

misc-uniqueptr-reset-release

Find and replace **unique_ptr::reset(release())** with **std::move()**.

Example:

```
std::unique_ptr<Foo> x, y;
x.reset(y.release()); -> x = std::move(y);
```

If **y** is already rvalue, **std::move()** is not added. **x** and **y** can also be **std::unique_ptr<Foo>***.

Options

IncludeStyle

A string specifying which include-style is used, *llvm* or *google*. Default is *llvm*.

misc-unused-alias-decls

Finds unused namespace alias declarations.

```
namespace my_namespace {
class C {};
}
namespace unused_alias = ::my_namespace;
```

misc-unused-parameters

Finds unused function parameters. Unused parameters may signify a bug in the code (e.g. when a different parameter is used instead). The suggested fixes either comment parameter name out or remove the parameter completely, if all callers of the function are in the same translation unit and can be updated.

The check is similar to the **-Wunused-parameter** compiler diagnostic and can be used to prepare a codebase to enabling of that diagnostic. By default the check is more permissive (see *StrictMode*).

```
void a(int i) { /*some code that doesn't use `i`*/ }

// becomes

void a(int  /*i*/) { /*some code that doesn't use `i`*/ }

static void staticFunctionA(int i);
static void staticFunctionA(int i) { /*some code that doesn't use `i`*/ }

// becomes

static void staticFunctionA()
static void staticFunctionA() { /*some code that doesn't use `i`*/ }
```

Options

StrictMode
When *false* (default value), the check will ignore trivially unused parameters, i.e. when the corresponding function has an empty body (and in case of constructors - no constructor initializers). When the function body is empty, an unused parameter is unlikely to be unnoticed by a human reader, and there's basically no place for a bug to hide.

misc-unused-using-decls
Finds unused **using** declarations.

Example:

```
namespace n { class C; }
using n::C;  // Never actually used.
```

modernize-avoid-bind
The check finds uses of **std::bind** and **boost::bind** and replaces them with lambdas. Lambdas will use value-capture unless reference capture is explicitly requested with **std::ref** or **boost::ref**.

It supports arbitrary callables including member functions, function objects, and free functions, and all variations thereof. Anything that you can pass to the first argument of **bind** should be diagnosable. Currently, the only known case where a fix-it is unsupported is when the same placeholder is specified

multiple times in the parameter list.

Given:

```
int add(int x, int y) { return x + y; }
```

Then:

```
void f() {
  int x = 2;
  auto clj = std::bind(add, x, _1);
}
```

is replaced by:

```
void f() {
  int x = 2;
  auto clj = [=](auto && arg1) { return add(x, arg1); };
}
```

std::bind can be hard to read and can result in larger object files and binaries due to type information that will not be produced by equivalent lambdas.

Options

PermissiveParameterList

If the option is set to *true*, the check will append **auto&&...** to the end of every placeholder parameter list. Without this, it is possible for a fix-it to perform an incorrect transformation in the case where the result of the **bind** is used in the context of a type erased functor such as **std::function** which allows mismatched arguments. For example:

```
int add(int x, int y) { return x + y; }
int foo() {
  std::function<int(int,int)> ignore_args = std::bind(add, 2, 2);
  return ignore_args(3, 3);
}
```

is valid code, and returns *4*. The actual values passed to **ignore_args** are simply ignored. Without **PermissiveParameterList**, this would be transformed into

```
int add(int x, int y) { return x + y; }
int foo() {
  std::function<int(int,int)> ignore_args = [] { return add(2, 2); }
  return ignore_args(3, 3);
}
```

which will *not* compile, since the lambda does not contain an **operator()** that accepts 2 arguments. With permissive parameter list, it instead generates

```
int add(int x, int y) { return x + y; }
int foo() {
  std::function<int(int,int)> ignore_args = [](auto&&...) { return add(2, 2); }
  return ignore_args(3, 3);
}
```

which is correct.

This check requires using C++14 or higher to run.

modernize-avoid-c-arrays

cppcoreguidelines-avoid-c-arrays redirects here as an alias for this check.

hicpp-avoid-c-arrays redirects here as an alias for this check.

Finds C-style array types and recommend to use **std::array<>** / **std::vector<>**. All types of C arrays are diagnosed.

However, fix-it are potentially dangerous in header files and are therefore not emitted right now.

```
int a[] = {1, 2}; // warning: do not declare C-style arrays, use std::array<> instead

int b[1]; // warning: do not declare C-style arrays, use std::array<> instead

void foo() {
  int c[b[0]]; // warning: do not declare C VLA arrays, use std::vector<> instead
}

template <typename T, int Size>
class array {
  T d[Size]; // warning: do not declare C-style arrays, use std::array<> instead
```

```
  int e[1]; // warning: do not declare C-style arrays, use std::array<> instead
};

array<int[4], 2> d; // warning: do not declare C-style arrays, use std::array<> instead

using k = int[4]; // warning: do not declare C-style arrays, use std::array<> instead
```

However, the **extern "C"** code is ignored, since it is common to share such headers between C code, and C++ code.

```
// Some header
extern "C" {

int f[] = {1, 2}; // not diagnosed

int j[1]; // not diagnosed

inline void bar() {
  {
    int j[j[0]]; // not diagnosed
  }
}

}
```

Similarly, the **main()** function is ignored. Its second and third parameters can be either **char* argv[]** or **char** argv**, but cannot be **std::array<>**.

modernize-concat-nested-namespaces

Checks for use of nested namespaces such as **namespace a { namespace b { ... } }** and suggests changing to the more concise syntax introduced in C++17: **namespace a::b { ... }**. Inline namespaces are not modified.

For example:

```
namespace n1 {
namespace n2 {
void t();
}
}
```

```
namespace n3 {
namespace n4 {
namespace n5 {
void t();
}
}
namespace n6 {
namespace n7 {
void t();
}
}
}
```

Will be modified to:

```
namespace n1::n2 {
void t();
}

namespace n3 {
namespace n4::n5 {
void t();
}
namespace n6::n7 {
void t();
}
}
```

modernize-deprecated-headers

Some headers from C library were deprecated in C++ and are no longer welcome in C++ codebases. Some have no effect in C++. For more details refer to the C++ 14 Standard [depr.c.headers] section.

This check replaces C standard library headers with their C++ alternatives and removes redundant ones.

```
// C++ source file...
#include <assert.h>
#include <stdbool.h>

// becomes
```

```
#include <cassert>
// No 'stdbool.h' here.
```

Important note: the Standard doesn't guarantee that the C++ headers declare all the same functions in the global namespace. The check in its current form can break the code that uses library symbols from the global namespace.

- <*assert.h*>
- <*complex.h*>
- <*ctype.h*>
- <*errno.h*>
- <*fenv.h*> // deprecated since C++11
- <*float.h*>
- <*inttypes.h*>
- <*limits.h*>
- <*locale.h*>
- <*math.h*>
- <*setjmp.h*>
- <*signal.h*>
- <*stdarg.h*>
- <*stddef.h*>
- <*stdint.h*>
- <*stdio.h*>
- <*stdlib.h*>

- <*string.h*>
- <*tgmath.h*> // deprecated since C++11
- <*time.h*>
- <*uchar.h*> // deprecated since C++11
- <*wchar.h*>
- <*wctype.h*>

If the specified standard is older than C++11 the check will only replace headers deprecated before C++11, otherwise -- every header that appeared in the previous list.

These headers don't have effect in C++:

- <*iso646.h*>
- <*stdalign.h*>
- <*stdbool.h*>

The checker ignores *include* directives within *extern "C" { ... }* blocks, since a library might want to expose some API for C and C++ libraries.

```
// C++ source file...
extern "C" {
#include <assert.h>  // Left intact.
#include <stdbool.h> // Left intact.
}
```

Options

CheckHeaderFile

clang-tidy cannot know if the header file included by the currently analyzed C++ source file is not included by any other C source files. Hence, to omit false-positives and wrong fixit-hints, we ignore emitting reports into header files. One can set this option to *true* if they know that the header files in the project are only used by C++ source file. Default is *false*.

modernize-deprecated-ios-base-aliases

Detects usage of the deprecated member types of **std::ios_base** and replaces those that have a non-deprecated equivalent.

Deprecated member type	Replacement
std::ios_base::io_state	**std::ios_base::iostate**
std::ios_base::open_mode	**std::ios_base::openmode**
std::ios_base::seek_dir	**std::ios_base::seekdir**
std::ios_base::streamoff	
std::ios_base::streampos	

modernize-loop-convert

This check converts **for(...; ...; ...)** loops to use the new range-based loops in C++11.

Three kinds of loops can be converted:

- Loops over statically allocated arrays.
- Loops over containers, using iterators.
- Loops over array-like containers, using **operator[]** and **at()**.

MinConfidence option

risky

In loops where the container expression is more complex than just a reference to a declared expression (a variable, function, enum, etc.), and some part of it appears elsewhere in the loop, we lower our confidence in the transformation due to the increased risk of changing semantics. Transformations for these loops are marked as *risky*, and thus will only be converted if the minimum required confidence level is set to *risky*.

```
int arr[10][20];
int l = 5;
```

```
for (int j = 0; j < 20; ++j)
  int k = arr[l][j] + l; // using l outside arr[l] is considered risky

for (int i = 0; i < obj.getVector().size(); ++i)
  obj.foo(10); // using 'obj' is considered risky
```

See *Range-based loops evaluate end() only once* for an example of an incorrect transformation when the minimum required confidence level is set to *risky*.

reasonable (Default)

If a loop calls **.end()** or **.size()** after each iteration, the transformation for that loop is marked as *reasonable*, and thus will be converted if the required confidence level is set to *reasonable* (default) or lower.

```
// using size() is considered reasonable
for (int i = 0; i < container.size(); ++i)
  cout << container[i];
```

safe

Any other loops that do not match the above criteria to be marked as *risky* or *reasonable* are marked *safe*, and thus will be converted if the required confidence level is set to *safe* or lower.

```
int arr[] = {1,2,3};

for (int i = 0; i < 3; ++i)
  cout << arr[i];
```

Example

Original:

```
const int N = 5;
int arr[] = {1,2,3,4,5};
vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);

// safe conversion
for (int i = 0; i < N; ++i)
  cout << arr[i];
```

```
// reasonable conversion
for (vector<int>::iterator it = v.begin(); it != v.end(); ++it)
  cout << *it;

// reasonable conversion
for (int i = 0; i < v.size(); ++i)
  cout << v[i];
```

After applying the check with minimum confidence level set to *reasonable* (default):

```
const int N = 5;
int arr[] = {1,2,3,4,5};
vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);

// safe conversion
for (auto & elem : arr)
  cout << elem;

// reasonable conversion
for (auto & elem : v)
  cout << elem;

// reasonable conversion
for (auto & elem : v)
  cout << elem;
```

Reverse Iterator Support

The converter is also capable of transforming iterator loops which use **rbegin** and **rend** for looping backwards over a container. Out of the box this will automatically happen in C++20 mode using the **ranges** library, however the check can be configured to work without C++20 by specifying a function to reverse a range and optionally the header file where that function lives.

UseCxx20ReverseRanges

When set to true convert loops when in C++20 or later mode using **std::ranges::reverse_view**. Default value is **true**.

MakeReverseRangeFunction

Specify the function used to reverse an iterator pair, the function should accept a class with **rbegin** and **rend** methods and return a class with **begin** and **end** methods that call the **rbegin** and **rend** methods respectively. Common examples are **ranges::reverse_view** and **llvm::reverse**. Default value is an empty string.

MakeReverseRangeHeader

Specifies the header file where *MakeReverseRangeFunction* is declared. For the previous examples this option would be set to **range/v3/view/reverse.hpp** and **llvm/ADT/STLExtras.h** respectively. If this is an empty string and *MakeReverseRangeFunction* is set, the check will proceed on the assumption that the function is already available in the translation unit. This can be wrapped in angle brackets to signify to add the include as a system include. Default value is an empty string.

IncludeStyle

A string specifying which include-style is used, *llvm* or *google*. Default is *llvm*.

Limitations

There are certain situations where the tool may erroneously perform transformations that remove information and change semantics. Users of the tool should be aware of the behavior and limitations of the check outlined by the cases below.

Comments inside loop headers

Comments inside the original loop header are ignored and deleted when transformed.

```
for (int i = 0; i < N; /* This will be deleted */ ++i) { }
```

Range-based loops evaluate end() only once

The C++11 range-based for loop calls **.end()** only once during the initialization of the loop. If in the original loop **.end()** is called after each iteration the semantics of the transformed loop may differ.

```
// The following is semantically equivalent to the C++11 range-based for loop,
// therefore the semantics of the header will not change.
for (iterator it = container.begin(), e = container.end(); it != e; ++it) { }

// Instead of calling .end() after each iteration, this loop will be
// transformed to call .end() only once during the initialization of the loop,
// which may affect semantics.
for (iterator it = container.begin(); it != container.end(); ++it) { }
```

As explained above, calling member functions of the container in the body of the loop is

considered *risky*. If the called member function modifies the container the semantics of the converted loop will differ due to **.end()** being called only once.

```
bool flag = false;
for (vector<T>::iterator it = vec.begin(); it != vec.end(); ++it) {
 // Add a copy of the first element to the end of the vector.
 if (!flag) {
  // This line makes this transformation 'risky'.
  vec.push_back(*it);
  flag = true;
 }
 cout << *it;
}
```

The original code above prints out the contents of the container including the newly added element while the converted loop, shown below, will only print the original contents and not the newly added element.

```
bool flag = false;
for (auto & elem : vec) {
 // Add a copy of the first element to the end of the vector.
 if (!flag) {
  // This line makes this transformation 'risky'
  vec.push_back(elem);
  flag = true;
 }
 cout << elem;
}
```

Semantics will also be affected if **.end()** has side effects. For example, in the case where calls to **.end()** are logged the semantics will change in the transformed loop if **.end()** was originally called after each iteration.

```
iterator end() {
 num_of_end_calls++;
 return container.end();
}
```

Overloaded operator->() with side effects

Similarly, if **operator->()** was overloaded to have side effects, such as logging, the semantics will

change. If the iterator's **operator->()** was used in the original loop it will be replaced with **<container element>.<member>** instead due to the implicit dereference as part of the range-based for loop. Therefore any side effect of the overloaded **operator->()** will no longer be performed.

```
for (iterator it = c.begin(); it != c.end(); ++it) {
  it->func(); // Using operator->()
}
// Will be transformed to:
for (auto & elem : c) {
  elem.func(); // No longer using operator->()
}
```

Pointers and references to containers

While most of the check's risk analysis is dedicated to determining whether the iterator or container was modified within the loop, it is possible to circumvent the analysis by accessing and modifying the container through a pointer or reference.

If the container were directly used instead of using the pointer or reference the following transformation would have only been applied at the *risky* level since calling a member function of the container is considered *risky*. The check cannot identify expressions associated with the container that are different than the one used in the loop header, therefore the transformation below ends up being performed at the *safe* level.

```
vector<int> vec;

vector<int> *ptr = &vec;
vector<int> &ref = vec;

for (vector<int>::iterator it = vec.begin(), e = vec.end(); it != e; ++it) {
  if (!flag) {
    // Accessing and modifying the container is considered risky, but the risk
    // level is not raised here.
    ptr->push_back(*it);
    ref.push_back(*it);
    flag = true;
  }
}
```

OpenMP

As range-based for loops are only available since OpenMP 5, this check should not be used on code

with a compatibility requirement of OpenMP prior to version 5. It is **intentional** that this check does not make any attempts to exclude incorrect diagnostics on OpenMP for loops prior to OpenMP 5.

To prevent this check to be applied (and to break) OpenMP for loops but still be applied to non-OpenMP for loops the usage of **NOLINT** (see *Suppressing Undesired Diagnostics*) on the specific for loops is recommended.

modernize-macro-to-enum

Replaces groups of adjacent macros with an unscoped anonymous enum. Using an unscoped anonymous enum ensures that everywhere the macro token was used previously, the enumerator name may be safely used.

This check can be used to enforce the C++ core guideline *Enum.1: Prefer enumerations over macros*, within the constraints outlined below.

Potential macros for replacement must meet the following constraints:

- Macros must expand only to integral literal tokens or expressions of literal tokens. The expression may contain any of the unary operators **-**, **+**, **~** or **!**, any of the binary operators **,**, **-**, **+**, *****, /, **%**, **&**, |, ^, <, >, <=, >=, ==, **!=**, ||, **&&**, <<, >> or <=>, the ternary operator **?:** and its *GNU extension*. Parenthesized expressions are also recognized. This recognizes most valid expressions. In particular, expressions with the **sizeof** operator are not recognized.

- Macros must be defined on sequential source file lines, or with only comment lines in between macro definitions.

- Macros must all be defined in the same source file.

- Macros must not be defined within a conditional compilation block. (Conditional include guards are exempt from this constraint.)

- Macros must not be defined adjacent to other preprocessor directives.

- Macros must not be used in any conditional preprocessing directive.

- Macros must not be used as arguments to other macros.

- Macros must not be undefined.

- Macros must be defined at the top-level, not inside any declaration or definition.

Each cluster of macros meeting the above constraints is presumed to be a set of values suitable for replacement by an anonymous enum. From there, a developer can give the anonymous enum a name and continue refactoring to a scoped enum if desired. Comments on the same line as a macro definition or between subsequent macro definitions are preserved in the output. No formatting is assumed in the provided replacements, although clang-tidy can optionally format all fixes.

WARNING:

Initializing expressions are assumed to be valid initializers for an enum. C requires that enum values fit into an **int**, but this may not be the case for some accepted constant expressions. For instance **1 << 40** will not fit into an **int** when the size of an **int** is 32 bits.

Examples:

```
#define RED   0xFF0000
#define GREEN 0x00FF00
#define BLUE  0x0000FF

#define TM_NONE (-1) // No method selected.
#define TM_ONE 1    // Use tailored method one.
#define TM_TWO 2    // Use tailored method two.  Method two
                    // is preferable to method one.
#define TM_THREE 3  // Use tailored method three.
```

becomes

```
enum {
RED = 0xFF0000,
GREEN = 0x00FF00,
BLUE = 0x0000FF
};

enum {
TM_NONE = (-1), // No method selected.
TM_ONE = 1,    // Use tailored method one.
TM_TWO = 2,    // Use tailored method two.  Method two
                // is preferable to method one.
TM_THREE = 3  // Use tailored method three.
};
```

modernize-make-shared

This check finds the creation of **std::shared_ptr** objects by explicitly calling the constructor and a **new** expression, and replaces it with a call to **std::make_shared**.

```
auto my_ptr = std::shared_ptr<MyPair>(new MyPair(1, 2));

// becomes

auto my_ptr = std::make_shared<MyPair>(1, 2);
```

This check also finds calls to **std::shared_ptr::reset()** with a **new** expression, and replaces it with a call to **std::make_shared**.

```
my_ptr.reset(new MyPair(1, 2));

// becomes

my_ptr = std::make_shared<MyPair>(1, 2);
```

Options

MakeSmartPtrFunction

A string specifying the name of make-shared-ptr function. Default is *std::make_shared*.

MakeSmartPtrFunctionHeader

A string specifying the corresponding header of make-shared-ptr function. Default is *memory*.

IncludeStyle

A string specifying which include-style is used, *llvm* or *google*. Default is *llvm*.

IgnoreMacros

If set to *true*, the check will not give warnings inside macros. Default is *true*.

IgnoreDefaultInitialization

If set to non-zero, the check does not suggest edits that will transform default initialization into value initialization, as this can cause performance regressions. Default is *1*.

modernize-make-unique

This check finds the creation of **std::unique_ptr** objects by explicitly calling the constructor and a **new** expression, and replaces it with a call to **std::make_unique**, introduced in C++14.

```
auto my_ptr = std::unique_ptr<MyPair>(new MyPair(1, 2));

// becomes

auto my_ptr = std::make_unique<MyPair>(1, 2);
```

This check also finds calls to **std::unique_ptr::reset()** with a **new** expression, and replaces it with a call to **std::make_unique**.

```
my_ptr.reset(new MyPair(1, 2));

// becomes

my_ptr = std::make_unique<MyPair>(1, 2);
```

Options

MakeSmartPtrFunction
A string specifying the name of make-unique-ptr function. Default is *std::make_unique*.

MakeSmartPtrFunctionHeader
A string specifying the corresponding header of make-unique-ptr function. Default is <*memory*>.

IncludeStyle
A string specifying which include-style is used, *llvm* or *google*. Default is *llvm*.

IgnoreMacros
If set to *true*, the check will not give warnings inside macros. Default is *true*.

IgnoreDefaultInitialization
If set to non-zero, the check does not suggest edits that will transform default initialization into value initialization, as this can cause performance regressions. Default is *1*.

modernize-pass-by-value

With move semantics added to the language and the standard library updated with move constructors added for many types it is now interesting to take an argument directly by value, instead of by const-reference, and then copy. This check allows the compiler to take care of choosing the best way to construct the copy.

The transformation is usually beneficial when the calling code passes an *rvalue* and assumes the move

construction is a cheap operation. This short example illustrates how the construction of the value happens:

```
void foo(std::string s);
std::string get_str();

void f(const std::string &str) {
  foo(str);       // lvalue  -> copy construction
  foo(get_str()); // prvalue -> move construction
}
```

NOTE:

Currently, only constructors are transformed to make use of pass-by-value. Contributions that handle other situations are welcome!

Pass-by-value in constructors

Replaces the uses of const-references constructor parameters that are copied into class fields. The parameter is then moved with *std::move()*.

Since **std::move()** is a library function declared in <*utility*> it may be necessary to add this include. The check will add the include directive when necessary.

```
#include <string>

class Foo {
public:
-  Foo(const std::string &Copied, const std::string &ReadOnly)
-    : Copied(Copied), ReadOnly(ReadOnly)
+  Foo(std::string Copied, const std::string &ReadOnly)
+    : Copied(std::move(Copied)), ReadOnly(ReadOnly)
   {}

private:
  std::string Copied;
  const std::string &ReadOnly;
};

std::string get_cwd();

void f(const std::string &Path) {
```

```
  // The parameter corresponding to 'get_cwd()' is move-constructed. By
  // using pass-by-value in the Foo constructor we managed to avoid a
  // copy-construction.
  Foo foo(get_cwd(), Path);
}
```

If the parameter is used more than once no transformation is performed since moved objects have an undefined state. It means the following code will be left untouched:

```
#include <string>

void pass(const std::string &S);

struct Foo {
  Foo(const std::string &S) : Str(S) {
    pass(S);
  }

  std::string Str;
};
```

Known limitations

A situation where the generated code can be wrong is when the object referenced is modified before the assignment in the init-list through a "hidden" reference.

Example:

```
std::string s("foo");

struct Base {
  Base() {
    s = "bar";
  }
};

struct Derived : Base {
-  Derived(const std::string &S) : Field(S)
+  Derived(std::string S) : Field(std::move(S))
   { }
```

```
  std::string Field;
};

void f() {
-  Derived d(s); // d.Field holds "bar"
+  Derived d(s); // d.Field holds "foo"
}
```

Note about delayed template parsing

When delayed template parsing is enabled, constructors part of templated contexts; templated constructors, constructors in class templates, constructors of inner classes of template classes, etc., are not transformed. Delayed template parsing is enabled by default on Windows as a Microsoft extension: *Clang Compiler User's Manual - Microsoft extensions*.

Delayed template parsing can be enabled using the *-fdelayed-template-parsing* flag and disabled using *-fno-delayed-template-parsing*.

Example:

```
template <typename T> class C {
  std::string S;

public:
=  // using -fdelayed-template-parsing (default on Windows)
=  C(const std::string &S) : S(S) {}

+  // using -fno-delayed-template-parsing (default on non-Windows systems)
+  C(std::string S) : S(std::move(S)) {}
};
```

SEE ALSO:

For more information about the pass-by-value idiom, read: *Want Speed? Pass by Value*.

Options

IncludeStyle

A string specifying which include-style is used, *llvm* or *google*. Default is *llvm*.

ValuesOnly

When *true*, the check only warns about copied parameters that are already passed by value. Default

is *false*.

modernize-raw-string-literal

This check selectively replaces string literals containing escaped characters with raw string literals.

Example:

```
const char *const Quotes{"embedded \"quotes\""};
const char *const Paragraph{"Line one.\nLine two.\nLine three.\n"};
const char *const SingleLine{"Single line.\n"};
const char *const TrailingSpace{"Look here -> \n"};
const char *const Tab{"One\tTwo\n"};
const char *const Bell{"Hello!\a  And welcome!"};
const char *const Path{"C:\\Program Files\\Vendor\\Application.exe"};
const char *const RegEx{"\\w\\([a-z]\\)"};
```

becomes

```
const char *const Quotes{R"(embedded "quotes")"};
const char *const Paragraph{"Line one.\nLine two.\nLine three.\n"};
const char *const SingleLine{"Single line.\n"};
const char *const TrailingSpace{"Look here -> \n"};
const char *const Tab{"One\tTwo\n"};
const char *const Bell{"Hello!\a  And welcome!"};
const char *const Path{R"(C:\Program Files\Vendor\Application.exe)"};
const char *const RegEx{R"(\w\([a-z]\))"};
```

The presence of any of the following escapes can cause the string to be converted to a raw string literal: \\, **'**, **"**, **?**, and octal or hexadecimal escapes for printable ASCII characters.

A string literal containing only escaped newlines is a common way of writing lines of text output. Introducing physical newlines with raw string literals in this case is likely to impede readability. These string literals are left unchanged.

An escaped horizontal tab, form feed, or vertical tab prevents the string literal from being converted. The presence of a horizontal tab, form feed or vertical tab in source code is not visually obvious.

modernize-redundant-void-arg

Find and remove redundant **void** argument lists.

Examples:

Initial code	Code with applied fixes
int f(void);	**int f();**
int (*f(void))(void);	**int (*f())();**
typedef int (*f_t(void))(void);	**typedef int (*f_t())();**
void (C::*p)(void);	**void (C::*p)();**
C::C(void) {}	**C::C() {}**
C::~C(void) {}	**C::~C() {}**

modernize-replace-auto-ptr

This check replaces the uses of the deprecated class **std::auto_ptr** by **std::unique_ptr** (introduced in C++11). The transfer of ownership, done by the copy-constructor and the assignment operator, is changed to match **std::unique_ptr** usage by using explicit calls to **std::move()**.

Migration example:

```
-void take_ownership_fn(std::auto_ptr<int> int_ptr);
+void take_ownership_fn(std::unique_ptr<int> int_ptr);

 void f(int x) {
-  std::auto_ptr<int> a(new int(x));
-  std::auto_ptr<int> b;
+  std::unique_ptr<int> a(new int(x));
+  std::unique_ptr<int> b;
```

```
- b = a;
- take_ownership_fn(b);
+ b = std::move(a);
+ take_ownership_fn(std::move(b));
 }
```

Since **std::move()** is a library function declared in **<utility>** it may be necessary to add this include. The check will add the include directive when necessary.

Known Limitations

- If headers modification is not activated or if a header is not allowed to be changed this check will produce broken code (compilation error), where the headers' code will stay unchanged while the code using them will be changed.

- Client code that declares a reference to an **std::auto_ptr** coming from code that can't be migrated (such as a header coming from a 3rd party library) will produce a compilation error after migration. This is because the type of the reference will be changed to **std::unique_ptr** but the type returned by the library won't change, binding a reference to **std::unique_ptr** from an **std::auto_ptr**. This pattern doesn't make much sense and usually **std::auto_ptr** are stored by value (otherwise what is the point in using them instead of a reference or a pointer?).

```
// <3rd-party header...>
std::auto_ptr<int> get_value();
const std::auto_ptr<int> & get_ref();

// <calling code (with migration)...>
-std::auto_ptr<int> a(get_value());
+std::unique_ptr<int> a(get_value()); // ok, unique_ptr constructed from auto_ptr

-const std::auto_ptr<int> & p = get_ptr();
+const std::unique_ptr<int> & p = get_ptr(); // won't compile
```

- Non-instantiated templates aren't modified.

```
template <typename X>
void f() {
  std::auto_ptr<X> p;
}
```

```
// only 'f<int>()' (or similar) will trigger the replacement.
```

Options

IncludeStyle
A string specifying which include-style is used, *llvm* or *google*. Default is *llvm*.

modernize-replace-disallow-copy-and-assign-macro

Finds macro expansions of **DISALLOW_COPY_AND_ASSIGN(Type)** and replaces them with a deleted copy constructor and a deleted assignment operator.

Before the **delete** keyword was introduced in C++11 it was common practice to declare a copy constructor and an assignment operator as private members. This effectively makes them unusable to the public API of a class.

With the advent of the **delete** keyword in C++11 we can abandon the **private** access of the copy constructor and the assignment operator and delete the methods entirely.

When running this check on a code like this:

```
class Foo {
private:
  DISALLOW_COPY_AND_ASSIGN(Foo);
};
```

It will be transformed to this:

```
class Foo {
private:
  Foo(const Foo &) = delete;
  const Foo &operator=(const Foo &) = delete;
};
```

Known Limitations

- Notice that the migration example above leaves the **private** access specification untouched. You might want to run the check *modernize-use-equals-delete* to get warnings for deleted functions in private sections.

Options

MacroName

A string specifying the macro name whose expansion will be replaced. Default is *DISALLOW_COPY_AND_ASSIGN*.

See: *https://en.cppreference.com/w/cpp/language/function#Deleted_functions*

modernize-replace-random-shuffle

This check will find occurrences of **std::random_shuffle** and replace it with **std::shuffle**. In C++17 **std::random_shuffle** will no longer be available and thus we need to replace it.

Below are two examples of what kind of occurrences will be found and two examples of what it will be replaced with.

```
std::vector<int> v;

// First example
std::random_shuffle(vec.begin(), vec.end());

// Second example
std::random_shuffle(vec.begin(), vec.end(), randomFunc);
```

Both of these examples will be replaced with:

```
std::shuffle(vec.begin(), vec.end(), std::mt19937(std::random_device()()));
```

The second example will also receive a warning that **randomFunc** is no longer supported in the same way as before so if the user wants the same functionality, the user will need to change the implementation of the **randomFunc**.

One thing to be aware of here is that **std::random_device** is quite expensive to initialize. So if you are using the code in a performance critical place, you probably want to initialize it elsewhere. Another thing is that the seeding quality of the suggested fix is quite poor: **std::mt19937** has an internal state of 624 32-bit integers, but is only seeded with a single integer. So if you require higher quality randomness, you should consider seeding better, for example:

```
std::shuffle(v.begin(), v.end(), []() {
  std::mt19937::result_type seeds[std::mt19937::state_size];
  std::random_device device;
  std::uniform_int_distribution<typename std::mt19937::result_type> dist;
  std::generate(std::begin(seeds), std::end(seeds), [&] { return dist(device); });
```

```
  std::seed_seq seq(std::begin(seeds), std::end(seeds));
  return std::mt19937(seq);
 }());
```

modernize-return-braced-init-list

Replaces explicit calls to the constructor in a return with a braced initializer list. This way the return type is not needlessly duplicated in the function definition and the return statement.

```
Foo bar() {
  Baz baz;
  return Foo(baz);
}

// transforms to:

Foo bar() {
  Baz baz;
  return {baz};
}
```

modernize-shrink-to-fit

Replace copy and swap tricks on shrinkable containers with the **shrink_to_fit()** method call.

The **shrink_to_fit()** method is more readable and more effective than the copy and swap trick to reduce the capacity of a shrinkable container. Note that, the **shrink_to_fit()** method is only available in C++11 and up.

modernize-unary-static-assert

The check diagnoses any **static_assert** declaration with an empty string literal and provides a fix-it to replace the declaration with a single-argument **static_assert** declaration.

The check is only applicable for C++17 and later code.

The following code:

```
void f_textless(int a) {
  static_assert(sizeof(a) <= 10, "");
}
```

is replaced by:

```
void f_textless(int a) {
  static_assert(sizeof(a) <= 10);
}
```

modernize-use-auto

This check is responsible for using the **auto** type specifier for variable declarations to *improve code readability and maintainability*. For example:

```
std::vector<int>::iterator I = my_container.begin();

// transforms to:

auto I = my_container.begin();
```

The **auto** type specifier will only be introduced in situations where the variable type matches the type of the initializer expression. In other words **auto** should deduce the same type that was originally spelled in the source. However, not every situation should be transformed:

```
int val = 42;
InfoStruct &I = SomeObject.getInfo();

// Should not become:

auto val = 42;
auto &I = SomeObject.getInfo();
```

In this example using **auto** for builtins doesn't improve readability. In other situations it makes the code less self-documenting impairing readability and maintainability. As a result, **auto** is used only introduced in specific situations described below.

Iterators

Iterator type specifiers tend to be long and used frequently, especially in loop constructs. Since the functions generating iterators have a common format, the type specifier can be replaced without obscuring the meaning of code while improving readability and maintainability.

```
for (std::vector<int>::iterator I = my_container.begin(),
                                E = my_container.end();
     I != E; ++I) {
}
```

```
// becomes

for (auto I = my_container.begin(), E = my_container.end(); I != E; ++I) {
}
```

The check will only replace iterator type-specifiers when all of the following conditions are satisfied:

- The iterator is for one of the standard containers in **std** namespace:
  - **array**
  - **deque**
  - **forward_list**
  - **list**
  - **vector**
  - **map**
  - **multimap**
  - **set**
  - **multiset**
  - **unordered_map**
  - **unordered_multimap**
  - **unordered_set**
  - **unordered_multiset**
  - **queue**
  - **priority_queue**

- **stack**

- The iterator is one of the possible iterator types for standard containers:
  - **iterator**
  - **reverse_iterator**
  - **const_iterator**
  - **const_reverse_iterator**
- In addition to using iterator types directly, typedefs or other ways of referring to those types are also allowed. However, implementation-specific types for which a type like **std::vector<int>::iterator** is itself a typedef will not be transformed. Consider the following examples:

```
// The following direct uses of iterator types will be transformed.
std::vector<int>::iterator I = MyVec.begin();
{
 using namespace std;
 list<int>::iterator I = MyList.begin();
}

// The type specifier for J would transform to auto since it's a typedef
// to a standard iterator type.
typedef std::map<int, std::string>::const_iterator map_iterator;
map_iterator J = MyMap.begin();

// The following implementation-specific iterator type for which
// std::vector<int>::iterator could be a typedef would not be transformed.
__gnu_cxx::__normal_iterator<int*, std::vector> K = MyVec.begin();
```

- The initializer for the variable being declared is not a braced initializer list. Otherwise, use of **auto** would cause the type of the variable to be deduced as **std::initializer_list**.

New expressions

Frequently, when a pointer is declared and initialized with **new**, the pointee type is written twice: in the declaration type and in the **new** expression. In this case, the declaration type can be replaced with **auto** improving readability and maintainability.

```
TypeName *my_pointer = new TypeName(my_param);

// becomes

auto *my_pointer = new TypeName(my_param);
```

The check will also replace the declaration type in multiple declarations, if the following conditions are satisfied:

- All declared variables have the same type (i.e. all of them are pointers to the same type).
- All declared variables are initialized with a **new** expression.
- The types of all the new expressions are the same than the pointee of the declaration type.

```
TypeName *my_first_pointer = new TypeName, *my_second_pointer = new TypeName;

// becomes

auto *my_first_pointer = new TypeName, *my_second_pointer = new TypeName;
```

Cast expressions

Frequently, when a variable is declared and initialized with a cast, the variable type is written twice: in the declaration type and in the cast expression. In this case, the declaration type can be replaced with **auto** improving readability and maintainability.

```
TypeName *my_pointer = static_cast<TypeName>(my_param);

// becomes

auto *my_pointer = static_cast<TypeName>(my_param);
```

The check handles **static_cast**, **dynamic_cast**, **const_cast**, **reinterpret_cast**, functional casts, C-style casts and function templates that behave as casts, such as **llvm::dyn_cast**, **boost::lexical_cast** and **gsl::narrow_cast**. Calls to function templates are considered to behave as casts if the first template argument is explicit and is a type, and the function returns that type, or a pointer or reference to it.

Known Limitations

- If the initializer is an explicit conversion constructor, the check will not replace the type specifier even though it would be safe to do so.

- User-defined iterators are not handled at this time.

Options

MinTypeNameLength

If the option is set to non-zero (default *5*), the check will ignore type names having a length less than the option value. The option affects expressions only, not iterators. Spaces between multi-lexeme type names (**long int**) are considered as one. If the *RemoveStars* option (see below) is set to *true*, then ***s** in the type are also counted as a part of the type name.

```
// MinTypeNameLength = 0, RemoveStars=0

int a = static_cast<int>(foo());           // ---> auto a = ...
// length(bool *) = 4
bool *b = new bool;                        // ---> auto *b = ...
unsigned c = static_cast<unsigned>(foo());  // ---> auto c = ...

// MinTypeNameLength = 5, RemoveStars=0

int a = static_cast<int>(foo());               // ---> int  a = ...
bool b = static_cast<bool>(foo());             // ---> bool b = ...
bool *pb = static_cast<bool*>(foo());          // ---> bool *pb = ...
unsigned c = static_cast<unsigned>(foo());     // ---> auto c = ...
// length(long <on-or-more-spaces> int) = 8
long int d = static_cast<long int>(foo());     // ---> auto d = ...

// MinTypeNameLength = 5, RemoveStars=1

int a = static_cast<int>(foo());               // ---> int  a = ...
// length(int * * ) = 5
int **pa = static_cast<int**>(foo());          // ---> auto pa = ...
bool b = static_cast<bool>(foo());             // ---> bool b = ...
bool *pb = static_cast<bool*>(foo());          // ---> auto pb = ...
unsigned c = static_cast<unsigned>(foo());     // ---> auto c = ...
long int d = static_cast<long int>(foo());     // ---> auto d = ...
```

RemoveStars

If the option is set to *true* (default is *false*), the check will remove stars from the non-typedef pointer types when replacing type names with **auto**. Otherwise, the check will leave stars. For example:

```
TypeName *my_first_pointer = new TypeName, *my_second_pointer = new TypeName;

// RemoveStars = 0

auto *my_first_pointer = new TypeName, *my_second_pointer = new TypeName;

// RemoveStars = 1

auto my_first_pointer = new TypeName, my_second_pointer = new TypeName;
```

modernize-use-bool-literals

Finds integer literals which are cast to **bool**.

```
bool p = 1;
bool f = static_cast<bool>(1);
std::ios_base::sync_with_stdio(0);
bool x = p ? 1 : 0;

// transforms to

bool p = true;
bool f = true;
std::ios_base::sync_with_stdio(false);
bool x = p ? true : false;
```

Options

IgnoreMacros

If set to *true*, the check will not give warnings inside macros. Default is *true*.

modernize-use-default

This check has been renamed to *modernize-use-equals-default*.

modernize-use-default-member-init

This check converts constructors' member initializers into the new default member initializers in C++11. Other member initializers that match the default member initializer are removed. This can

reduce repeated code or allow use of '= default'.

```
struct A {
  A() : i(5), j(10.0) {}
  A(int i) : i(i), j(10.0) {}
  int i;
  double j;
};

// becomes

struct A {
  A() {}
  A(int i) : i(i) {}
  int i{5};
  double j{10.0};
};
```

NOTE:

Only converts member initializers for built-in types, enums, and pointers. The *readability-redundant-member-init* check will remove redundant member initializers for classes.

Options

UseAssignment

If this option is set to *true* (default is *false*), the check will initialize members with an assignment. For example:

```
struct A {
  A() {}
  A(int i) : i(i) {}
  int i = 5;
  double j = 10.0;
};
```

IgnoreMacros

If this option is set to *true* (default is *true*), the check will not warn about members declared inside macros.

modernize-use-emplace

The check flags insertions to an STL-style container done by calling the **push_back** method with an explicitly-constructed temporary of the container element type. In this case, the corresponding **emplace_back** method results in less verbose and potentially more efficient code. Right now the check doesn't support **push_front** and **insert**. It also doesn't support **insert** functions for associative containers because replacing **insert** with **emplace** may result in *speed regression*, but it might get support with some addition flag in the future.

By default only **std::vector**, **std::deque**, **std::list** are considered. This list can be modified using the *ContainersWithPushBack* option.

This check also reports when an **emplace**-like method is improperly used, for example using **emplace_back** while also calling a constructor. This creates a temporary that requires at best a move and at worst a copy. Almost all **emplace**-like functions in the STL are covered by this, with **try_emplace** on **std::map** and **std::unordered_map** being the exception as it behaves slightly differently than all the others. More containers can be added with the *EmplacyFunctions* option, so long as the container defines a **value_type** type, and the **emplace**-like functions construct a **value_type** object.

Before:

```
std::vector<MyClass> v;
v.push_back(MyClass(21, 37));
v.emplace_back(MyClass(21, 37));

std::vector<std::pair<int, int>> w;

w.push_back(std::pair<int, int>(21, 37));
w.push_back(std::make_pair(21L, 37L));
w.emplace_back(std::make_pair(21L, 37L));
```

After:

```
std::vector<MyClass> v;
v.emplace_back(21, 37);
v.emplace_back(21, 37);

std::vector<std::pair<int, int>> w;
w.emplace_back(21, 37);
w.emplace_back(21L, 37L);
w.emplace_back(21L, 37L);
```

By default, the check is able to remove unnecessary **std::make_pair** and **std::make_tuple** calls from **push_back** calls on containers of **std::pair** and **std::tuple**. Custom tuple-like types can be modified by the *TupleTypes* option; custom make functions can be modified by the *TupleMakeFunctions* option.

The other situation is when we pass arguments that will be converted to a type inside a container.

Before:

```
std::vector<boost::optional<std::string> > v;
v.push_back("abc");
```

After:

```
std::vector<boost::optional<std::string> > v;
v.emplace_back("abc");
```

In some cases the transformation would be valid, but the code wouldn't be exception safe. In this case the calls of **push_back** won't be replaced.

```
std::vector<std::unique_ptr<int>> v;
v.push_back(std::unique_ptr<int>(new int(0)));
auto *ptr = new int(1);
v.push_back(std::unique_ptr<int>(ptr));
```

This is because replacing it with **emplace_back** could cause a leak of this pointer if **emplace_back** would throw exception before emplacement (e.g. not enough memory to add a new element).

For more info read item 42 - "Consider emplacement instead of insertion." of Scott Meyers "Effective Modern C++".

The default smart pointers that are considered are **std::unique_ptr**, **std::shared_ptr**, **std::auto_ptr**. To specify other smart pointers or other classes use the *SmartPointers* option.

Check also doesn't fire if any argument of the constructor call would be:

- a bit-field (bit-fields can't bind to rvalue/universal reference)

- a **new** expression (to avoid leak)
- if the argument would be converted via derived-to-base cast.

This check requires C++11 or higher to run.

Options

ContainersWithPushBack
Semicolon-separated list of class names of custom containers that support **push_back**.

IgnoreImplicitConstructors
When *true*, the check will ignore implicitly constructed arguments of **push_back**, e.g.

```
std::vector<std::string> v;
v.push_back("a"); // Ignored when IgnoreImplicitConstructors is ‘true‘.
```

Default is *false*.

SmartPointers
Semicolon-separated list of class names of custom smart pointers.

TupleTypes
Semicolon-separated list of **std::tuple**-like class names.

TupleMakeFunctions
Semicolon-separated list of **std::make_tuple**-like function names. Those function calls will be removed from **push_back** calls and turned into **emplace_back**.

EmplacyFunctions
Semicolon-separated list of containers without their template parameters and some **emplace**-like method of the container. Example: **vector::emplace_back**. Those methods will be checked for improper use and the check will report when a temporary is unnecessarily created.

Example

```
std::vector<MyTuple<int, bool, char>> x;
x.push_back(MakeMyTuple(1, false, ’x’));
x.emplace_back(MakeMyTuple(1, false, ’x’));
```

transforms to:

```
std::vector<MyTuple<int, bool, char>> x;
x.emplace_back(1, false, 'x');
x.emplace_back(1, false, 'x');
```

when *TupleTypes* is set to **MyTuple**, *TupleMakeFunctions* is set to **MakeMyTuple**, and *EmplacyFunctions* is set to **vector::emplace_back**.

modernize-use-equals-default

This check replaces default bodies of special member functions with **= default;**. The explicitly defaulted function declarations enable more opportunities in optimization, because the compiler might treat explicitly defaulted functions as trivial.

```
struct A {
  A() {}
  ~A();
};
A::~A() {}

// becomes

struct A {
  A() = default;
  ~A();
};
A::~A() = default;
```

NOTE:

Move-constructor and move-assignment operator are not supported yet.

Options

IgnoreMacros

If set to *true*, the check will not give warnings inside macros. Default is *true*.

modernize-use-equals-delete

This check marks unimplemented private special member functions with **= delete**. To avoid false-positives, this check only applies in a translation unit that has all other member functions implemented.

```
struct A {
private:
  A(const A&);
  A& operator=(const A&);
};

// becomes

struct A {
private:
  A(const A&) = delete;
  A& operator=(const A&) = delete;
};
```

IgnoreMacros
If this option is set to *true* (default is *true*), the check will not warn about functions declared inside macros.

modernize-use-nodiscard
Adds **[[nodiscard]]** attributes (introduced in C++17) to member functions in order to highlight at compile time which return values should not be ignored.

Member functions need to satisfy the following conditions to be considered by this check:

- no **[[nodiscard]]**, **[[noreturn]]**, **__attribute__((warn_unused_result))**, **[[clang::warn_unused_result]]** nor **[[gcc::warn_unused_result]]** attribute,
- non-void return type,
- non-template return types,
- const member function,
- non-variadic functions,
- no non-const reference parameters,
- no pointer parameters,
- no template parameters,

- no template function parameters,
- not be a member of a class with mutable member variables,
- no Lambdas,
- no conversion functions.

Such functions have no means of altering any state or passing values other than via the return type. Unless the member functions are altering state via some external call (e.g. I/O).

Example

```
bool empty() const;
bool empty(int i) const;
```

transforms to:

```
[[nodiscard]] bool empty() const;
[[nodiscard]] bool empty(int i) const;
```

Options

ReplacementString
Specifies a macro to use instead of **[[nodiscard]]**. This is useful when maintaining source code that needs to compile with a pre-C++17 compiler.

Example

```
bool empty() const;
bool empty(int i) const;
```

transforms to:

```
NO_DISCARD bool empty() const;
NO_DISCARD bool empty(int i) const;
```

if the *ReplacementString* option is set to *NO_DISCARD*.

NOTE:

If the *ReplacementString* is not a C++ attribute, but instead a macro, then that macro must be defined in scope or the fix-it will not be applied.

NOTE:
For alternative **__attribute__** syntax options to mark functions as **[[nodiscard]]** in non-c++17 source code. See *https://clang.llvm.org/docs/AttributeReference.html#nodiscard-warn-unused-result*

modernize-use-noexcept

This check replaces deprecated dynamic exception specifications with the appropriate noexcept specification (introduced in C++11). By default this check will replace **throw()** with **noexcept**, and **throw(<exception>[,...])** or **throw(...)** with **noexcept(false)**.

Example

```
void foo() throw();
void bar() throw(int) {}
```

transforms to:

```
void foo() noexcept;
void bar() noexcept(false) {}
```

Options

ReplacementString

Users can use *ReplacementString* to specify a macro to use instead of **noexcept**. This is useful when maintaining source code that uses custom exception specification marking other than **noexcept**. Fix-it hints will only be generated for non-throwing specifications.

Example

```
void bar() throw(int);
void foo() throw();
```

transforms to:

```
void bar() throw(int);  // No fix-it generated.
void foo() NOEXCEPT;
```

if the *ReplacementString* option is set to *NOEXCEPT*.

UseNoexceptFalse

Enabled by default, disabling will generate fix-it hints that remove throwing dynamic exception specs, e.g., **throw(<something>)**, completely without providing a replacement text, except for destructors and delete operators that are **noexcept(true)** by default.

Example

```
void foo() throw(int) {}

struct bar {
  void foobar() throw(int);
  void operator delete(void *ptr) throw(int);
  void operator delete[](void *ptr) throw(int);
  ~bar() throw(int);
}
```

transforms to:

```
void foo() {}

struct bar {
  void foobar();
  void operator delete(void *ptr) noexcept(false);
  void operator delete[](void *ptr) noexcept(false);
  ~bar() noexcept(false);
}
```

if the *UseNoexceptFalse* option is set to *false*.

modernize-use-nullptr

The check converts the usage of null pointer constants (e.g. **NULL**, **0**) to use the new C++11 **nullptr** keyword.

Example

```
void assignment() {
  char *a = NULL;
```

```
  char *b = 0;
  char c = 0;
}

int *ret_ptr() {
  return 0;
}
```

transforms to:

```
void assignment() {
  char *a = nullptr;
  char *b = nullptr;
  char c = 0;
}

int *ret_ptr() {
  return nullptr;
}
```

Options

NullMacros

Comma-separated list of macro names that will be transformed along with **NULL**. By default this check will only replace the **NULL** macro and will skip any similar user-defined macros.

Example

```
#define MY_NULL (void*)0
void assignment() {
  void *p = MY_NULL;
}
```

transforms to:

```
#define MY_NULL NULL
void assignment() {
  int *p = nullptr;
}
```

if the *NullMacros* option is set to **MY_NULL**.

modernize-use-override

Adds **override** (introduced in C++11) to overridden virtual functions and removes **virtual** from those functions as it is not required.

virtual on non base class implementations was used to help indicate to the user that a function was virtual. C++ compilers did not use the presence of this to signify an overridden function.

In C++ 11 **override** and **final** keywords were introduced to allow overridden functions to be marked appropriately. Their presence allows compilers to verify that an overridden function correctly overrides a base class implementation.

This can be useful as compilers can generate a compile time error when:

- The base class implementation function signature changes.
- The user has not created the override with the correct signature.

Options

IgnoreDestructors

If set to *true*, this check will not diagnose destructors. Default is *false*.

AllowOverrideAndFinal

If set to *true*, this check will not diagnose **override** as redundant with **final**. This is useful when code will be compiled by a compiler with warning/error checking flags requiring **override** explicitly on overridden members, such as **gcc -Wsuggest-override**/**gcc -Werror=suggest-override**. Default is *false*.

OverrideSpelling

Specifies a macro to use instead of **override**. This is useful when maintaining source code that also needs to compile with a pre-C++11 compiler.

FinalSpelling

Specifies a macro to use instead of **final**. This is useful when maintaining source code that also needs to compile with a pre-C++11 compiler.

NOTE:

For more information on the use of **override** see

https://en.cppreference.com/w/cpp/language/override

modernize-use-trailing-return-type

Rewrites function signatures to use a trailing return type (introduced in C++11). This transformation is purely stylistic. The return type before the function name is replaced by **auto** and inserted after the function parameter list (and qualifiers).

Example

```
int f1();
inline int f2(int arg) noexcept;
virtual float f3() const && = delete;
```

transforms to:

```
auto f1() -> int;
inline auto f2(int arg) -> int noexcept;
virtual auto f3() const && -> float = delete;
```

Known Limitations

The following categories of return types cannot be rewritten currently:

- function pointers
- member function pointers
- member pointers

Unqualified names in the return type might erroneously refer to different entities after the rewrite. Preventing such errors requires a full lookup of all unqualified names present in the return type in the scope of the trailing return type location. This location includes e.g. function parameter names and members of the enclosing class (including all inherited classes). Such a lookup is currently not implemented.

Given the following piece of code

```
struct S { long long value; };
S f(unsigned S) { return {S * 2}; }
class CC {
  int S;
```

```
  struct S m();
};
S CC::m() { return {0}; }
```

a careless rewrite would produce the following output:

```
struct S { long long value; };
auto f(unsigned S) -> S { return {S * 2}; } // error
class CC {
  int S;
  auto m() -> struct S;
};
auto CC::m() -> S { return {0}; } // error
```

This code fails to compile because the S in the context of f refers to the equally named function parameter. Similarly, the S in the context of m refers to the equally named class member. The check can currently only detect and avoid a clash with a function parameter name.

modernize-use-transparent-functors

Prefer transparent functors to non-transparent ones. When using transparent functors, the type does not need to be repeated. The code is easier to read, maintain and less prone to errors. It is not possible to introduce unwanted conversions.

```
// Non-transparent functor
std::map<int, std::string, std::greater<int>> s;

// Transparent functor.
std::map<int, std::string, std::greater<>> s;

// Non-transparent functor
using MyFunctor = std::less<MyType>;
```

It is not always a safe transformation though. The following case will be untouched to preserve the semantics.

```
// Non-transparent functor
std::map<const char *, std::string, std::greater<std::string>> s;
```

Options

SafeMode

If the option is set to *true*, the check will not diagnose cases where using a transparent functor cannot be guaranteed to produce identical results as the original code. The default value for this option is *false*.

This check requires using C++14 or higher to run.

modernize-use-uncaught-exceptions

This check will warn on calls to **std::uncaught_exception** and replace them with calls to **std::uncaught_exceptions**, since **std::uncaught_exception** was deprecated in C++17.

Below are a few examples of what kind of occurrences will be found and what they will be replaced with.

```
#define MACRO1 std::uncaught_exception
#define MACRO2 std::uncaught_exception

int uncaught_exception() {
  return 0;
}

int main() {
  int res;

  res = uncaught_exception();
  // No warning, since it is not the deprecated function from namespace std

  res = MACRO2();
  // Warning, but will not be replaced

  res = std::uncaught_exception();
  // Warning and replaced

  using std::uncaught_exception;
  // Warning and replaced

  res = uncaught_exception();
  // Warning and replaced
}
```

After applying the fixes the code will look like the following:

```
#define MACRO1 std::uncaught_exception
#define MACRO2 std::uncaught_exception

int uncaught_exception() {
  return 0;
}

int main() {
  int res;

  res = uncaught_exception();

  res = MACRO2();

  res = std::uncaught_exceptions();

  using std::uncaught_exceptions;

  res = uncaught_exceptions();
}
```

modernize-use-using

The check converts the usage of **typedef** with **using** keyword.

Before:

```
typedef int variable;

class Class{};
typedef void (Class::* MyPtrType)() const;

typedef struct { int a; } R_t, *R_p;
```

After:

```
using variable = int;

class Class{};
```

```
using MyPtrType = void (Class::*)() const;

using R_t = struct { int a; };
using R_p = R_t*;
```

This check requires using C++11 or higher to run.

Options

IgnoreMacros
If set to *true*, the check will not give warnings inside macros. Default is *true*.

mpi-buffer-deref
This check verifies if a buffer passed to an MPI (Message Passing Interface) function is sufficiently dereferenced. Buffers should be passed as a single pointer or array. As MPI function signatures specify **void *** for their buffer types, insufficiently dereferenced buffers can be passed, like for example as double pointers or multidimensional arrays, without a compiler warning emitted.

Examples:

```
// A double pointer is passed to the MPI function.
char *buf;
MPI_Send(&buf, 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);

// A multidimensional array is passed to the MPI function.
short buf[1][1];
MPI_Send(buf, 1, MPI_SHORT, 0, 0, MPI_COMM_WORLD);

// A pointer to an array is passed to the MPI function.
short *buf[1];
MPI_Send(buf, 1, MPI_SHORT, 0, 0, MPI_COMM_WORLD);
```

mpi-type-mismatch
This check verifies if buffer type and MPI (Message Passing Interface) datatype pairs match for used MPI functions. All MPI datatypes defined by the MPI standard (3.1) are verified by this check. User defined typedefs, custom MPI datatypes and null pointer constants are skipped, in the course of verification.

Example:

```
// In this case, the buffer type matches MPI datatype.
char buf;
MPI_Send(&buf, 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);

// In the following case, the buffer type does not match MPI datatype.
int buf;
MPI_Send(&buf, 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
```

objc-assert-equals
Finds improper usages of *XCTAssertEqual* and *XCTAssertNotEqual* and replaces them with *XCTAssertEqualObjects* or *XCTAssertNotEqualObjects*.

This makes tests less fragile, as many improperly rely on pointer equality for strings that have equal values. This assumption is not guarantted by the language.

objc-avoid-nserror-init
Finds improper initialization of **NSError** objects.

According to Apple developer document, we should always use factory method **errorWithDomain:code:userInfo:** to create new NSError objects instead of **[NSError alloc] init]**. Otherwise it will lead to a warning message during runtime.

The corresponding information about **NSError** creation:
https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/ErrorHandlingCocoa/CreateCustomizel

objc-dealloc-in-category
Finds implementations of **-dealloc** in Objective-C categories. The category implementation will override any **-dealloc** in the class implementation, potentially causing issues.

Classes implement **-dealloc** to perform important actions to deallocate an object. If a category on the class implements **-dealloc**, it will override the class's implementation and unexpected deallocation behavior may occur.

objc-forbidden-subclassing
Finds Objective-C classes which are subclasses of classes which are not designed to be subclassed.

By default, includes a list of Objective-C classes which are publicly documented as not supporting subclassing.

NOTE:

Instead of using this check, for code under your control, you should add **__attribute__((objc_subclassing_restricted))** before your **@interface** declarations to ensure the compiler prevents others from subclassing your Objective-C classes. See *https://clang.llvm.org/docs/AttributeReference.html#objc-subclassing-restricted*

Options

ForbiddenSuperClassNames
Semicolon-separated list of names of Objective-C classes which do not support subclassing.

Defaults to *ABNewPersonViewController;ABPeoplePickerNavigationController;ABPersonViewController;ABUnknownPerso*

objc-missing-hash
Finds Objective-C implementations that implement **-isEqual:** without also appropriately implementing **-hash**.

Apple documentation highlights that objects that are equal must have the same hash value: *https://developer.apple.com/documentation/objectivec/1418956-nsobject/1418795-isequal?language=objc*

Note that the check only verifies the presence of **-hash** in scenarios where its omission could result in unexpected behavior. The verification of the implementation of **-hash** is the responsibility of the developer, e.g., through the addition of unit tests to verify the implementation.

objc-nsinvocation-argument-lifetime
Finds calls to **NSInvocation** methods under ARC that don't have proper argument object lifetimes. When passing Objective-C objects as parameters to the **NSInvocation** methods **getArgument:atIndex:** and **getReturnValue:**, the values are copied by value into the argument pointer, which leads to incorrect releasing behavior if the object pointers are not declared **__unsafe_unretained**.

For code:

```
id arg;
[invocation getArgument:&arg atIndex:2];

__strong id returnValue;
[invocation getReturnValue:&returnValue];
```

The fix will be:

```
__unsafe_unretained id arg;
[invocation getArgument:&arg atIndex:2];

__unsafe_unretained id returnValue;
[invocation getReturnValue:&returnValue];
```

The check will warn on being passed instance variable references that have lifetimes other than **__unsafe_unretained**, but does not propose a fix:

```
// "id _returnValue" is declaration of instance variable of class.
[invocation getReturnValue:&self->_returnValue];
```

objc-property-declaration

Finds property declarations in Objective-C files that do not follow the pattern of property names in Apple's programming guide. The property name should be in the format of Lower Camel Case.

For code:

```
@property(nonatomic, assign) int LowerCamelCase;
```

The fix will be:

```
@property(nonatomic, assign) int lowerCamelCase;
```

The check will only fix 'CamelCase' to 'camelCase'. In some other cases we will only provide warning messages since the property name could be complicated. Users will need to come up with a proper name by their own.

This check also accepts special acronyms as prefixes or suffixes. Such prefixes or suffixes will suppress the Lower Camel Case check according to the guide:
https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/CodingGuidelines/Articles/Nami

For a full list of well-known acronyms:
https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/CodingGuidelines/Articles/APIA

The corresponding style rule:
https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/CodingGuidelines/Articles/Nami

The check will also accept property declared in category with a prefix of lowercase letters followed by a '_' to avoid naming conflict. For example:

@property(nonatomic, assign) int abc_lowerCamelCase;

The corresponding style rule:
https://developer.apple.com/library/content/qa/qa1908/_index.html

objc-super-self
Finds invocations of **-self** on super instances in initializers of subclasses of **NSObject** and recommends calling a superclass initializer instead.

Invoking **-self** on super instances in initializers is a common programmer error when the programmer's original intent is to call a superclass initializer. Failing to call a superclass initializer breaks initializer chaining and can result in invalid object initialization.

openmp-exception-escape
Analyzes OpenMP Structured Blocks and checks that no exception escapes out of the Structured Block it was thrown in.

As per the OpenMP specification, a structured block is an executable statement, possibly compound, with a single entry at the top and a single exit at the bottom. Which means, **throw** may not be used to 'exit' out of the structured block. If an exception is not caught in the same structured block it was thrown in, the behavior is undefined.

FIXME: this check does not model SEH, **setjmp**/**longjmp**.

WARNING! This check may be expensive on large source files.

Options

IgnoredExceptions
Comma-separated list containing type names which are not counted as thrown exceptions in the check. Default value is an empty string.

openmp-use-default-none
Finds OpenMP directives that are allowed to contain a **default** clause, but either don't specify it or the clause is specified but with the kind other than **none**, and suggests to use the **default(none)** clause.

Using **default(none)** clause forces developers to explicitly specify data sharing attributes for the variables referenced in the construct, thus making it obvious which variables are referenced, and what is their data sharing attribute, thus increasing readability and possibly making errors easier to spot.

Example

```
// ‘‘for‘‘ directive cannot have ‘‘default‘‘ clause, no diagnostics.
void n0(const int a) {
#pragma omp for
  for (int b = 0; b < a; b++)
    ;
}

// ‘‘parallel‘‘ directive.

// ‘‘parallel‘‘ directive can have ‘‘default‘‘ clause, but said clause is not
// specified, diagnosed.
void p0_0() {
#pragma omp parallel
  ;
  // WARNING: OpenMP directive ‘‘parallel‘‘ does not specify ‘‘default‘‘
  //          clause. Consider specifying ‘‘default(none)‘‘ clause.
}

// ‘‘parallel‘‘ directive can have ‘‘default‘‘ clause, and said clause is
// specified, with ‘‘none‘‘ kind, all good.
void p0_1() {
#pragma omp parallel default(none)
  ;
}

// ‘‘parallel‘‘ directive can have ‘‘default‘‘ clause, and said clause is
// specified, but with ‘‘shared‘‘ kind, which is not ‘‘none‘‘, diagnose.
void p0_2() {
#pragma omp parallel default(shared)
  ;
  // WARNING: OpenMP directive ‘‘parallel‘‘ specifies ‘‘default(shared)‘‘
  //          clause. Consider using ‘‘default(none)‘‘ clause instead.
}

// ‘‘parallel‘‘ directive can have ‘‘default‘‘ clause, and said clause is
// specified, but with ‘‘firstprivate‘‘ kind, which is not ‘‘none‘‘, diagnose.
void p0_3() {
#pragma omp parallel default(firstprivate)
```

```
  ;
  // WARNING: OpenMP directive ‘‘parallel‘‘ specifies ‘‘default(firstprivate)‘‘
  //          clause. Consider using ‘‘default(none)‘‘ clause instead.
}
```

performance-faster-string-find

Optimize calls to **std::string::find()** and friends when the needle passed is a single character string literal. The character literal overload is more efficient.

Examples:

```
str.find("A");

// becomes

str.find('A');
```

Options

StringLikeClasses

Semicolon-separated list of names of string-like classes. By default only **::std::basic_string** and **::std::basic_string_view** are considered. The check will only consider member functions named **find**, **rfind**, **find_first_of**, **find_first_not_of**, **find_last_of**, or **find_last_not_of** within these classes.

performance-for-range-copy

Finds C++11 for ranges where the loop variable is copied in each iteration but it would suffice to obtain it by const reference.

The check is only applied to loop variables of types that are expensive to copy which means they are not trivially copyable or have a non-trivial copy constructor or destructor.

To ensure that it is safe to replace the copy with a const reference the following heuristic is employed:

1. The loop variable is const qualified.

2. The loop variable is not const, but only const methods or operators are invoked on it, or it is used as const reference or value argument in constructors or function calls.

Options

WarnOnAllAutoCopies
When *true*, warns on any use of *auto* as the type of the range-based for loop variable. Default is *false*.

AllowedTypes
A semicolon-separated list of names of types allowed to be copied in each iteration. Regular expressions are accepted, e.g. *[Rr]ef(erence)?$* matches every type with suffix *Ref, ref, Reference* and *reference*. The default is empty. If a name in the list contains the sequence *::* it is matched against the qualified typename (i.e. *namespace::Type*, otherwise it is matched against only the type name (i.e. *Type*).

performance-implicit-cast-in-loop
This check has been renamed to *performance-implicit-conversion-in-loop*.

performance-implicit-conversion-in-loop
This warning appears in a range-based loop with a loop variable of const ref type where the type of the variable does not match the one returned by the iterator. This means that an implicit conversion happens, which can for example result in expensive deep copies.

Example:

```
map<int, vector<string>> my_map;
for (const pair<int, vector<string>>& p : my_map) {}
// The iterator type is in fact pair<const int, vector<string>>, which means
// that the compiler added a conversion, resulting in a copy of the vectors.
```

The easiest solution is usually to use **const auto&** instead of writing the type manually.

performance-inefficient-algorithm
Warns on inefficient use of STL algorithms on associative containers.

Associative containers implement some of the algorithms as methods which should be preferred to the algorithms in the algorithm header. The methods can take advantage of the order of the elements.

```
std::set<int> s;
auto it = std::find(s.begin(), s.end(), 43);

// becomes

auto it = s.find(43);
```

```
std::set<int> s;
auto c = std::count(s.begin(), s.end(), 43);

// becomes

auto c = s.count(43);
```

performance-inefficient-string-concatenation

This check warns about the performance overhead arising from concatenating strings using the **operator+**, for instance:

```
std::string a("Foo"), b("Bar");
a = a + b;
```

Instead of this structure you should use **operator+=** or **std::string**'s (**std::basic_string**) class member function **append()**. For instance:

```
std::string a("Foo"), b("Baz");
for (int i = 0; i < 20000; ++i) {
  a = a + "Bar" + b;
}
```

Could be rewritten in a greatly more efficient way like:

```
std::string a("Foo"), b("Baz");
for (int i = 0; i < 20000; ++i) {
  a.append("Bar").append(b);
}
```

And this can be rewritten too:

```
void f(const std::string&) {}
std::string a("Foo"), b("Baz");
void g() {
  f(a + "Bar" + b);
}
```

In a slightly more efficient way like:

```
void f(const std::string&) {}
```

```
std::string a("Foo"), b("Baz");
void g() {
   f(std::string(a).append("Bar").append(b));
}
```

Options

StrictMode
When *false*, the check will only check the string usage in **while**, **for** and **for-range** statements. Default is *false*.

performance-inefficient-vector-operation
Finds possible inefficient **std::vector** operations (e.g. **push_back**, **emplace_back**) that may cause unnecessary memory reallocations.

It can also find calls that add element to protobuf repeated field in a loop without calling Reserve() before the loop. Calling Reserve() first can avoid unnecessary memory reallocations.

Currently, the check only detects following kinds of loops with a single statement body:

- Counter-based for loops start with 0:

```
std::vector<int> v;
for (int i = 0; i < n; ++i) {
  v.push_back(n);
  // This will trigger the warning since the push_back may cause multiple
  // memory reallocations in v. This can be avoid by inserting a 'reserve(n)'
  // statement before the for statement.
}

SomeProto p;
for (int i = 0; i < n; ++i) {
  p.add_xxx(n);
  // This will trigger the warning since the add_xxx may cause multiple memory
  // reallocations. This can be avoid by inserting a
  // 'p.mutable_xxx().Reserve(n)' statement before the for statement.
}
```

- For-range loops like **for (range-declaration : range_expression)**, the type of **range_expression** can be **std::vector**, **std::array**, **std::deque**, **std::set**, **std::unordered_set**, **std::map**, **std::unordered_set**:

```
std::vector<int> data;
std::vector<int> v;

for (auto element : data) {
  v.push_back(element);
  // This will trigger the warning since the 'push_back' may cause multiple
  // memory reallocations in v. This can be avoid by inserting a
  // 'reserve(data.size())' statement before the for statement.
}
```

Options

VectorLikeClasses
Semicolon-separated list of names of vector-like classes. By default only **::std::vector** is considered.

EnableProto
When *true*, the check will also warn on inefficient operations for proto repeated fields. Otherwise, the check only warns on inefficient vector operations. Default is *false*.

performance-move-const-arg
The check warns

- if **std::move()** is called with a constant argument,
- if **std::move()** is called with an argument of a trivially-copyable type,
- if the result of **std::move()** is passed as a const reference argument.

In all three cases, the check will suggest a fix that removes the **std::move()**.

Here are examples of each of the three cases:

```
const string s;
return std::move(s);  // Warning: std::move of the const variable has no effect

int x;
return std::move(x);  // Warning: std::move of the variable of a trivially-copyable type has no effect

void f(const string &s);
```

```
string s;
f(std::move(s));  // Warning: passing result of std::move as a const reference argument; no move will actually happ
```

Options

CheckTriviallyCopyableMove
If *true*, enables detection of trivially copyable types that do not have a move constructor. Default is *true*.

CheckMoveToConstRef
If *true*, enables detection of *std::move()* passed as a const reference argument. Default is *true*.

performance-move-constructor-init
"cert-oop11-cpp" redirects here as an alias for this check.

The check flags user-defined move constructors that have a ctor-initializer initializing a member or base class through a copy constructor instead of a move constructor.

performance-no-automatic-move
Finds local variables that cannot be automatically moved due to constness.

Under *certain conditions*, local values are automatically moved out when returning from a function. A common mistake is to declare local **lvalue** variables **const**, which prevents the move.

Example *[1]*:

```
StatusOr<std::vector<int>> Cool() {
 std::vector<int> obj = ...;
 return obj;  // calls StatusOr::StatusOr(std::vector<int>&&)
}

StatusOr<std::vector<int>> NotCool() {
 const std::vector<int> obj = ...;
 return obj;  // calls ‘StatusOr::StatusOr(const std::vector<int>&)‘
}
```

The former version (**Cool**) should be preferred over the latter (**Uncool**) as it will avoid allocations and potentially large memory copies.

Semantics

In the example above, **StatusOr::StatusOr(T&&)** have the same semantics as long as the copy and move constructors for **T** have the same semantics. Note that there is no guarantee that **S::S(T&&)** and **S::S(const T&)** have the same semantics for any single **S**, so we're not providing automated fixes for this check, and judgement should be exerted when making the suggested changes.

-Wreturn-std-move

Another case where the move cannot happen is the following:

```
StatusOr<std::vector<int>> Uncool() {
 std::vector<int>&& obj = ...;
 return obj;  // calls ‘StatusOr::StatusOr(const std::vector<int>&)‘
}
```

In that case the fix is more consensual: just *return std::move(obj)*. This is handled by the *-Wreturn-std-move* warning.

performance-no-int-to-ptr

Diagnoses every integer to pointer cast.

While casting an (integral) pointer to an integer is obvious - you just get the integral value of the pointer, casting an integer to an (integral) pointer is deceivingly different. While you will get a pointer with that integral value, if you got that integral value via a pointer-to-integer cast originally, the new pointer will lack the provenance information from the original pointer.

So while (integral) pointer to integer casts are effectively no-ops, and are transparent to the optimizer, integer to (integral) pointer casts are *NOT* transparent, and may conceal information from optimizer.

While that may be the intention, it is not always so. For example, let's take a look at a routine to align the pointer up to the multiple of 16: The obvious, naive implementation for that is:

```
char* src(char* maybe_underbiased_ptr) {
 uintptr_t maybe_underbiased_intptr = (uintptr_t)maybe_underbiased_ptr;
 uintptr_t aligned_biased_intptr = maybe_underbiased_intptr + 15;
 uintptr_t aligned_intptr = aligned_biased_intptr & (~15);
 return (char*)aligned_intptr; // warning: avoid integer to pointer casts [performance-no-int-to-ptr]
}
```

The check will rightfully diagnose that cast.

But when provenance concealment is not the goal of the code, but an accident, this example can

be rewritten as follows, without using integer to pointer cast:

```
char*
tgt(char* maybe_underbiased_ptr) {
  uintptr_t maybe_underbiased_intptr = (uintptr_t)maybe_underbiased_ptr;
  uintptr_t aligned_biased_intptr = maybe_underbiased_intptr + 15;
  uintptr_t aligned_intptr = aligned_biased_intptr & (~15);
  uintptr_t bias = aligned_intptr - maybe_underbiased_intptr;
  return maybe_underbiased_ptr + bias;
}
```

performance-noexcept-move-constructor

The check flags user-defined move constructors and assignment operators not marked with **noexcept** or marked with **noexcept(expr)** where **expr** evaluates to **false** (but is not a **false** literal itself).

Move constructors of all the types used with STL containers, for example, need to be declared **noexcept**. Otherwise STL will choose copy constructors instead. The same is valid for move assignment operations.

performance-trivially-destructible

Finds types that could be made trivially-destructible by removing out-of-line defaulted destructor declarations.

```
struct A: TrivialType {
 ~A(); // Makes A non-trivially-destructible.
 TrivialType trivial_fields;
};
A::~A() = default;
```

performance-type-promotion-in-math-fn

Finds calls to C math library functions (from **math.h** or, in C++, **cmath**) with implicit **float** to **double** promotions.

For example, warns on **::sin(0.f)**, because this function's parameter is a double. You probably meant to call **std::sin(0.f)** (in C++), or **sinf(0.f)** (in C).

```
float a;
asin(a);

// becomes
```

```
float a;
std::asin(a);
```

performance-unnecessary-copy-initialization

Finds local variable declarations that are initialized using the copy constructor of a non-trivially-copyable type but it would suffice to obtain a const reference.

The check is only applied if it is safe to replace the copy by a const reference. This is the case when the variable is const qualified or when it is only used as a const, i.e. only const methods or operators are invoked on it, or it is used as const reference or value argument in constructors or function calls.

Example:

```
const string& constReference();
void Function() {
  // The warning will suggest making this a const reference.
  const string UnnecessaryCopy = constReference();
}

struct Foo {
  const string& name() const;
};
void Function(const Foo& foo) {
  // The warning will suggest making this a const reference.
  string UnnecessaryCopy1 = foo.name();
  UnnecessaryCopy1.find("bar");

  // The warning will suggest making this a const reference.
  string UnnecessaryCopy2 = UnnecessaryCopy1;
  UnnecessaryCopy2.find("bar");
}
```

Options

AllowedTypes

A semicolon-separated list of names of types allowed to be initialized by copying. Regular expressions are accepted, e.g. *[Rr]ef(erence)?$* matches every type with suffix *Ref, ref, Reference* and *reference*. The default is empty. If a name in the list contains the sequence *::* it is matched against the qualified typename (i.e. *namespace::Type*, otherwise it is matched against only the type name (i.e. *Type*).

ExcludedContainerTypes

A semicolon-separated list of names of types whose methods are allowed to return the const reference the variable is copied from. When an expensive to copy variable is copy initialized by the return value from a type on this list the check does not trigger. This can be used to exclude types known to be const incorrect or where the lifetime or immutability of returned references is not tied to mutations of the container. An example are view types that don't own the underlying data. Like for *AllowedTypes* above, regular expressions are accepted and the inclusion of *::* determines whether the qualified typename is matched or not.

performance-unnecessary-value-param

Flags value parameter declarations of expensive to copy types that are copied for each invocation but it would suffice to pass them by const reference.

The check is only applied to parameters of types that are expensive to copy which means they are not trivially copyable or have a non-trivial copy constructor or destructor.

To ensure that it is safe to replace the value parameter with a const reference the following heuristic is employed:

1. the parameter is const qualified;

2. the parameter is not const, but only const methods or operators are invoked on it, or it is used as const reference or value argument in constructors or function calls.

   Example:

```
void f(const string Value) {
  // The warning will suggest making Value a reference.
}

void g(ExpensiveToCopy Value) {
  // The warning will suggest making Value a const reference.
  Value.ConstMethd();
  ExpensiveToCopy Copy(Value);
}
```

   If the parameter is not const, only copied or assigned once and has a non-trivial move-constructor or move-assignment operator respectively the check will suggest to move it.

   Example:

```
void setValue(string Value) {
  Field = Value;
}
```

Will become:

```
#include <utility>

void setValue(string Value) {
  Field = std::move(Value);
}
```

Options

IncludeStyle
A string specifying which include-style is used, *llvm* or *google*. Default is *llvm*.

AllowedTypes
A semicolon-separated list of names of types allowed to be passed by value. Regular expressions are accepted, e.g. *[Rr]ef(erence)?$* matches every type with suffix *Ref*, *ref*, *Reference* and *reference*. The default is empty. If a name in the list contains the sequence *::* it is matched against the qualified typename (i.e. *namespace::Type*, otherwise it is matched against only the type name (i.e. *Type*).

portability-restrict-system-includes
Checks to selectively allow or disallow a configurable list of system headers.

For example:

In order to **only** allow *zlib.h* from the system you would set the options to *-*,zlib.h*.

```
#include <curses.h>       // Bad: disallowed system header.
#include <openssl/ssl.h>  // Bad: disallowed system header.
#include <zlib.h>         // Good: allowed system header.
#include "src/myfile.h"   // Good: non-system header always allowed.
```

In order to allow everything **except** *zlib.h* from the system you would set the options to **,-zlib.h*.

```
#include <curses.h>       // Good: allowed system header.
#include <openssl/ssl.h>  // Good: allowed system header.
```

```
#include <zlib.h>         // Bad: disallowed system header.
#include "src/myfile.h"   // Good: non-system header always allowed.
```

Since the options support globbing you can use wildcarding to allow groups of headers.

-,openssl/*.h* will allow all openssl headers but disallow any others.

```
#include <curses.h>       // Bad: disallowed system header.
#include <openssl/ssl.h>  // Good: allowed system header.
#include <openssl/rsa.h>  // Good: allowed system header.
#include <zlib.h>         // Bad: disallowed system header.
#include "src/myfile.h"   // Good: non-system header always allowed.
```

Options

Includes

A string containing a comma separated glob list of allowed include filenames. Similar to the -checks glob list for running clang-tidy itself, the two wildcard characters are * and -, to include and exclude globs, respectively. The default is *, which allows all includes.

portability-simd-intrinsics

Finds SIMD intrinsics calls and suggests **std::experimental::simd** (*P0214*) alternatives.

If the option *Suggest* is set to *true*, for

```
_mm_add_epi32(a, b); // x86
vec_add(a, b);       // Power
```

the check suggests an alternative: **operator+** on **std::experimental::simd** objects.

Otherwise, it just complains the intrinsics are non-portable (and there are *P0214* alternatives).

Many architectures provide SIMD operations (e.g. x86 SSE/AVX, Power AltiVec/VSX, ARM NEON). It is common that SIMD code implementing the same algorithm, is written in multiple target-dispatching pieces to optimize for different architectures or micro-architectures.

The C++ standard proposal *P0214* and its extensions cover many common SIMD operations. By migrating from target-dependent intrinsics to *P0214* operations, the SIMD code can be simplified and pieces for different targets can be unified.

Refer to *P0214* for introduction and motivation for the data-parallel standard library.

Options

Suggest
If this option is set to *true* (default is *false*), the check will suggest *P0214* alternatives, otherwise it only points out the intrinsic function is non-portable.

Std The namespace used to suggest *P0214* alternatives. If not specified, *std::* for *-std=c++20* and *std::experimental::* for *-std=c++11*.

portability-std-allocator-const
Report use of **std::vector<const T>** (and similar containers of const elements). These are not allowed in standard C++, and should usually be **std::vector<T>** instead."

Per C++ **[allocator.requirements.general]**: "T is any cv-unqualified object type", **std::allocator<const T>** is undefined. Many standard containers use **std::allocator** by default and therefore their **const T** instantiations are undefined.

libc++ defines **std::allocator<const T>** as an extension which will be removed in the future.

libstdc++ and MSVC do not support **std::allocator<const T>**:

```
// libstdc++ has a better diagnostic since https://gcc.gnu.org/bugzilla/show_bug.cgi?id=48101
std::deque<const int> deque; // error: static assertion failed: std::deque must have a non-const, non-volatile value_t
std::set<const int> set; // error: static assertion failed: std::set must have a non-const, non-volatile value_type
std::vector<int* const> vector; // error: static assertion failed: std::vector must have a non-const, non-volatile value

// MSVC
// error C2338: static_assert failed: 'The C++ Standard forbids containers of const elements because allocator<cons
```

Code bases only compiled with libc++ may accrue such undefined usage. This check finds such code and prevents backsliding while clean-up is ongoing.

readability-avoid-const-params-in-decls
Checks whether a function declaration has parameters that are top level **const**.

const values in declarations do not affect the signature of a function, so they should not be put there.

Examples:

```
void f(const string);   // Bad: const is top level.
void f(const string&);  // Good: const is not top level.
```

readability-braces-around-statements

google-readability-braces-around-statements redirects here as an alias for this check.

Checks that bodies of **if** statements and loops (**for**, **do while**, and **while**) are inside braces.

Before:

```
if (condition)
  statement;
```

After:

```
if (condition) {
  statement;
}
```

Options

ShortStatementLines

Defines the minimal number of lines that the statement should have in order to trigger this check.

The number of lines is counted from the end of condition or initial keyword (**do**/**else**) until the last line of the inner statement. Default value *0* means that braces will be added to all statements (not having them already).

readability-const-return-type

Checks for functions with a **const**-qualified return type and recommends removal of the **const** keyword. Such use of *const* is usually superfluous, and can prevent valuable compiler optimizations. Does not (yet) fix trailing return types.

Examples:

```
const int foo();
const Clazz foo();
Clazz *const foo();
```

Note that this applies strictly to top-level qualification, which excludes pointers or references to

const values. For example, these are fine:

```
const int* foo();
const int& foo();
const Clazz* foo();
```

readability-container-contains

Finds usages of **container.count()** and **container.find() == container.end()** which should be replaced by a call to the **container.contains()** method introduced in C++ 20.

Whether an element is contained inside a container should be checked with **contains** instead of **count**/**find** because **contains** conveys the intent more clearly. Furthermore, for containers which permit multiple entries per key (**multimap**, **multiset**, ...), **contains** is more efficient than **count** because **count** has to do unnecessary additional work.

Examples:

Initial expression	Result
myMap.find(x) == myMap.end()	**!myMap.contains(x)**
myMap.find(x) != myMap.end()	**myMap.contains(x)**
if (myMap.count(x))	**if (myMap.contains(x))**
bool exists = myMap.count(x)	**bool exists = myMap.contains(x)**
bool exists = myMap.count(x) > 0	**bool exists = myMap.contains(x)**
bool exists = myMap.count(x) >= 1	**bool exists = myMap.contains(x)**
bool missing = myMap.count(x)	**bool missing =**

== 0	**!myMap.contains(x)**

This check applies to **std::set**, **std::unordered_set**, **std::map**, **std::unordered_map** and the corresponding multi-key variants. It is only active for C++20 and later, as the **contains** method was only added in C++20.

readability-container-data-pointer

Finds cases where code could use **data()** rather than the address of the element at index 0 in a container. This pattern is commonly used to materialize a pointer to the backing data of a container. **std::vector** and **std::string** provide a **data()** accessor to retrieve the data pointer which should be preferred.

This also ensures that in the case that the container is empty, the data pointer access does not perform an errant memory access.

readability-container-size-empty

Checks whether a call to the **size()** method can be replaced with a call to **empty()**.

The emptiness of a container should be checked using the **empty()** method instead of the **size()** method. It is not guaranteed that **size()** is a constant-time function, and it is generally more efficient and also shows clearer intent to use **empty()**. Furthermore some containers may implement the **empty()** method but not implement the **size()** method. Using **empty()** whenever possible makes it easier to switch to another container in the future.

The check issues warning if a container has **size()** and **empty()** methods matching following signatures:

```
size_type size() const;
bool empty() const;
```

size_type can be any kind of integer type.

readability-convert-member-functions-to-static

Finds non-static member functions that can be made **static** because the functions don’t use **this**.

After applying modifications as suggested by the check, running the check again might find more opportunities to mark member functions **static**.

After making a member function **static**, you might want to run the check *readability-static-accessed-through-instance* to replace calls like **Instance.method()** by **Class::method()**.

readability-delete-null-pointer

Checks the **if** statements where a pointer's existence is checked and then deletes the pointer. The check is unnecessary as deleting a null pointer has no effect.

```
int *p;
if (p)
  delete p;
```

readability-duplicate-include

Looks for duplicate includes and removes them. The check maintains a list of included files and looks for duplicates. If a macro is defined or undefined then the list of included files is cleared.

Examples:

```
#include <memory>
#include <vector>
#include <memory>
```

becomes

```
#include <memory>
#include <vector>
```

Because of the intervening macro definitions, this code remains unchanged:

```
#undef NDEBUG
#include "assertion.h"
// ...code with assertions enabled

#define NDEBUG
#include "assertion.h"
// ...code with assertions disabled
```

readability-else-after-return

LLVM Coding Standards advises to reduce indentation where possible and where it makes understanding code easier. Early exit is one of the suggested enforcements of that. Please do not use **else** or **else if** after something that interrupts control flow - like **return**, **break**, **continue**, **throw**.

The following piece of code illustrates how the check works. This piece of code:

```
void foo(int Value) {
  int Local = 0;
  for (int i = 0; i < 42; i++) {
    if (Value == 1) {
      return;
    } else {
      Local++;
    }

    if (Value == 2)
      continue;
    else
      Local++;

    if (Value == 3) {
      throw 42;
    } else {
      Local++;
    }
  }
}
```

Would be transformed into:

```
void foo(int Value) {
  int Local = 0;
  for (int i = 0; i < 42; i++) {
    if (Value == 1) {
      return;
    }
    Local++;

    if (Value == 2)
      continue;
    Local++;

    if (Value == 3) {
      throw 42;
    }
    Local++;
```

```
  }
}
```

Options

WarnOnUnfixable
When *true*, emit a warning for cases where the check can't output a Fix-It. These can occur with declarations inside the **else** branch that would have an extended lifetime if the **else** branch was removed. Default value is *true*.

WarnOnConditionVariables
When *true*, the check will attempt to refactor a variable defined inside the condition of the **if** statement that is used in the **else** branch defining them just before the **if** statement. This can only be done if the **if** statement is the last statement in its parent's scope. Default value is *true*.

LLVM alias
There is an alias of this check called llvm-else-after-return. In that version the options *WarnOnUnfixable* and *WarnOnConditionVariables* are both set to *false* by default.

This check helps to enforce this *LLVM Coding Standards recommendation*.

readability-function-cognitive-complexity
Checks function Cognitive Complexity metric.

The metric is implemented as per the *COGNITIVE COMPLEXITY by SonarSource* specification version 1.2 (19 April 2017).

Options

Threshold
Flag functions with Cognitive Complexity exceeding this number. The default is *25*.

DescribeBasicIncrements
If set to *true*, then for each function exceeding the complexity threshold the check will issue additional diagnostics on every piece of code (loop, *if* statement, etc.) which contributes to that complexity. See also the examples below. Default is *true*.

IgnoreMacros
If set to *true*, the check will ignore code inside macros. Note, that also any macro arguments are ignored, even if they should count to the complexity. As this might change in the future, this

option isn’t guaranteed to be forward-compatible. Default is *false*.

Building blocks

There are three basic building blocks of a Cognitive Complexity metric:

Increment

The following structures increase the function’s Cognitive Complexity metric (by *1*):

- Conditional operators:
  - **if()**
  - **else if()**
  - **else**
  - **cond ? true : false**
- **switch()**
- Loops:
  - **for()**
  - C++11 range-based **for()**
  - **while()**
  - **do while()**
- **catch ()**
- **goto LABEL**, **goto *(&&LABEL))**,
- sequences of binary logical operators:
  - **boolean1 || boolean2**
  - **boolean1 && boolean2**

Nesting level

While by itself the nesting level does not change the function's Cognitive Complexity metric, it is tracked, and is used by the next, third building block. The following structures increase the nesting level (by *1*):

- Conditional operators:
  - **if()**
  - **else if()**
  - **else**
  - **cond ? true : false**
- **switch()**
- Loops:
  - **for()**
  - C++11 range-based **for()**
  - **while()**
  - **do while()**
- **catch ()**
- Nested functions:
  - C++11 Lambda
  - Nested **class**
  - Nested **struct**
- GNU statement expression
- Apple Block Declaration

Nesting increment

This is where the previous basic building block, *Nesting level*, matters. The following structures increase the function's Cognitive Complexity metric by the current *Nesting level*:

- Conditional operators:
  - **if()**
  - **cond ? true : false**
- **switch()**
- Loops:
  - **for()**
  - C++11 range-based **for()**
  - **while()**
  - **do while()**
- **catch ()**

Examples

The simplest case. This function has Cognitive Complexity of *0*.

```
void function0() {}
```

Slightly better example. This function has Cognitive Complexity of *1*.

```
int function1(bool var) {
  if(var) // +1, nesting level +1
    return 42;
  return 0;
}
```

Full example. This function has Cognitive Complexity of *3*.

```
int function3(bool var1, bool var2) {
```

```
  if(var1) { // +1, nesting level +1
    if(var2)  // +2 (1 + current nesting level of 1), nesting level +1
      return 42;
  }

  return 0;
}
```

In the last example, the check will flag *function3* if the option Threshold is set to *2* or smaller. If the option DescribeBasicIncrements is set to *true*, it will additionally flag the two *if* statements with the amounts by which they increase to the complexity of the function and the current nesting level.

Limitations

The metric is implemented with two notable exceptions:

- *preprocessor conditionals* (**#ifdef**, **#if**, **#elif**, **#else**, **#endif**) are not accounted for.

- *each method in a recursion cycle* is not accounted for. It can't be fully implemented, because cross-translational-unit analysis would be needed, which is currently not possible in clang-tidy.

readability-function-size

google-readability-function-size redirects here as an alias for this check.

Checks for large functions based on various metrics.

Options

LineThreshold

Flag functions exceeding this number of lines. The default is *-1* (ignore the number of lines).

StatementThreshold

Flag functions exceeding this number of statements. This may differ significantly from the number of lines for macro-heavy code. The default is *800*.

BranchThreshold

Flag functions exceeding this number of control statements. The default is *-1* (ignore the number of branches).

ParameterThreshold
Flag functions that exceed a specified number of parameters. The default is *-1* (ignore the number of parameters).

NestingThreshold
Flag compound statements which create next nesting level after *NestingThreshold*. This may differ significantly from the expected value for macro-heavy code. The default is *-1* (ignore the nesting level).

VariableThreshold
Flag functions exceeding this number of variables declared in the body. The default is *-1* (ignore the number of variables). Please note that function parameters and variables declared in lambdas, GNU Statement Expressions, and nested class inline functions are not counted.

readability-identifier-length
This check finds variables and function parameters whose length are too short. The desired name length is configurable.

Special cases are supported for loop counters and for exception variable names.

Options
The following options are described below:

- *MinimumVariableNameLength, IgnoredVariableNames*
- *MinimumParameterNameLength, IgnoredParameterNames*
- *MinimumLoopCounterNameLength, IgnoredLoopCounterNames*
- *MinimumExceptionNameLength, IgnoredExceptionVariableNames*

MinimumVariableNameLength
All variables (other than loop counter, exception names and function parameters) are expected to have at least a length of *MinimumVariableNameLength* (default is *3*). Setting it to *0* or *1* disables the check entirely.

```
int doubler(int x)   // warns that x is too short
{
  return 2 * x;
}
```

This check does not have any fix suggestions in the general case since variable names have semantic value.

IgnoredVariableNames

Specifies a regular expression for variable names that are to be ignored. The default value is empty, thus no names are ignored.

MinimumParameterNameLength

All function parameter names are expected to have a length of at least *MinimumParameterNameLength* (default is *3*). Setting it to *0* or *1* disables the check entirely.

```
int i = 42;    // warns that 'i' is too short
```

This check does not have any fix suggestions in the general case since variable names have semantic value.

IgnoredParameterNames

Specifies a regular expression for parameters that are to be ignored. The default value is *^[n]$* for historical reasons.

MinimumLoopCounterNameLength

Loop counter variables are expected to have a length of at least *MinimumLoopCounterNameLength* characters (default is *2*). Setting it to *0* or *1* disables the check entirely.

```
// This warns that 'q' is too short.
for (int q = 0; q < size; ++ q) {
  // ...
}
```

IgnoredLoopCounterNames

Specifies a regular expression for counter names that are to be ignored. The default value is *^[ijk_]$*; the first three symbols for historical reasons and the last one since it is frequently used as a "don't care" value, specifically in tools such as Google Benchmark.

```
// This does not warn by default, for historical reasons.
for (int i = 0; i < size; ++ i) {
  // ...
}
```

MinimumExceptionNameLength

Exception clause variables are expected to have a length of at least *MinimumExceptionNameLength* (default is *2*). Setting it to *0* or *1* disables the check entirely.

```
try {
  // ...
}
// This warns that 'e' is too short.
catch (const std::exception& x) {
  // ...
}
```

IgnoredExceptionVariableNames

Specifies a regular expression for exception variable names that are to be ignored. The default value is *^[e]$* mainly for historical reasons.

```
try {
  // ...
}
// This does not warn by default, for historical reasons.
catch (const std::exception& e) {
  // ...
}
```

readability-identifier-naming

Checks for identifiers naming style mismatch.

This check will try to enforce coding guidelines on the identifiers naming. It supports one of the following casing types and tries to convert from one to another if a mismatch is detected

Casing types include:

- **lower_case**,
- **UPPER_CASE**,
- **camelBack**,
- **CamelCase**,

- **camel_Snake_Back**,
- **Camel_Snake_Case**,
- **aNy_CasE**.

  It also supports a fixed prefix and suffix that will be prepended or appended to the identifiers, regardless of the casing.

  Many configuration options are available, in order to be able to create different rules for different kinds of identifiers. In general, the rules are falling back to a more generic rule if the specific case is not configured.

  The naming of virtual methods is reported where they occur in the base class, but not where they are overridden, as it can't be fixed locally there. This also applies for pseudo-override patterns like CRTP.

Options

The following options are described below:

- *AbstractClassCase*, *AbstractClassPrefix*, *AbstractClassSuffix*, *AbstractClassIgnoredRegexp*, *AbstractClassHungarianPrefix*
- *AggressiveDependentMemberLookup*
- *ClassCase*, *ClassPrefix*, *ClassSuffix*, *ClassIgnoredRegexp*, *ClassHungarianPrefix*
- *ClassConstantCase*, *ClassConstantPrefix*, *ClassConstantSuffix*, *ClassConstantIgnoredRegexp*, *ClassConstantHungarianPrefix*
- *ClassMemberCase*, *ClassMemberPrefix*, *ClassMemberSuffix*, *ClassMemberIgnoredRegexp*, *ClassMemberHungarianPrefix*
- *ClassMethodCase*, *ClassMethodPrefix*, *ClassMethodSuffix*, *ClassMethodIgnoredRegexp*
- *ConstantCase*, *ConstantPrefix*, *ConstantSuffix*, *ConstantIgnoredRegexp*, *ConstantHungarianPrefix*
- *ConstantMemberCase*, *ConstantMemberPrefix*, *ConstantMemberSuffix*, *ConstantMemberIgnoredRegexp*, *ConstantMemberHungarianPrefix*

- *ConstantParameterCase*, *ConstantParameterPrefix*, *ConstantParameterSuffix*, *ConstantParameterIgnoredRegexp*, *ConstantParameterHungarianPrefix*

- *ConstantPointerParameterCase*, *ConstantPointerParameterPrefix*, *ConstantPointerParameterSuffix*, *ConstantPointerParameterIgnoredRegexp*, *ConstantPointerParameterHungarianPrefix*

- *ConstexprFunctionCase*, *ConstexprFunctionPrefix*, *ConstexprFunctionSuffix*, *ConstexprFunctionIgnoredRegexp*

- *ConstexprMethodCase*, *ConstexprMethodPrefix*, *ConstexprMethodSuffix*, *ConstexprMethodIgnoredRegexp*

- *ConstexprVariableCase*, *ConstexprVariablePrefix*, *ConstexprVariableSuffix*, *ConstexprVariableIgnoredRegexp*, *ConstexprVariableHungarianPrefix*

- *EnumCase*, *EnumPrefix*, *EnumSuffix*, *EnumIgnoredRegexp*

- *EnumConstantCase*, *EnumConstantPrefix*, *EnumConstantSuffix*, *EnumConstantIgnoredRegexp*, *EnumConstantHungarianPrefix*

- *FunctionCase*, *FunctionPrefix*, *FunctionSuffix*, *FunctionIgnoredRegexp*

- *GetConfigPerFile*

- *GlobalConstantCase*, *GlobalConstantPrefix*, *GlobalConstantSuffix*, *GlobalConstantIgnoredRegexp*, *GlobalConstantHungarianPrefix*

- *GlobalConstantPointerCase*, *GlobalConstantPointerPrefix*, *GlobalConstantPointerSuffix*, *GlobalConstantPointerIgnoredRegexp*, *GlobalConstantPointerHungarianPrefix*

- *GlobalFunctionCase*, *GlobalFunctionPrefix*, *GlobalFunctionSuffix*, *GlobalFunctionIgnoredRegexp*

- *GlobalPointerCase*, *GlobalPointerPrefix*, *GlobalPointerSuffix*, *GlobalPointerIgnoredRegexp*, *GlobalPointerHungarianPrefix*

- *GlobalVariableCase*, *GlobalVariablePrefix*, *GlobalVariableSuffix*, *GlobalVariableIgnoredRegexp*, *GlobalVariableHungarianPrefix*

- *IgnoreMainLikeFunctions*
- *InlineNamespaceCase, InlineNamespacePrefix, InlineNamespaceSuffix, InlineNamespaceIgnoredRegexp*
- *LocalConstantCase, LocalConstantPrefix, LocalConstantSuffix, LocalConstantIgnoredRegexp, LocalConstantHungarianPrefix*
- *LocalConstantPointerCase, LocalConstantPointerPrefix, LocalConstantPointerSuffix, LocalConstantPointerIgnoredRegexp, LocalConstantPointerHungarianPrefix*
- *LocalPointerCase, LocalPointerPrefix, LocalPointerSuffix, LocalPointerIgnoredRegexp, LocalPointerHungarianPrefix*
- *LocalVariableCase, LocalVariablePrefix, LocalVariableSuffix, LocalVariableIgnoredRegexp, LocalVariableHungarianPrefix*
- *MacroDefinitionCase, MacroDefinitionPrefix, MacroDefinitionSuffix, MacroDefinitionIgnoredRegexp*
- *MemberCase, MemberPrefix, MemberSuffix, MemberIgnoredRegexp, MemberHungarianPrefix*
- *MethodCase, MethodPrefix, MethodSuffix, MethodIgnoredRegexp*
- *NamespaceCase, NamespacePrefix, NamespaceSuffix, NamespaceIgnoredRegexp*
- *ParameterCase, ParameterPrefix, ParameterSuffix, ParameterIgnoredRegexp, ParameterHungarianPrefix*
- *ParameterPackCase, ParameterPackPrefix, ParameterPackSuffix, ParameterPackIgnoredRegexp*
- *PointerParameterCase, PointerParameterPrefix, PointerParameterSuffix, PointerParameterIgnoredRegexp, PointerParameterHungarianPrefix*
- *PrivateMemberCase, PrivateMemberPrefix, PrivateMemberSuffix, PrivateMemberIgnoredRegexp, PrivateMemberHungarianPrefix*
- *PrivateMethodCase, PrivateMethodPrefix, PrivateMethodSuffix, PrivateMethodIgnoredRegexp*
- *ProtectedMemberCase, ProtectedMemberPrefix, ProtectedMemberSuffix,*

ProtectedMemberIgnoredRegexp, ProtectedMemberHungarianPrefix

- *ProtectedMethodCase, ProtectedMethodPrefix, ProtectedMethodSuffix, ProtectedMethodIgnoredRegexp*
- *PublicMemberCase, PublicMemberPrefix, PublicMemberSuffix, PublicMemberIgnoredRegexp, PublicMemberHungarianPrefix*
- *PublicMethodCase, PublicMethodPrefix, PublicMethodSuffix, PublicMethodIgnoredRegexp*
- *ScopedEnumConstantCase, ScopedEnumConstantPrefix, ScopedEnumConstantSuffix, ScopedEnumConstantIgnoredRegexp*
- *StaticConstantCase, StaticConstantPrefix, StaticConstantSuffix, StaticConstantIgnoredRegexp, StaticConstantHungarianPrefix*
- *StaticVariableCase, StaticVariablePrefix, StaticVariableSuffix, StaticVariableIgnoredRegexp, StaticVariableHungarianPrefix*
- *StructCase, StructPrefix, StructSuffix, StructIgnoredRegexp*
- *TemplateParameterCase, TemplateParameterPrefix, TemplateParameterSuffix, TemplateParameterIgnoredRegexp*
- *TemplateTemplateParameterCase, TemplateTemplateParameterPrefix, TemplateTemplateParameterSuffix, TemplateTemplateParameterIgnoredRegexp*
- *TypeAliasCase, TypeAliasPrefix, TypeAliasSuffix, TypeAliasIgnoredRegexp*
- *TypedefCase, TypedefPrefix, TypedefSuffix, TypedefIgnoredRegexp*
- *TypeTemplateParameterCase, TypeTemplateParameterPrefix, TypeTemplateParameterSuffix, TypeTemplateParameterIgnoredRegexp*
- *UnionCase, UnionPrefix, UnionSuffix, UnionIgnoredRegexp*
- *ValueTemplateParameterCase, ValueTemplateParameterPrefix, ValueTemplateParameterSuffix, ValueTemplateParameterIgnoredRegexp*
- *VariableCase, VariablePrefix, VariableSuffix, VariableIgnoredRegexp,*

VariableHungarianPrefix

- *VirtualMethodCase*, *VirtualMethodPrefix*, *VirtualMethodSuffix*, *VirtualMethodIgnoredRegexp*

AbstractClassCase
When defined, the check will ensure abstract class names conform to the selected casing.

AbstractClassPrefix
When defined, the check will ensure abstract class names will add the prefixed with the given value (regardless of casing).

AbstractClassIgnoredRegexp
Identifier naming checks won't be enforced for abstract class names matching this regular expression.

AbstractClassSuffix
When defined, the check will ensure abstract class names will add the suffix with the given value (regardless of casing).

AbstractClassHungarianPrefix
When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- AbstractClassCase of **lower_case**
- AbstractClassPrefix of **pre_**
- AbstractClassSuffix of **_post**
- AbstractClassHungarianPrefix of **On**

Identifies and/or transforms abstract class names as follows:

Before:

```
class ABSTRACT_CLASS {
public:
  ABSTRACT_CLASS();
```

```
};
```

After:

```
class pre_abstract_class_post {
public:
 pre_abstract_class_post();
};
```

AggressiveDependentMemberLookup

When set to *true* the check will look in dependent base classes for dependent member references that need changing. This can lead to errors with template specializations so the default value is *false*.

For example using values of:

- ClassMemberCase of **lower_case**

Before:

```
template <typename T>
struct Base {
 T BadNamedMember;
};

template <typename T>
struct Derived : Base<T> {
 void reset() {
  this->BadNamedMember = 0;
 }
};
```

After if AggressiveDependentMemberLookup is *false*:

```
template <typename T>
struct Base {
 T bad_named_member;
};

template <typename T>
```

```
struct Derived : Base<T> {
  void reset() {
    this->BadNamedMember = 0;
  }
};
```

After if AggressiveDependentMemberLookup is *true*:

```
template <typename T>
struct Base {
  T bad_named_member;
};

template <typename T>
struct Derived : Base<T> {
  void reset() {
    this->bad_named_member = 0;
  }
};
```

ClassCase
When defined, the check will ensure class names conform to the selected casing.

ClassPrefix
When defined, the check will ensure class names will add the prefixed with the given value (regardless of casing).

ClassIgnoredRegexp
Identifier naming checks won’t be enforced for class names matching this regular expression.

ClassSuffix
When defined, the check will ensure class names will add the suffix with the given value (regardless of casing).

ClassHungarianPrefix
When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- ClassCase of **lower_case**
- ClassPrefix of **pre_**
- ClassSuffix of **_post**
- ClassHungarianPrefix of **On**

  Identifies and/or transforms class names as follows:

  Before:

```
class FOO {
public:
  FOO();
  ~FOO();
};
```

  After:

```
class pre_foo_post {
public:
  pre_foo_post();
  ~pre_foo_post();
};
```

ClassConstantCase
When defined, the check will ensure class constant names conform to the selected casing.

ClassConstantPrefix
When defined, the check will ensure class constant names will add the prefixed with the given value (regardless of casing).

ClassConstantIgnoredRegexp
Identifier naming checks won’t be enforced for class constant names matching this regular expression.

ClassConstantSuffix
When defined, the check will ensure class constant names will add the suffix with the given value (regardless of casing).

ClassConstantHungarianPrefix
When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- ClassConstantCase of **lower_case**
- ClassConstantPrefix of **pre_**
- ClassConstantSuffix of **_post**
- ClassConstantHungarianPrefix of **On**

Identifies and/or transforms class constant names as follows:

Before:

```
class FOO {
public:
  static const int CLASS_CONSTANT;
};
```

After:

```
class FOO {
public:
  static const int pre_class_constant_post;
};
```

ClassMemberCase
When defined, the check will ensure class member names conform to the selected casing.

ClassMemberPrefix
When defined, the check will ensure class member names will add the prefixed with the given value (regardless of casing).

ClassMemberIgnoredRegexp
Identifier naming checks won’t be enforced for class member names matching this regular expression.

ClassMemberSuffix
When defined, the check will ensure class member names will add the suffix with the given value (regardless of casing).

ClassMemberHungarianPrefix
When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- ClassMemberCase of **lower_case**
- ClassMemberPrefix of **pre_**
- ClassMemberSuffix of **_post**
- ClassMemberHungarianPrefix of **On**

Identifies and/or transforms class member names as follows:

Before:

```
class FOO {
public:
  static int CLASS_CONSTANT;
};
```

After:

```
class FOO {
public:
  static int pre_class_constant_post;
};
```

ClassMethodCase
When defined, the check will ensure class method names conform to the selected casing.

ClassMethodPrefix
When defined, the check will ensure class method names will add the prefixed with the given value (regardless of casing).

ClassMethodIgnoredRegexp
Identifier naming checks won't be enforced for class method names matching this regular expression.

ClassMethodSuffix
When defined, the check will ensure class method names will add the suffix with the given value (regardless of casing).

For example using values of:

- ClassMethodCase of **lower_case**
- ClassMethodPrefix of **pre_**
- ClassMethodSuffix of **_post**

Identifies and/or transforms class method names as follows:

Before:

```
class FOO {
public:
  int CLASS_MEMBER();
};
```

After:

```
class FOO {
public:
  int pre_class_member_post();
};
```

ConstantCase
When defined, the check will ensure constant names conform to the selected casing.

ConstantPrefix
When defined, the check will ensure constant names will add the prefixed with the given value (regardless of casing).

ConstantIgnoredRegexp

Identifier naming checks won’t be enforced for constant names matching this regular expression.

ConstantSuffix
When defined, the check will ensure constant names will add the suffix with the given value (regardless of casing).

ConstantHungarianPrefix
When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- ConstantCase of **lower_case**
- ConstantPrefix of **pre_**
- ConstantSuffix of **_post**
- ConstantHungarianPrefix of **On**

Identifies and/or transforms constant names as follows:

Before:

```
void function() { unsigned const MyConst_array[] = {1, 2, 3}; }
```

After:

```
void function() { unsigned const pre_myconst_array_post[] = {1, 2, 3}; }
```

ConstantMemberCase
When defined, the check will ensure constant member names conform to the selected casing.

ConstantMemberPrefix
When defined, the check will ensure constant member names will add the prefixed with the given value (regardless of casing).

ConstantMemberIgnoredRegexp
Identifier naming checks won’t be enforced for constant member names matching this regular expression.

ConstantMemberSuffix
When defined, the check will ensure constant member names will add the suffix with the given value (regardless of casing).

ConstantMemberHungarianPrefix
When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- ConstantMemberCase of **lower_case**
- ConstantMemberPrefix of **pre_**
- ConstantMemberSuffix of **_post**
- ConstantMemberHungarianPrefix of **On**

Identifies and/or transforms constant member names as follows:

Before:

```
class Foo {
  char const MY_ConstMember_string[4] = "123";
}
```

After:

```
class Foo {
  char const pre_my_constmember_string_post[4] = "123";
}
```

ConstantParameterCase
When defined, the check will ensure constant parameter names conform to the selected casing.

ConstantParameterPrefix
When defined, the check will ensure constant parameter names will add the prefixed with the given value (regardless of casing).

ConstantParameterIgnoredRegexp

Identifier naming checks won’t be enforced for constant parameter names matching this regular expression.

ConstantParameterSuffix
When defined, the check will ensure constant parameter names will add the suffix with the given value (regardless of casing).

ConstantParameterHungarianPrefix
When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- ConstantParameterCase of **lower_case**
- ConstantParameterPrefix of **pre_**
- ConstantParameterSuffix of **_post**
- ConstantParameterHungarianPrefix of **On**

Identifies and/or transforms constant parameter names as follows:

Before:

```
void GLOBAL_FUNCTION(int PARAMETER_1, int const CONST_parameter);
```

After:

```
void GLOBAL_FUNCTION(int PARAMETER_1, int const pre_const_parameter_post);
```

ConstantPointerParameterCase
When defined, the check will ensure constant pointer parameter names conform to the selected casing.

ConstantPointerParameterPrefix
When defined, the check will ensure constant pointer parameter names will add the prefixed with the given value (regardless of casing).

ConstantPointerParameterIgnoredRegexp

Identifier naming checks won’t be enforced for constant pointer parameter names matching this regular expression.

ConstantPointerParameterSuffix
When defined, the check will ensure constant pointer parameter names will add the suffix with the given value (regardless of casing).

ConstantPointerParameterHungarianPrefix
When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- ConstantPointerParameterCase of **lower_case**
- ConstantPointerParameterPrefix of **pre_**
- ConstantPointerParameterSuffix of **_post**
- ConstantPointerParameterHungarianPrefix of **On**

Identifies and/or transforms constant pointer parameter names as follows:

Before:

```
void GLOBAL_FUNCTION(int const *CONST_parameter);
```

After:

```
void GLOBAL_FUNCTION(int const *pre_const_parameter_post);
```

ConstexprFunctionCase
When defined, the check will ensure constexpr function names conform to the selected casing.

ConstexprFunctionPrefix
When defined, the check will ensure constexpr function names will add the prefixed with the given value (regardless of casing).

ConstexprFunctionIgnoredRegexp
Identifier naming checks won’t be enforced for constexpr function names matching this regular

expression.

ConstexprFunctionSuffix
When defined, the check will ensure constexpr function names will add the suffix with the given value (regardless of casing).

For example using values of:

- ConstexprFunctionCase of **lower_case**
- ConstexprFunctionPrefix of **pre_**
- ConstexprFunctionSuffix of **_post**

Identifies and/or transforms constexpr function names as follows:

Before:

```
constexpr int CE_function() { return 3; }
```

After:

```
constexpr int pre_ce_function_post() { return 3; }
```

ConstexprMethodCase
When defined, the check will ensure constexpr method names conform to the selected casing.

ConstexprMethodPrefix
When defined, the check will ensure constexpr method names will add the prefixed with the given value (regardless of casing).

ConstexprMethodIgnoredRegexp
Identifier naming checks won’t be enforced for constexpr method names matching this regular expression.

ConstexprMethodSuffix
When defined, the check will ensure constexpr method names will add the suffix with the given value (regardless of casing).

For example using values of:

- ConstexprMethodCase of **lower_case**
- ConstexprMethodPrefix of **pre_**
- ConstexprMethodSuffix of **_post**

  Identifies and/or transforms constexpr method names as follows:

  Before:

```
class Foo {
public:
  constexpr int CST_expr_Method() { return 2; }
}
```

  After:

```
class Foo {
public:
  constexpr int pre_cst_expr_method_post() { return 2; }
}
```

ConstexprVariableCase
When defined, the check will ensure constexpr variable names conform to the selected casing.

ConstexprVariablePrefix
When defined, the check will ensure constexpr variable names will add the prefixed with the given value (regardless of casing).

ConstexprVariableIgnoredRegexp
Identifier naming checks won’t be enforced for constexpr variable names matching this regular expression.

ConstexprVariableSuffix
When defined, the check will ensure constexpr variable names will add the suffix with the given value (regardless of casing).

ConstexprVariableHungarianPrefix
When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- ConstexprVariableCase of **lower_case**
- ConstexprVariablePrefix of **pre_**
- ConstexprVariableSuffix of **_post**
- ConstexprVariableHungarianPrefix of **On**

Identifies and/or transforms constexpr variable names as follows:

Before:

```
constexpr int ConstExpr_variable = MyConstant;
```

After:

```
constexpr int pre_constexpr_variable_post = MyConstant;
```

EnumCase
When defined, the check will ensure enumeration names conform to the selected casing.

EnumPrefix
When defined, the check will ensure enumeration names will add the prefixed with the given value (regardless of casing).

EnumIgnoredRegexp
Identifier naming checks won't be enforced for enumeration names matching this regular expression.

EnumSuffix
When defined, the check will ensure enumeration names will add the suffix with the given value (regardless of casing).

For example using values of:

- EnumCase of **lower_case**
- EnumPrefix of **pre_**

- EnumSuffix of **_post**

  Identifies and/or transforms enumeration names as follows:

  Before:

```
enum FOO { One, Two, Three };
```

  After:

```
enum pre_foo_post { One, Two, Three };
```

EnumConstantCase
When defined, the check will ensure enumeration constant names conform to the selected casing.

EnumConstantPrefix
When defined, the check will ensure enumeration constant names will add the prefixed with the given value (regardless of casing).

EnumConstantIgnoredRegexp
Identifier naming checks won’t be enforced for enumeration constant names matching this regular expression.

EnumConstantSuffix
When defined, the check will ensure enumeration constant names will add the suffix with the given value (regardless of casing).

EnumConstantHungarianPrefix
When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

  For example using values of:

- EnumConstantCase of **lower_case**
- EnumConstantPrefix of **pre_**
- EnumConstantSuffix of **_post**
- EnumConstantHungarianPrefix of **On**

Identifies and/or transforms enumeration constant names as follows:

Before:

```
enum FOO { One, Two, Three };
```

After:

```
enum FOO { pre_One_post, pre_Two_post, pre_Three_post };
```

FunctionCase
When defined, the check will ensure function names conform to the selected casing.

FunctionPrefix
When defined, the check will ensure function names will add the prefixed with the given value (regardless of casing).

FunctionIgnoredRegexp
Identifier naming checks won't be enforced for function names matching this regular expression.

FunctionSuffix
When defined, the check will ensure function names will add the suffix with the given value (regardless of casing).

For example using values of:

- FunctionCase of **lower_case**
- FunctionPrefix of **pre_**
- FunctionSuffix of **_post**

Identifies and/or transforms function names as follows:

Before:

```
char MY_Function_string();
```

After:

char pre_my_function_string_post();

GetConfigPerFile
When *true* the check will look for the configuration for where an identifier is declared. Useful for when included header files use a different style. Default value is *true*.

GlobalConstantCase
When defined, the check will ensure global constant names conform to the selected casing.

GlobalConstantPrefix
When defined, the check will ensure global constant names will add the prefixed with the given value (regardless of casing).

GlobalConstantIgnoredRegexp
Identifier naming checks won't be enforced for global constant names matching this regular expression.

GlobalConstantSuffix
When defined, the check will ensure global constant names will add the suffix with the given value (regardless of casing).

GlobalConstantHungarianPrefix
When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- GlobalConstantCase of **lower_case**
- GlobalConstantPrefix of **pre_**
- GlobalConstantSuffix of **_post**
- GlobalConstantHungarianPrefix of **On**

Identifies and/or transforms global constant names as follows:

Before:

unsigned const MyConstGlobal_array[] = {1, 2, 3};

After:

unsigned const pre_myconstglobal_array_post[] = {1, 2, 3};

GlobalConstantPointerCase
When defined, the check will ensure global constant pointer names conform to the selected casing.

GlobalConstantPointerPrefix
When defined, the check will ensure global constant pointer names will add the prefixed with the given value (regardless of casing).

GlobalConstantPointerIgnoredRegexp
Identifier naming checks won't be enforced for global constant pointer names matching this regular expression.

GlobalConstantPointerSuffix
When defined, the check will ensure global constant pointer names will add the suffix with the given value (regardless of casing).

GlobalConstantPointerHungarianPrefix
When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- GlobalConstantPointerCase of **lower_case**
- GlobalConstantPointerPrefix of **pre_**
- GlobalConstantPointerSuffix of **_post**
- GlobalConstantPointerHungarianPrefix of **On**

Identifies and/or transforms global constant pointer names as follows:

Before:

int *const MyConstantGlobalPointer = nullptr;

After:

int *const pre_myconstantglobalpointer_post = nullptr;

GlobalFunctionCase
When defined, the check will ensure global function names conform to the selected casing.

GlobalFunctionPrefix
When defined, the check will ensure global function names will add the prefixed with the given value (regardless of casing).

GlobalFunctionIgnoredRegexp
Identifier naming checks won't be enforced for global function names matching this regular expression.

GlobalFunctionSuffix
When defined, the check will ensure global function names will add the suffix with the given value (regardless of casing).

For example using values of:

- GlobalFunctionCase of **lower_case**
- GlobalFunctionPrefix of **pre_**
- GlobalFunctionSuffix of **_post**

Identifies and/or transforms global function names as follows:

Before:

void GLOBAL_FUNCTION(int PARAMETER_1, int const CONST_parameter);

After:

void pre_global_function_post(int PARAMETER_1, int const CONST_parameter);

GlobalPointerCase
When defined, the check will ensure global pointer names conform to the selected casing.

GlobalPointerPrefix
When defined, the check will ensure global pointer names will add the prefixed with the given

value (regardless of casing).

GlobalPointerIgnoredRegexp
Identifier naming checks won't be enforced for global pointer names matching this regular expression.

GlobalPointerSuffix
When defined, the check will ensure global pointer names will add the suffix with the given value (regardless of casing).

GlobalPointerHungarianPrefix
When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- GlobalPointerCase of **lower_case**
- GlobalPointerPrefix of **pre_**
- GlobalPointerSuffix of **_post**
- GlobalPointerHungarianPrefix of **On**

Identifies and/or transforms global pointer names as follows:

Before:

```
int *GLOBAL3;
```

After:

```
int *pre_global3_post;
```

GlobalVariableCase
When defined, the check will ensure global variable names conform to the selected casing.

GlobalVariablePrefix
When defined, the check will ensure global variable names will add the prefixed with the given value (regardless of casing).

GlobalVariableIgnoredRegexp
Identifier naming checks won't be enforced for global variable names matching this regular expression.

GlobalVariableSuffix
When defined, the check will ensure global variable names will add the suffix with the given value (regardless of casing).

GlobalVariableHungarianPrefix
When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- GlobalVariableCase of **lower_case**
- GlobalVariablePrefix of **pre_**
- GlobalVariableSuffix of **_post**
- GlobalVariableHungarianPrefix of **On**

Identifies and/or transforms global variable names as follows:

Before:

int GLOBAL3;

After:

int pre_global3_post;

IgnoreMainLikeFunctions
When set to *true* functions that have a similar signature to **main** or **wmain** won't enforce checks on the names of their parameters. Default value is *false*.

InlineNamespaceCase
When defined, the check will ensure inline namespaces names conform to the selected casing.

InlineNamespacePrefix

When defined, the check will ensure inline namespaces names will add the prefixed with the given value (regardless of casing).

InlineNamespaceIgnoredRegexp
Identifier naming checks won't be enforced for inline namespaces names matching this regular expression.

InlineNamespaceSuffix
When defined, the check will ensure inline namespaces names will add the suffix with the given value (regardless of casing).

For example using values of:

- InlineNamespaceCase of **lower_case**
- InlineNamespacePrefix of **pre_**
- InlineNamespaceSuffix of **_post**

Identifies and/or transforms inline namespaces names as follows:

Before:

```
namespace FOO_NS {
inline namespace InlineNamespace {
...
}
} // namespace FOO_NS
```

After:

```
namespace FOO_NS {
inline namespace pre_inlinenamespace_post {
...
}
} // namespace FOO_NS
```

LocalConstantCase
When defined, the check will ensure local constant names conform to the selected casing.

LocalConstantPrefix
When defined, the check will ensure local constant names will add the prefixed with the given value (regardless of casing).

LocalConstantIgnoredRegexp
Identifier naming checks won’t be enforced for local constant names matching this regular expression.

LocalConstantSuffix
When defined, the check will ensure local constant names will add the suffix with the given value (regardless of casing).

LocalConstantHungarianPrefix
When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- LocalConstantCase of **lower_case**
- LocalConstantPrefix of **pre_**
- LocalConstantSuffix of **_post**
- LocalConstantHungarianPrefix of **On**

Identifies and/or transforms local constant names as follows:

Before:

```
void foo() { int const local_Constant = 3; }
```

After:

```
void foo() { int const pre_local_constant_post = 3; }
```

LocalConstantPointerCase
When defined, the check will ensure local constant pointer names conform to the selected casing.

LocalConstantPointerPrefix

When defined, the check will ensure local constant pointer names will add the prefixed with the given value (regardless of casing).

LocalConstantPointerIgnoredRegexp
Identifier naming checks won't be enforced for local constant pointer names matching this regular expression.

LocalConstantPointerSuffix
When defined, the check will ensure local constant pointer names will add the suffix with the given value (regardless of casing).

LocalConstantPointerHungarianPrefix
When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- LocalConstantPointerCase of **lower_case**
- LocalConstantPointerPrefix of **pre_**
- LocalConstantPointerSuffix of **_post**
- LocalConstantPointerHungarianPrefix of **On**

Identifies and/or transforms local constant pointer names as follows:

Before:

```
void foo() { int const *local_Constant = 3; }
```

After:

```
void foo() { int const *pre_local_constant_post = 3; }
```

LocalPointerCase
When defined, the check will ensure local pointer names conform to the selected casing.

LocalPointerPrefix
When defined, the check will ensure local pointer names will add the prefixed with the given value

(regardless of casing).

LocalPointerIgnoredRegexp
Identifier naming checks won’t be enforced for local pointer names matching this regular expression.

LocalPointerSuffix
When defined, the check will ensure local pointer names will add the suffix with the given value (regardless of casing).

LocalPointerHungarianPrefix
When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- LocalPointerCase of **lower_case**
- LocalPointerPrefix of **pre_**
- LocalPointerSuffix of **_post**
- LocalPointerHungarianPrefix of **On**

Identifies and/or transforms local pointer names as follows:

Before:

```
void foo() { int *local_Constant; }
```

After:

```
void foo() { int *pre_local_constant_post; }
```

LocalVariableCase
When defined, the check will ensure local variable names conform to the selected casing.

LocalVariablePrefix
When defined, the check will ensure local variable names will add the prefixed with the given value (regardless of casing).

LocalVariableIgnoredRegexp
Identifier naming checks won't be enforced for local variable names matching this regular expression.

For example using values of:

- LocalVariableCase of **CamelCase**
- LocalVariableIgnoredRegexp of **\w{1,2}**

Will exclude variables with a length less than or equal to 2 from the camel case check applied to other variables.

LocalVariableSuffix
When defined, the check will ensure local variable names will add the suffix with the given value (regardless of casing).

LocalVariableHungarianPrefix
When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- LocalVariableCase of **lower_case**
- LocalVariablePrefix of **pre_**
- LocalVariableSuffix of **_post**
- LocalVariableHungarianPrefix of **On**

Identifies and/or transforms local variable names as follows:

Before:

```
void foo() { int local_Constant; }
```

After:

```
void foo() { int pre_local_constant_post; }
```

MacroDefinitionCase
When defined, the check will ensure macro definitions conform to the selected casing.

MacroDefinitionPrefix
When defined, the check will ensure macro definitions will add the prefixed with the given value (regardless of casing).

MacroDefinitionIgnoredRegexp
Identifier naming checks won't be enforced for macro definitions matching this regular expression.

MacroDefinitionSuffix
When defined, the check will ensure macro definitions will add the suffix with the given value (regardless of casing).

For example using values of:

- MacroDefinitionCase of **lower_case**
- MacroDefinitionPrefix of **pre_**
- MacroDefinitionSuffix of **_post**

Identifies and/or transforms macro definitions as follows:

Before:

```
#define MY_MacroDefinition
```

After:

```
#define pre_my_macro_definition_post
```

Note: This will not warn on builtin macros or macros defined on the command line using the **-D** flag.

MemberCase
When defined, the check will ensure member names conform to the selected casing.

MemberPrefix
When defined, the check will ensure member names will add the prefixed with the given value

(regardless of casing).

MemberIgnoredRegexp
Identifier naming checks won't be enforced for member names matching this regular expression.

MemberSuffix
When defined, the check will ensure member names will add the suffix with the given value (regardless of casing).

MemberHungarianPrefix
When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- MemberCase of **lower_case**
- MemberPrefix of **pre_**
- MemberSuffix of **_post**
- MemberHungarianPrefix of **On**

Identifies and/or transforms member names as follows:

Before:

```
class Foo {
  char MY_ConstMember_string[4];
}
```

After:

```
class Foo {
  char pre_my_constmember_string_post[4];
}
```

MethodCase
When defined, the check will ensure method names conform to the selected casing.

MethodPrefix
When defined, the check will ensure method names will add the prefixed with the given value (regardless of casing).

MethodIgnoredRegexp
Identifier naming checks won't be enforced for method names matching this regular expression.

MethodSuffix
When defined, the check will ensure method names will add the suffix with the given value (regardless of casing).

For example using values of:

- MethodCase of **lower_case**
- MethodPrefix of **pre_**
- MethodSuffix of **_post**

Identifies and/or transforms method names as follows:

Before:

```
class Foo {
  char MY_Method_string();
}
```

After:

```
class Foo {
  char pre_my_method_string_post();
}
```

NamespaceCase
When defined, the check will ensure namespace names conform to the selected casing.

NamespacePrefix
When defined, the check will ensure namespace names will add the prefixed with the given value (regardless of casing).

NamespaceIgnoredRegexp
Identifier naming checks won't be enforced for namespace names matching this regular expression.

NamespaceSuffix
When defined, the check will ensure namespace names will add the suffix with the given value (regardless of casing).

For example using values of:

- NamespaceCase of **lower_case**
- NamespacePrefix of **pre_**
- NamespaceSuffix of **_post**

Identifies and/or transforms namespace names as follows:

Before:

```
namespace FOO_NS {
...
}
```

After:

```
namespace pre_foo_ns_post {
...
}
```

ParameterCase
When defined, the check will ensure parameter names conform to the selected casing.

ParameterPrefix
When defined, the check will ensure parameter names will add the prefixed with the given value (regardless of casing).

ParameterIgnoredRegexp
Identifier naming checks won't be enforced for parameter names matching this regular expression.

ParameterSuffix
When defined, the check will ensure parameter names will add the suffix with the given value (regardless of casing).

ParameterHungarianPrefix
When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- ParameterCase of **lower_case**
- ParameterPrefix of **pre_**
- ParameterSuffix of **_post**
- ParameterHungarianPrefix of **On**

Identifies and/or transforms parameter names as follows:

Before:

```
void GLOBAL_FUNCTION(int PARAMETER_1, int const CONST_parameter);
```

After:

```
void GLOBAL_FUNCTION(int pre_parameter_post, int const CONST_parameter);
```

ParameterPackCase
When defined, the check will ensure parameter pack names conform to the selected casing.

ParameterPackPrefix
When defined, the check will ensure parameter pack names will add the prefixed with the given value (regardless of casing).

ParameterPackIgnoredRegexp
Identifier naming checks won’t be enforced for parameter pack names matching this regular expression.

ParameterPackSuffix

When defined, the check will ensure parameter pack names will add the suffix with the given value (regardless of casing).

For example using values of:

- ParameterPackCase of **lower_case**
- ParameterPackPrefix of **pre_**
- ParameterPackSuffix of **_post**

Identifies and/or transforms parameter pack names as follows:

Before:

```
template <typename... TYPE_parameters> {
  void FUNCTION(int... TYPE_parameters);
}
```

After:

```
template <typename... TYPE_parameters> {
  void FUNCTION(int... pre_type_parameters_post);
}
```

PointerParameterCase
When defined, the check will ensure pointer parameter names conform to the selected casing.

PointerParameterPrefix
When defined, the check will ensure pointer parameter names will add the prefixed with the given value (regardless of casing).

PointerParameterIgnoredRegexp
Identifier naming checks won’t be enforced for pointer parameter names matching this regular expression.

PointerParameterSuffix
When defined, the check will ensure pointer parameter names will add the suffix with the given value (regardless of casing).

PointerParameterHungarianPrefix
When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- PointerParameterCase of **lower_case**
- PointerParameterPrefix of **pre_**
- PointerParameterSuffix of **_post**
- PointerParameterHungarianPrefix of **On**

Identifies and/or transforms pointer parameter names as follows:

Before:

void FUNCTION(int *PARAMETER);

After:

void FUNCTION(int *pre_parameter_post);

PrivateMemberCase
When defined, the check will ensure private member names conform to the selected casing.

PrivateMemberPrefix
When defined, the check will ensure private member names will add the prefixed with the given value (regardless of casing).

PrivateMemberIgnoredRegexp
Identifier naming checks won’t be enforced for private member names matching this regular expression.

PrivateMemberSuffix
When defined, the check will ensure private member names will add the suffix with the given value (regardless of casing).

PrivateMemberHungarianPrefix

When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- PrivateMemberCase of **lower_case**
- PrivateMemberPrefix of **pre_**
- PrivateMemberSuffix of **_post**
- PrivateMemberHungarianPrefix of **On**

Identifies and/or transforms private member names as follows:

Before:

```
class Foo {
private:
  int Member_Variable;
}
```

After:

```
class Foo {
private:
  int pre_member_variable_post;
}
```

PrivateMethodCase
When defined, the check will ensure private method names conform to the selected casing.

PrivateMethodPrefix
When defined, the check will ensure private method names will add the prefixed with the given value (regardless of casing).

PrivateMethodIgnoredRegexp
Identifier naming checks won’t be enforced for private method names matching this regular expression.

PrivateMethodSuffix
When defined, the check will ensure private method names will add the suffix with the given value (regardless of casing).

For example using values of:

- PrivateMethodCase of **lower_case**
- PrivateMethodPrefix of **pre_**
- PrivateMethodSuffix of **_post**

Identifies and/or transforms private method names as follows:

Before:

```
class Foo {
private:
  int Member_Method();
}
```

After:

```
class Foo {
private:
  int pre_member_method_post();
}
```

ProtectedMemberCase
When defined, the check will ensure protected member names conform to the selected casing.

ProtectedMemberPrefix
When defined, the check will ensure protected member names will add the prefixed with the given value (regardless of casing).

ProtectedMemberIgnoredRegexp
Identifier naming checks won’t be enforced for protected member names matching this regular expression.

ProtectedMemberSuffix

When defined, the check will ensure protected member names will add the suffix with the given value (regardless of casing).

ProtectedMemberHungarianPrefix

When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- ProtectedMemberCase of **lower_case**
- ProtectedMemberPrefix of **pre_**
- ProtectedMemberSuffix of **_post**
- ProtectedMemberHungarianPrefix of **On**

Identifies and/or transforms protected member names as follows:

Before:

```
class Foo {
protected:
  int Member_Variable;
}
```

After:

```
class Foo {
protected:
  int pre_member_variable_post;
}
```

ProtectedMethodCase

When defined, the check will ensure protected method names conform to the selected casing.

ProtectedMethodPrefix

When defined, the check will ensure protected method names will add the prefixed with the given value (regardless of casing).

ProtectedMethodIgnoredRegexp
Identifier naming checks won’t be enforced for protected method names matching this regular expression.

ProtectedMethodSuffix
When defined, the check will ensure protected method names will add the suffix with the given value (regardless of casing).

For example using values of:

- ProtectedMethodCase of **lower_case**
- ProtectedMethodPrefix of **pre_**
- ProtectedMethodSuffix of **_post**

Identifies and/or transforms protect method names as follows:

Before:

```
class Foo {
protected:
  int Member_Method();
}
```

After:

```
class Foo {
protected:
  int pre_member_method_post();
}
```

PublicMemberCase
When defined, the check will ensure public member names conform to the selected casing.

PublicMemberPrefix
When defined, the check will ensure public member names will add the prefixed with the given value (regardless of casing).

PublicMemberIgnoredRegexp

Identifier naming checks won’t be enforced for public member names matching this regular expression.

PublicMemberSuffix
When defined, the check will ensure public member names will add the suffix with the given value (regardless of casing).

PublicMemberHungarianPrefix
When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- PublicMemberCase of **lower_case**
- PublicMemberPrefix of **pre_**
- PublicMemberSuffix of **_post**
- PublicMemberHungarianPrefix of **On**

Identifies and/or transforms public member names as follows:

Before:

```
class Foo {
public:
  int Member_Variable;
}
```

After:

```
class Foo {
public:
  int pre_member_variable_post;
}
```

PublicMethodCase
When defined, the check will ensure public method names conform to the selected casing.

PublicMethodPrefix
When defined, the check will ensure public method names will add the prefixed with the given value (regardless of casing).

PublicMethodIgnoredRegexp
Identifier naming checks won’t be enforced for public method names matching this regular expression.

PublicMethodSuffix
When defined, the check will ensure public method names will add the suffix with the given value (regardless of casing).

For example using values of:

- PublicMethodCase of **lower_case**
- PublicMethodPrefix of **pre_**
- PublicMethodSuffix of **_post**

Identifies and/or transforms public method names as follows:

Before:

```
class Foo {
public:
  int Member_Method();
}
```

After:

```
class Foo {
public:
  int pre_member_method_post();
}
```

ScopedEnumConstantCase
When defined, the check will ensure scoped enum constant names conform to the selected casing.

ScopedEnumConstantPrefix

When defined, the check will ensure scoped enum constant names will add the prefixed with the given value (regardless of casing).

ScopedEnumConstantIgnoredRegexp
Identifier naming checks won't be enforced for scoped enum constant names matching this regular expression.

ScopedEnumConstantSuffix
When defined, the check will ensure scoped enum constant names will add the suffix with the given value (regardless of casing).

ScopedEnumConstantHungarianPrefix
When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- ScopedEnumConstantCase of **lower_case**
- ScopedEnumConstantPrefix of **pre_**
- ScopedEnumConstantSuffix of **_post**
- ScopedEnumConstantHungarianPrefix of **On**

Identifies and/or transforms enumeration constant names as follows:

Before:

```
enum class FOO { One, Two, Three };
```

After:

```
enum class FOO { pre_One_post, pre_Two_post, pre_Three_post };
```

StaticConstantCase
When defined, the check will ensure static constant names conform to the selected casing.

StaticConstantPrefix
When defined, the check will ensure static constant names will add the prefixed with the given

value (regardless of casing).

StaticConstantIgnoredRegexp
Identifier naming checks won’t be enforced for static constant names matching this regular expression.

StaticConstantSuffix
When defined, the check will ensure static constant names will add the suffix with the given value (regardless of casing).

StaticConstantHungarianPrefix
When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- StaticConstantCase of **lower_case**
- StaticConstantPrefix of **pre_**
- StaticConstantSuffix of **_post**
- StaticConstantHungarianPrefix of **On**

Identifies and/or transforms static constant names as follows:

Before:

```
static unsigned const MyConstStatic_array[] = {1, 2, 3};
```

After:

```
static unsigned const pre_myconststatic_array_post[] = {1, 2, 3};
```

StaticVariableCase
When defined, the check will ensure static variable names conform to the selected casing.

StaticVariablePrefix
When defined, the check will ensure static variable names will add the prefixed with the given value (regardless of casing).

StaticVariableIgnoredRegexp
Identifier naming checks won't be enforced for static variable names matching this regular expression.

StaticVariableSuffix
When defined, the check will ensure static variable names will add the suffix with the given value (regardless of casing).

StaticVariableHungarianPrefix
When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- StaticVariableCase of **lower_case**
- StaticVariablePrefix of **pre_**
- StaticVariableSuffix of **_post**
- StaticVariableHungarianPrefix of **On**

Identifies and/or transforms static variable names as follows:

Before:

```
static unsigned MyStatic_array[] = {1, 2, 3};
```

After:

```
static unsigned pre_mystatic_array_post[] = {1, 2, 3};
```

StructCase
When defined, the check will ensure struct names conform to the selected casing.

StructPrefix
When defined, the check will ensure struct names will add the prefixed with the given value (regardless of casing).

StructIgnoredRegexp

Identifier naming checks won’t be enforced for struct names matching this regular expression.

StructSuffix
When defined, the check will ensure struct names will add the suffix with the given value (regardless of casing).

For example using values of:

- StructCase of **lower_case**
- StructPrefix of **pre_**
- StructSuffix of **_post**

Identifies and/or transforms struct names as follows:

Before:

```
struct FOO {
  FOO();
  ~FOO();
};
```

After:

```
struct pre_foo_post {
  pre_foo_post();
  ~pre_foo_post();
};
```

TemplateParameterCase
When defined, the check will ensure template parameter names conform to the selected casing.

TemplateParameterPrefix
When defined, the check will ensure template parameter names will add the prefixed with the given value (regardless of casing).

TemplateParameterIgnoredRegexp
Identifier naming checks won’t be enforced for template parameter names matching this regular expression.

TemplateParameterSuffix
When defined, the check will ensure template parameter names will add the suffix with the given value (regardless of casing).

For example using values of:

- TemplateParameterCase of **lower_case**
- TemplateParameterPrefix of **pre_**
- TemplateParameterSuffix of **_post**

Identifies and/or transforms template parameter names as follows:

Before:

```
template <typename T> class Foo {};
```

After:

```
template <typename pre_t_post> class Foo {};
```

TemplateTemplateParameterCase
When defined, the check will ensure template template parameter names conform to the selected casing.

TemplateTemplateParameterPrefix
When defined, the check will ensure template template parameter names will add the prefixed with the given value (regardless of casing).

TemplateTemplateParameterIgnoredRegexp
Identifier naming checks won’t be enforced for template template parameter names matching this regular expression.

TemplateTemplateParameterSuffix
When defined, the check will ensure template template parameter names will add the suffix with the given value (regardless of casing).

For example using values of:

- TemplateTemplateParameterCase of **lower_case**
- TemplateTemplateParameterPrefix of **pre_**
- TemplateTemplateParameterSuffix of **_post**

Identifies and/or transforms template template parameter names as follows:

Before:

```
template <template <typename> class TPL_parameter, int COUNT_params,
    typename... TYPE_parameters>
```

After:

```
template <template <typename> class pre_tpl_parameter_post, int COUNT_params,
    typename... TYPE_parameters>
```

TypeAliasCase
When defined, the check will ensure type alias names conform to the selected casing.

TypeAliasPrefix
When defined, the check will ensure type alias names will add the prefixed with the given value (regardless of casing).

TypeAliasIgnoredRegexp
Identifier naming checks won’t be enforced for type alias names matching this regular expression.

TypeAliasSuffix
When defined, the check will ensure type alias names will add the suffix with the given value (regardless of casing).

For example using values of:

- TypeAliasCase of **lower_case**
- TypeAliasPrefix of **pre_**
- TypeAliasSuffix of **_post**

Identifies and/or transforms type alias names as follows:

Before:

```
using MY_STRUCT_TYPE = my_structure;
```

After:

```
using pre_my_struct_type_post = my_structure;
```

TypedefCase
When defined, the check will ensure typedef names conform to the selected casing.

TypedefPrefix
When defined, the check will ensure typedef names will add the prefixed with the given value (regardless of casing).

TypedefIgnoredRegexp
Identifier naming checks won't be enforced for typedef names matching this regular expression.

TypedefSuffix
When defined, the check will ensure typedef names will add the suffix with the given value (regardless of casing).

For example using values of:

- TypedefCase of **lower_case**
- TypedefPrefix of **pre_**
- TypedefSuffix of **_post**

Identifies and/or transforms typedef names as follows:

Before:

```
typedef int MYINT;
```

After:

typedef int pre_myint_post;

TypeTemplateParameterCase
When defined, the check will ensure type template parameter names conform to the selected casing.

TypeTemplateParameterPrefix
When defined, the check will ensure type template parameter names will add the prefixed with the given value (regardless of casing).

TypeTemplateParameterIgnoredRegexp
Identifier naming checks won't be enforced for type template names matching this regular expression.

TypeTemplateParameterSuffix
When defined, the check will ensure type template parameter names will add the suffix with the given value (regardless of casing).

For example using values of:

- TypeTemplateParameterCase of **lower_case**
- TypeTemplateParameterPrefix of **pre_**
- TypeTemplateParameterSuffix of **_post**

Identifies and/or transforms type template parameter names as follows:

Before:

```
template <template <typename> class TPL_parameter, int COUNT_params,
    typename... TYPE_parameters>
```

After:

```
template <template <typename> class TPL_parameter, int COUNT_params,
    typename... pre_type_parameters_post>
```

UnionCase
When defined, the check will ensure union names conform to the selected casing.

UnionPrefix
When defined, the check will ensure union names will add the prefixed with the given value (regardless of casing).

UnionIgnoredRegexp
Identifier naming checks won't be enforced for union names matching this regular expression.

UnionSuffix
When defined, the check will ensure union names will add the suffix with the given value (regardless of casing).

For example using values of:

- UnionCase of **lower_case**
- UnionPrefix of **pre_**
- UnionSuffix of **_post**

Identifies and/or transforms union names as follows:

Before:

```
union FOO {
  int a;
  char b;
};
```

After:

```
union pre_foo_post {
  int a;
  char b;
};
```

ValueTemplateParameterCase
When defined, the check will ensure value template parameter names conform to the selected casing.

ValueTemplateParameterPrefix

When defined, the check will ensure value template parameter names will add the prefixed with the given value (regardless of casing).

ValueTemplateParameterIgnoredRegexp
Identifier naming checks won’t be enforced for value template parameter names matching this regular expression.

ValueTemplateParameterSuffix
When defined, the check will ensure value template parameter names will add the suffix with the given value (regardless of casing).

For example using values of:

- ValueTemplateParameterCase of **lower_case**
- ValueTemplateParameterPrefix of **pre_**
- ValueTemplateParameterSuffix of **_post**

Identifies and/or transforms value template parameter names as follows:

Before:

```
template <template <typename> class TPL_parameter, int COUNT_params,
    typename... TYPE_parameters>
```

After:

```
template <template <typename> class TPL_parameter, int pre_count_params_post,
    typename... TYPE_parameters>
```

VariableCase
When defined, the check will ensure variable names conform to the selected casing.

VariablePrefix
When defined, the check will ensure variable names will add the prefixed with the given value (regardless of casing).

VariableIgnoredRegexp
Identifier naming checks won’t be enforced for variable names matching this regular expression.

VariableSuffix
When defined, the check will ensure variable names will add the suffix with the given value (regardless of casing).

VariableHungarianPrefix
When enabled, the check ensures that the declared identifier will have a Hungarian notation prefix based on the declared type.

For example using values of:

- VariableCase of **lower_case**
- VariablePrefix of **pre_**
- VariableSuffix of **_post**
- VariableHungarianPrefix of **On**

Identifies and/or transforms variable names as follows:

Before:

unsigned MyVariable;

After:

unsigned pre_myvariable_post;

VirtualMethodCase
When defined, the check will ensure virtual method names conform to the selected casing.

VirtualMethodPrefix
When defined, the check will ensure virtual method names will add the prefixed with the given value (regardless of casing).

VirtualMethodIgnoredRegexp
Identifier naming checks won’t be enforced for virtual method names matching this regular expression.

VirtualMethodSuffix

When defined, the check will ensure virtual method names will add the suffix with the given value (regardless of casing).

For example using values of:

- VirtualMethodCase of **lower_case**
- VirtualMethodPrefix of **pre_**
- VirtualMethodSuffix of **_post**

Identifies and/or transforms virtual method names as follows:

Before:

```
class Foo {
public:
  virtual int MemberFunction();
}
```

After:

```
class Foo {
public:
  virtual int pre_member_function_post();
}
```

The default mapping table of Hungarian Notation

In Hungarian notation, a variable name starts with a group of lower-case letters which are mnemonics for the type or purpose of that variable, followed by whatever name the programmer has chosen; this last part is sometimes distinguished as the given name. The first character of the given name can be capitalized to separate it from the type indicators (see also CamelCase). Otherwise the case of this character denotes scope.

The following table is the default mapping table of Hungarian Notation which maps Decl to its prefix string. You can also have your own style in config file.

Primitive Types	Microsoft data types				

Type	Prefix	Type	Prefix	Type	Prefix
int8_t	i8	signed int	si	BOOL	b
int16_t	i16	signed short	ss	BOOLEAN	b
int32_t	i32	signed short int	ssi	BYTE	by
int64_t	i64	signed long long int	slli	CHAR	c
uint8_t	u8	signed long long	sll	UCHAR	uc
uint16_t	u16	signed long int	sli	SHORT	s
uint32_t	u32	signed long	sl	USHORT	us
uint64_t	u64	signed	s	WORD	w
char8_t	c8	unsigned long long int	ulli	DWORD	dw
char16_t	c16	unsigned long long	ull	DWORD32	dw32
char32_t	c32	unsigned long int	uli	DWORD64	dw64
float	f	unsigned long	ul	LONG	l
double	d	unsigned short int	usi	ULONG	ul

char	c	unsigned short	us	ULONG32	ul32
bool	b	unsigned int	ui	ULONG64	ul64
_Bool	b	unsigned	u	ULONGLONG	ull
int	i	long long int	lli	HANDLE	h
size_t	n	long double	ld	INT	i
short	s	long long	ll	INT8	i8
signed	i	long int	li	INT16	i16
unsigned	u	long	l	INT32	i32
long	l	ptrdiff_t	p	INT64	i64
long long	ll			UINT	ui
unsigned long	ul			UINT8	u8
long double	ld			UINT16	u16
ptrdiff_t	p			UINT32	u32
wchar_t	wc			UINT64	u64
short int	si			PVOID	p

```
+------------------+-----------------+-------------------+---------+-------------------+---------+
|short             |s                |                   |         |                   |         |
+------------------+-----------------+-------------------+---------+-------------------+---------+
```

There are more trivial options for Hungarian Notation:

HungarianNotation.General.*
Options are not belonging to any specific Decl.

HungarianNotation.CString.*
Options for NULL-terminated string.

HungarianNotation.DerivedType.*
Options for derived types.

HungarianNotation.PrimitiveType.*
Options for primitive types.

HungarianNotation.UserDefinedType.*
Options for user-defined types.

Options for Hungarian Notation

- *HungarianNotation.General.TreatStructAsClass*
- *HungarianNotation.DerivedType.Array*
- *HungarianNotation.DerivedType.Pointer*
- *HungarianNotation.DerivedType.FunctionPointer*
- *HungarianNotation.CString.CharPrinter*
- *HungarianNotation.CString.CharArray*
- *HungarianNotation.CString.WideCharPrinter*
- *HungarianNotation.CString.WideCharArray*
- *HungarianNotation.PrimitiveType.**

- *HungarianNotation.UserDefinedType.**

HungarianNotation.General.TreatStructAsClass
When defined, the check will treat naming of struct as a class. The default value is *false*.

HungarianNotation.DerivedType.Array
When defined, the check will ensure variable name will add the prefix with the given string. The default prefix is *a*.

HungarianNotation.DerivedType.Pointer
When defined, the check will ensure variable name will add the prefix with the given string. The default prefix is *p*.

HungarianNotation.DerivedType.FunctionPointer
When defined, the check will ensure variable name will add the prefix with the given string. The default prefix is *fn*.

Before:

```
// Array
int DataArray[2] = {0};

// Pointer
void *DataBuffer = NULL;

// FunctionPointer
typedef void (*FUNC_PTR)();
FUNC_PTR FuncPtr = NULL;
```

After:

```
// Array
int aDataArray[2] = {0};

// Pointer
void *pDataBuffer = NULL;

// FunctionPointer
typedef void (*FUNC_PTR)();
FUNC_PTR fnFuncPtr = NULL;
```

HungarianNotation.CString.CharPrinter
When defined, the check will ensure variable name will add the prefix with the given string. The default prefix is *sz*.

HungarianNotation.CString.CharArray
When defined, the check will ensure variable name will add the prefix with the given string. The default prefix is *sz*.

HungarianNotation.CString.WideCharPrinter
When defined, the check will ensure variable name will add the prefix with the given string. The default prefix is *wsz*.

HungarianNotation.CString.WideCharArray
When defined, the check will ensure variable name will add the prefix with the given string. The default prefix is *wsz*.

Before:

```
// CharPrinter
const char *NamePtr = "Name";

// CharArray
const char NameArray[] = "Name";

// WideCharPrinter
const wchar_t *WideNamePtr = L"Name";

// WideCharArray
const wchar_t WideNameArray[] = L"Name";
```

After:

```
// CharPrinter
const char *szNamePtr = "Name";

// CharArray
const char szNameArray[] = "Name";

// WideCharPrinter
const wchar_t *wszWideNamePtr = L"Name";
```

```
// WideCharArray
const wchar_t wszWideNameArray[] = L"Name";
```

HungarianNotation.PrimitiveType.*

When defined, the check will ensure variable name of involved primitive types will add the prefix with the given string. The default prefixes are defined in the default mapping table.

HungarianNotation.UserDefinedType.*

When defined, the check will ensure variable name of involved primitive types will add the prefix with the given string. The default prefixes are defined in the default mapping table.

Before:

```
int8_t   ValueI8      = 0;
int16_t  ValueI16     = 0;
int32_t  ValueI32     = 0;
int64_t  ValueI64     = 0;
uint8_t  ValueU8      = 0;
uint16_t ValueU16     = 0;
uint32_t ValueU32     = 0;
uint64_t ValueU64     = 0;
float    ValueFloat   = 0.0;
double   ValueDouble  = 0.0;
ULONG    ValueUlong   = 0;
DWORD    ValueDword   = 0;
```

After:

```
int8_t   i8ValueI8    = 0;
int16_t  i16ValueI16  = 0;
int32_t  i32ValueI32  = 0;
int64_t  i64ValueI64  = 0;
uint8_t  u8ValueU8    = 0;
uint16_t u16ValueU16  = 0;
uint32_t u32ValueU32  = 0;
uint64_t u64ValueU64  = 0;
float    fValueFloat  = 0.0;
double   dValueDouble = 0.0;
ULONG    ulValueUlong = 0;
DWORD    dwValueDword = 0;
```

readability-implicit-bool-cast

This check has been renamed to *readability-implicit-bool-conversion*.

readability-implicit-bool-conversion

This check can be used to find implicit conversions between built-in types and booleans. Depending on use case, it may simply help with readability of the code, or in some cases, point to potential bugs which remain unnoticed due to implicit conversions.

The following is a real-world example of bug which was hiding behind implicit **bool** conversion:

```
class Foo {
  int m_foo;

public:
  void setFoo(bool foo) { m_foo = foo; } // warning: implicit conversion bool -> int
  int getFoo() { return m_foo; }
};

void use(Foo& foo) {
  bool value = foo.getFoo(); // warning: implicit conversion int -> bool
}
```

This code is the result of unsuccessful refactoring, where type of **m_foo** changed from **bool** to **int**. The programmer forgot to change all occurrences of **bool**, and the remaining code is no longer correct, yet it still compiles without any visible warnings.

In addition to issuing warnings, fix-it hints are provided to help solve the reported issues. This can be used for improving readability of code, for example:

```
void conversionsToBool() {
  float floating;
  bool boolean = floating;
  // ^ propose replacement: bool boolean = floating != 0.0f;

  int integer;
  if (integer) {}
  // ^ propose replacement: if (integer != 0) {}

  int* pointer;
  if (!pointer) {}
```

```
  // ^ propose replacement: if (pointer == nullptr) {}

  while (1) {}
  // ^ propose replacement: while (true) {}
}

void functionTakingInt(int param);

void conversionsFromBool() {
  bool boolean;
  functionTakingInt(boolean);
  // ^ propose replacement: functionTakingInt(static_cast<int>(boolean));

  functionTakingInt(true);
  // ^ propose replacement: functionTakingInt(1);
}
```

In general, the following conversion types are checked:

- integer expression/literal to boolean (conversion from a single bit bitfield to boolean is explicitly allowed, since there's no ambiguity / information loss in this case),
- floating expression/literal to boolean,
- pointer/pointer to member/**nullptr**/**NULL** to boolean,
- boolean expression/literal to integer (conversion from boolean to a single bit bitfield is explicitly allowed),
- boolean expression/literal to floating.

The rules for generating fix-it hints are:

- in case of conversions from other built-in type to bool, an explicit comparison is proposed to make it clear what exactly is being compared:
  - **bool boolean = floating;** is changed to **bool boolean = floating == 0.0f;**,
  - for other types, appropriate literals are used (**0**, **0u**, **0.0f**, **0.0**, **nullptr**),

- in case of negated expressions conversion to bool, the proposed replacement with comparison is simplified:
  - **if (!pointer)** is changed to **if (pointer == nullptr)**,
- in case of conversions from bool to other built-in types, an explicit **static_cast** is proposed to make it clear that a conversion is taking place:
  - **int integer = boolean;** is changed to **int integer = static_cast<int>(boolean);**,
- if the conversion is performed on type literals, an equivalent literal is proposed, according to what type is actually expected, for example:
  - **functionTakingBool(0);** is changed to **functionTakingBool(false);**,
  - **functionTakingInt(true);** is changed to **functionTakingInt(1);**,
  - for other types, appropriate literals are used (**false**, **true**, **0**, **1**, **0u**, **1u**, **0.0f**, **1.0f**, **0.0**, **1.0f**).

    Some additional accommodations are made for pre-C++11 dialects:

- **false** literal conversion to pointer is detected,
- instead of **nullptr** literal, **0** is proposed as replacement.

    Occurrences of implicit conversions inside macros and template instantiations are deliberately ignored, as it is not clear how to deal with such cases.

Options

AllowIntegerConditions
When *true*, the check will allow conditional integer conversions. Default is *false*.

AllowPointerConditions
When *true*, the check will allow conditional pointer conversions. Default is *false*.

readability-inconsistent-declaration-parameter-name
Find function declarations which differ in parameter names.

Example:

```
// in foo.hpp:
void foo(int a, int b, int c);

// in foo.cpp:
void foo(int d, int e, int f); // warning
```

This check should help to enforce consistency in large projects, where it often happens that a definition of function is refactored, changing the parameter names, but its declaration in header file is not updated. With this check, we can easily find and correct such inconsistencies, keeping declaration and definition always in sync.

Unnamed parameters are allowed and are not taken into account when comparing function declarations, for example:

```
void foo(int a);
void foo(int); // no warning
```

One name is also allowed to be a case-insensitive prefix/suffix of the other:

```
void foo(int count);
void foo(int count_input) { // no warning
  int count = adjustCount(count_input);
}
```

To help with refactoring, in some cases fix-it hints are generated to align parameter names to a single naming convention. This works with the assumption that the function definition is the most up-to-date version, as it directly references parameter names in its body. Example:

```
void foo(int a); // warning and fix-it hint (replace "a" to "b")
int foo(int b) { return b + 2; } // definition with use of "b"
```

In the case of multiple redeclarations or function template specializations, a warning is issued for every redeclaration or specialization inconsistent with the definition or the first declaration seen in a translation unit.

IgnoreMacros

If this option is set to *true* (default is *true*), the check will not warn about names declared inside macros.

Strict

If this option is set to *true* (default is *false*), then names must match exactly (or be absent).

readability-isolate-declaration

Detects local variable declarations declaring more than one variable and tries to refactor the code to one statement per declaration.

The automatic code-transformation will use the same indentation as the original for every created statement and add a line break after each statement. It keeps the order of the variable declarations consistent, too.

```
void f() {
  int * pointer = nullptr, value = 42, * const const_ptr = &value;
  // This declaration will be diagnosed and transformed into:
  // int * pointer = nullptr;
  // int value = 42;
  // int * const const_ptr = &value;
}
```

The check excludes places where it is necessary or common to declare multiple variables in one statement and there is no other way supported in the language. Please note that structured bindings are not considered.

```
// It is not possible to transform this declaration and doing the declaration
// before the loop will increase the scope of the variable 'Begin' and 'End'
// which is undesirable.
for (int Begin = 0, End = 100; Begin < End; ++Begin);
if (int Begin = 42, Result = some_function(Begin); Begin == Result);

// It is not possible to transform this declaration because the result is
// not functionality preserving as 'j' and 'k' would not be part of the
// 'if' statement anymore.
if (SomeCondition())
  int i = 42, j = 43, k = function(i,j);
```

Limitations

Global variables and member variables are excluded.

The check currently does not support the automatic transformation of member-pointer-types.

```
struct S {
```

```
  int a;
  const int b;
  void f() {}
};

void f() {
  // Only a diagnostic message is emitted
  int S::*p = &S::a, S::*const q = &S::a;
}
```

Furthermore, the transformation is very cautious when it detects various kinds of macros or preprocessor directives in the range of the statement. In this case the transformation will not happen to avoid unexpected side-effects due to macros.

```
#define NULL 0
#define MY_NICE_TYPE int **
#define VAR_NAME(name) name##__LINE__
#define A_BUNCH_OF_VARIABLES int m1 = 42, m2 = 43, m3 = 44;

void macros() {
  int *p1 = NULL, *p2 = NULL;
  // Will be transformed to
  // int *p1 = NULL;
  // int *p2 = NULL;

  MY_NICE_TYPE p3, v1, v2;
  // Won't be transformed, but a diagnostic is emitted.

  int VAR_NAME(v3),
      VAR_NAME(v4),
      VAR_NAME(v5);
  // Won't be transformed, but a diagnostic is emitted.

  A_BUNCH_OF_VARIABLES
  // Won't be transformed, but a diagnostic is emitted.

  int Unconditional,
#if CONFIGURATION
      IfConfigured = 42,
#else
```

```
    IfConfigured = 0;
#endif
  // Won't be transformed, but a diagnostic is emitted.
}
```

readability-magic-numbers

Detects magic numbers, integer or floating point literals that are embedded in code and not introduced via constants or symbols.

Many coding guidelines advise replacing the magic values with symbolic constants to improve readability. Here are a few references:

- *Rule ES.45: Avoid "magic constants"; use symbolic constants in C++ Core Guidelines*
- *Rule 5.1.1 Use symbolic names instead of literal values in code in High Integrity C++*
- Item 17 in "C++ Coding Standards: 101 Rules, Guidelines and Best Practices" by Herb Sutter and Andrei Alexandrescu
- Chapter 17 in "Clean Code - A handbook of agile software craftsmanship." by Robert C. Martin
- Rule 20701 in "TRAIN REAL TIME DATA PROTOCOL Coding Rules" by Armin-Hagen Weiss, Bombardier
- *http://wiki.c2.com/?MagicNumber*

Examples of magic values:

```
double circleArea = 3.1415926535 * radius * radius;

double totalCharge = 1.08 * itemPrice;

int getAnswer() {
  return -3; // FILENOTFOUND
}

for (int mm = 1; mm <= 12; ++mm) {
  std::cout << month[mm] << '\n';
}
```

Example with magic values refactored:

```
double circleArea = M_PI * radius * radius;

const double TAX_RATE = 0.08;  // or make it variable and read from a file

double totalCharge = (1.0 + TAX_RATE) * itemPrice;

int getAnswer() {
  return E_FILE_NOT_FOUND;
}

for (int mm = 1; mm <= MONTHS_IN_A_YEAR; ++mm) {
  std::cout << month[mm] << '\n';
}
```

For integral literals by default only *0* and *1* (and *-1*) integer values are accepted without a warning. This can be overridden with the *IgnoredIntegerValues* option. Negative values are accepted if their absolute value is present in the *IgnoredIntegerValues* list.

As a special case for integral values, all powers of two can be accepted without warning by enabling the *IgnorePowersOf2IntegerValues* option.

For floating point literals by default the *0.0* floating point value is accepted without a warning. The set of ignored floating point literals can be configured using the *IgnoredFloatingPointValues* option. For each value in that set, the given string value is converted to a floating-point value representation used by the target architecture. If a floating-point literal value compares equal to one of the converted values, then that literal is not diagnosed by this check. Because floating-point equality is used to determine whether to diagnose or not, the user needs to be aware of the details of floating-point representations for any values that cannot be precisely represented for their target architecture.

For each value in the *IgnoredFloatingPointValues* set, both the single-precision form and double-precision form are accepted (for example, if 3.14 is in the set, neither 3.14f nor 3.14 will produce a warning).

Scientific notation is supported for both source code input and option. Alternatively, the check for the floating point numbers can be disabled for all floating point values by enabling the *IgnoreAllFloatingPointValues* option.

Since values *0* and *0.0* are so common as the base counter of loops, or initialization values for sums, they are always accepted without warning, even if not present in the respective ignored values list.

Options

IgnoredIntegerValues
Semicolon-separated list of magic positive integers that will be accepted without a warning. Default values are *{1, 2, 3, 4}*, and *0* is accepted unconditionally.

IgnorePowersOf2IntegerValues
Boolean value indicating whether to accept all powers-of-two integer values without warning. Default value is *false*.

IgnoredFloatingPointValues
Semicolon-separated list of magic positive floating point values that will be accepted without a warning. Default values are *{1.0, 100.0}* and *0.0* is accepted unconditionally.

IgnoreAllFloatingPointValues
Boolean value indicating whether to accept all floating point values without warning. Default value is *false*.

IgnoreBitFieldsWidths
Boolean value indicating whether to accept magic numbers as bit field widths without warning. This is useful for example for register definitions which are generated from hardware specifications. Default value is *true*.

readability-make-member-function-const

Finds non-static member functions that can be made **const** because the functions don’t use **this** in a non-const way.

This check tries to annotate methods according to *logical constness* (not physical constness). Therefore, it will suggest to add a **const** qualifier to a non-const method only if this method does something that is already possible though the public interface on a **const** pointer to the object:

- reading a public member variable
- calling a public const-qualified member function
- returning const-qualified **this**

- passing const-qualified **this** as a parameter.

  This check will also suggest to add a **const** qualifier to a non-const method if this method uses private data and functions in a limited number of ways where logical constness and physical constness coincide:

- reading a member variable of builtin type

  Specifically, this check will not suggest to add a **const** to a non-const method if the method reads a private member variable of pointer type because that allows to modify the pointee which might not preserve logical constness. For the same reason, it does not allow to call private member functions or member functions on private member variables.

  In addition, this check ignores functions that

- are declared **virtual**

- contain a **const_cast**

- are templated or part of a class template

- have an empty body

- do not (implicitly) use **this** at all (see *readability-convert-member-functions-to-static*).

  The following real-world examples will be preserved by the check:

```
class E1 {
  Pimpl &getPimpl() const;
public:
  int &get() {
    // Calling a private member function disables this check.
    return getPimpl()->i;
  }
  ...
};

class E2 {
public:
  const int *get() const;
```

```
  // const_cast disables this check.
  S *get() {
    return const_cast<int*>(const_cast<const C*>(this)->get());
  }
  ...
};
```

After applying modifications as suggested by the check, running the check again might find more opportunities to mark member functions **const**.

readability-misleading-indentation

Correct indentation helps to understand code. Mismatch of the syntactical structure and the indentation of the code may hide serious problems. Missing braces can also make it significantly harder to read the code, therefore it is important to use braces.

The way to avoid dangling else is to always check that an **else** belongs to the **if** that begins in the same column.

You can omit braces when your inner part of e.g. an **if** statement has only one statement in it. Although in that case you should begin the next statement in the same column with the **if**.

Examples:

```
// Dangling else:
if (cond1)
  if (cond2)
    foo1();
else
  foo2();  // Wrong indentation: else belongs to if(cond2) statement.

// Missing braces:
if (cond1)
  foo1();
  foo2();  // Not guarded by if(cond1).
```

Limitations

Note that this check only works as expected when the tabs or spaces are used consistently and not mixed.

readability-misplaced-array-index

This check warns for unusual array index syntax.

The following code has unusual array index syntax:

```
void f(int *X, int Y) {
  Y[X] = 0;
}
```

becomes

```
void f(int *X, int Y) {
  X[Y] = 0;
}
```

The check warns about such unusual syntax for readability reasons:

- There are programmers that are not familiar with this unusual syntax.
- It is possible that variables are mixed up.

readability-named-parameter

Find functions with unnamed arguments.

The check implements the following rule originating in the Google C++ Style Guide:

https://google.github.io/styleguide/cppguide.html#Function_Declarations_and_Definitions

All parameters should be named, with identical names in the declaration and implementation.

Corresponding cpplint.py check name: *readability/function*.

readability-non-const-parameter

The check finds function parameters of a pointer type that could be changed to point to a constant type instead.

When **const** is used properly, many mistakes can be avoided. Advantages when using **const** properly:

- prevent unintentional modification of data;
- get additional warnings such as using uninitialized data;

- make it easier for developers to see possible side effects.

  This check is not strict about constness, it only warns when the constness will make the function interface safer.

```
// warning here; the declaration "const char *p" would make the function
// interface safer.
char f1(char *p) {
  return *p;
}

// no warning; the declaration could be more const "const int * const p" but
// that does not make the function interface safer.
int f2(const int *p) {
  return *p;
}

// no warning; making x const does not make the function interface safer
int f3(int x) {
  return x;
}

// no warning; Technically, *p can be const ("const struct S *p"). But making
// *p const could be misleading. People might think that it’s safe to pass
// const data to this function.
struct S { int *a; int *b; };
int f3(struct S *p) {
  *(p->a) = 0;
}

// no warning; p is referenced by an lvalue.
void f4(int *p) {
  int &x = *p;
}
```

readability-qualified-auto

Adds pointer qualifications to **auto**-typed variables that are deduced to pointers.

LLVM Coding Standards advises to make it obvious if a **auto** typed variable is a pointer. This check will transform **auto** to **auto *** when the type is deduced to be a pointer.

```
for (auto Data : MutatablePtrContainer) {
  change(*Data);
}
for (auto Data : ConstantPtrContainer) {
  observe(*Data);
}
```

Would be transformed into:

```
for (auto *Data : MutatablePtrContainer) {
  change(*Data);
}
for (const auto *Data : ConstantPtrContainer) {
  observe(*Data);
}
```

Note **const volatile** qualified types will retain their **const** and **volatile** qualifiers. Pointers to pointers will not be fully qualified.

```
const auto Foo = cast<int *>(Baz1);
const auto Bar = cast<const int *>(Baz2);
volatile auto FooBar = cast<int *>(Baz3);
auto BarFoo = cast<int **>(Baz4);
```

Would be transformed into:

```
auto *const Foo = cast<int *>(Baz1);
const auto *const Bar = cast<const int *>(Baz2);
auto *volatile FooBar = cast<int *>(Baz3);
auto *BarFoo = cast<int **>(Baz4);
```

Options

AddConstToQualified

When set to *true* the check will add const qualifiers variables defined as **auto *** or **auto &** when applicable. Default value is *true*.

```
auto Foo1 = cast<const int *>(Bar1);
auto *Foo2 = cast<const int *>(Bar2);
auto &Foo3 = cast<const int &>(Bar3);
```

If AddConstToQualified is set to *false*, it will be transformed into:

```
const auto *Foo1 = cast<const int *>(Bar1);
auto *Foo2 = cast<const int *>(Bar2);
auto &Foo3 = cast<const int &>(Bar3);
```

Otherwise it will be transformed into:

```
const auto *Foo1 = cast<const int *>(Bar1);
const auto *Foo2 = cast<const int *>(Bar2);
const auto &Foo3 = cast<const int &>(Bar3);
```

Note in the LLVM alias, the default value is *false*.

readability-redundant-access-specifiers

Finds classes, structs, and unions containing redundant member (field and method) access specifiers.

Example

```
class Foo {
public:
  int x;
  int y;
public:
  int z;
protected:
  int a;
public:
  int c;
}
```

In the example above, the second **public** declaration can be removed without any changes of behavior.

Options

CheckFirstDeclaration

If set to *true*, the check will also diagnose if the first access specifier declaration is redundant (e.g. **private** inside **class**, or **public** inside **struct** or **union**). Default is *false*.

Example

```
struct Bar {
public:
  int x;
}
```

If *CheckFirstDeclaration* option is enabled, a warning about redundant access specifier will be emitted, because **public** is the default member access for structs.

readability-redundant-control-flow

This check looks for procedures (functions returning no value) with **return** statements at the end of the function. Such **return** statements are redundant.

Loop statements (**for**, **while**, **do while**) are checked for redundant **continue** statements at the end of the loop body.

Examples:

The following function *f* contains a redundant **return** statement:

```
extern void g();
void f() {
  g();
  return;
}
```

becomes

```
extern void g();
void f() {
  g();
}
```

The following function *k* contains a redundant **continue** statement:

```
void k() {
  for (int i = 0; i < 10; ++i) {
    continue;
  }
```

```
}
```

becomes

```
void k() {
  for (int i = 0; i < 10; ++i) {
  }
}
```

readability-redundant-declaration

Finds redundant variable and function declarations.

```
extern int X;
extern int X;
```

becomes

```
extern int X;
```

Such redundant declarations can be removed without changing program behavior. They can for instance be unintentional left overs from previous refactorings when code has been moved around. Having redundant declarations could in worst case mean that there are typos in the code that cause bugs.

Normally the code can be automatically fixed, **clang-tidy** can remove the second declaration. However there are 2 cases when you need to fix the code manually:

- When the declarations are in different header files;
- When multiple variables are declared together.

Options

IgnoreMacros

If set to *true*, the check will not give warnings inside macros. Default is *true*.

readability-redundant-function-ptr-dereference

Finds redundant dereferences of a function pointer.

Before:

```
int f(int,int);
int (*p)(int, int) = &f;

int i = (**p)(10, 50);
```

After:

```
int f(int,int);
int (*p)(int, int) = &f;

int i = (*p)(10, 50);
```

readability-redundant-member-init

Finds member initializations that are unnecessary because the same default constructor would be called if they were not present.

Example

```
// Explicitly initializing the member s is unnecessary.
class Foo {
public:
  Foo() : s() {}

private:
  std::string s;
};
```

Options

IgnoreBaseInCopyConstructors

Default is *false*.

When *true*, the check will ignore unnecessary base class initializations within copy constructors, since some compilers issue warnings/errors when base classes are not explicitly initialized in copy constructors. For example, **gcc** with **-Wextra** or **-Werror=extra** issues warning or error **base class 'Bar' should be explicitly initialized in the copy constructor** if **Bar()** were removed in the following example:

```
// Explicitly initializing member s and base class Bar is unnecessary.
struct Foo : public Bar {
```

```
  // Remove s() below. If IgnoreBaseInCopyConstructors!=0, keep Bar().
  Foo(const Foo& foo) : Bar(), s() {}
  std::string s;
};
```

readability-redundant-preprocessor

Finds potentially redundant preprocessor directives. At the moment the following cases are detected:

- *#ifdef* .. *#endif* pairs which are nested inside an outer pair with the same condition. For example:

```
#ifdef FOO
#ifdef FOO // inner ifdef is considered redundant
void f();
#endif
#endif
```

- Same for *#ifndef* .. *#endif* pairs. For example:

```
#ifndef FOO
#ifndef FOO // inner ifndef is considered redundant
void f();
#endif
#endif
```

- *#ifndef* inside an *#ifdef* with the same condition:

```
#ifdef FOO
#ifndef FOO // inner ifndef is considered redundant
void f();
#endif
#endif
```

- *#ifdef* inside an *#ifndef* with the same condition:

```
#ifndef FOO
#ifdef FOO // inner ifdef is considered redundant
void f();
#endif
#endif
```

- *#if .. #endif* pairs which are nested inside an outer pair with the same condition. For example:

```
#define FOO 4
#if FOO == 4
#if FOO == 4 // inner if is considered redundant
void f();
#endif
#endif
```

readability-redundant-smartptr-get

Find and remove redundant calls to smart pointer's **.get()** method.

Examples:

```
ptr.get()->Foo()  ==>  ptr->Foo()
*ptr.get()  ==>  *ptr
*ptr->get()  ==>  **ptr
if (ptr.get() == nullptr) ... => if (ptr == nullptr) ...
```

IgnoreMacros

If this option is set to *true* (default is *true*), the check will not warn about calls inside macros.

readability-redundant-string-cstr

Finds unnecessary calls to **std::string::c_str()** and **std::string::data()**.

readability-redundant-string-init

Finds unnecessary string initializations.

Examples

```
// Initializing string with empty string literal is unnecessary.
std::string a = "";
std::string b("");

// becomes

std::string a;
std::string b;

// Initializing a string_view with an empty string literal produces an
```

```
// instance that compares equal to string_view().
std::string_view a = "";
std::string_view b("");

// becomes
std::string_view a;
std::string_view b;
```

Options

StringNames

Default is *::std::basic_string;::std::basic_string_view*.

Semicolon-delimited list of class names to apply this check to. By default *::std::basic_string* applies to **std::string** and **std::wstring**. Set to e.g. *::std::basic_string;llvm::StringRef;QString* to perform this check on custom classes.

readability-simplify-boolean-expr

Looks for boolean expressions involving boolean constants and simplifies them to use the appropriate boolean expression directly. Simplifies boolean expressions by application of DeMorgan's Theorem.

Examples:

Initial expression	Result
if (b == true)	**if (b)**
if (b == false)	**if (!b)**
if (b && true)	**if (b)**
if (b && false)	**if (false)**
if (b ||	**if**

true)	**(true)**
if (b || false)	**if (b)**
e ? true : false	**e**
e ? false : true	**!e**
if (true) t(); else f();	**t();**
if (false) t(); else f();	**f();**
if (e) return true; else return false;	**return e;**
if (e) return false; else return true;	**return !e;**
if (e) b = true; else b = false;	**b = e;**
if (e) b = false; else b = true;	**b = !e;**
if (e) return true; return false;	**return e;**
if (e) return false; return true;	**return !e;**
!(!a || b)	**a && !b**
!(a || !b)	**!a && b**

!(!a || !b)	**a && b**
!(!a && b)	**a || !b**
!(a && !b)	**!a || b**
!(!a && b)	**a || b**

The resulting expression e is modified as follows:

1. Unnecessary parentheses around the expression are removed.

2. Negated applications of **!** are eliminated.

3. Negated applications of comparison operators are changed to use the opposite condition.

4. Implicit conversions of pointers, including pointers to members, to **bool** are replaced with explicit comparisons to **nullptr** in C++11 or **NULL** in C++98/03.

5. Implicit casts to **bool** are replaced with explicit casts to **bool**.

6. Object expressions with **explicit operator bool** conversion operators are replaced with explicit casts to **bool**.

7. Implicit conversions of integral types to **bool** are replaced with explicit comparisons to **0**.

Examples:

1. The ternary assignment **bool b = (i < 0) ? true : false;** has redundant parentheses and becomes **bool b = i < 0;**.

2. The conditional return **if (!b) return false; return true;** has an implied double negation and becomes **return b;**.

3. The conditional return **if (i < 0) return false; return true;** becomes **return i >= 0;**.

   The conditional return **if (i != 0) return false; return true;** becomes **return i == 0;**.

4. The conditional return **if (p) return true; return false;** has an implicit conversion of a pointer to **bool** and becomes **return p != nullptr;**.

   The ternary assignment **bool b = (i & 1) ? true : false;** has an implicit conversion of **i & 1** to **bool** and becomes **bool b = (i & 1) != 0;**.

5. The conditional return **if (i & 1) return true; else return false;** has an implicit conversion of an integer quantity **i & 1** to **bool** and becomes **return (i & 1) != 0;**

6. Given **struct X { explicit operator bool(); };**, and an instance **x** of **struct X**, the conditional return **if (x) return true; return false;** becomes **return static_cast<bool>(x);**

Options

ChainedConditionalReturn
If *true*, conditional boolean return statements at the end of an **if/else if** chain will be transformed. Default is *false*.

ChainedConditionalAssignment
If *true*, conditional boolean assignments at the end of an **if/else if** chain will be transformed. Default is *false*.

SimplifyDeMorgan
If *true*, DeMorgan's Theorem will be applied to simplify negated conjunctions and disjunctions. Default is *true*.

SimplifyDeMorganRelaxed
If *true*, *SimplifyDeMorgan* will also transform negated conjunctions and disjunctions where there is no negation on either operand. This option has no effect if *SimplifyDeMorgan* is *false*. Default is *false*.

When Enabled:

```
bool X = !(A && B)
bool Y = !(A || B)
```

Would be transformed to:

```
bool X = !A || !B
bool Y = !A && !B
```

readability-simplify-subscript-expr

This check simplifies subscript expressions. Currently this covers calling **.data()** and immediately doing an array subscript operation to obtain a single element, in which case simply calling **operator[]** suffice.

Examples:

```
std::string s = ...;
char c = s.data()[i];  // char c = s[i];
```

Options

Types

The list of type(s) that triggers this check. Default is *::std::basic_string;::std::basic_string_view;::std::vector;::std::array*

readability-static-accessed-through-instance

Checks for member expressions that access static members through instances, and replaces them with uses of the appropriate qualified-id.

Example:

The following code:

```
struct C {
  static void foo();
  static int x;
};

C *c1 = new C();
c1->foo();
c1->x;
```

is changed to:

```
C *c1 = new C();
```

```
C::foo();
C::x;
```

readability-static-definition-in-anonymous-namespace

Finds static function and variable definitions in anonymous namespace.

In this case, **static** is redundant, because anonymous namespace limits the visibility of definitions to a single translation unit.

```
namespace {
  static int a = 1; // Warning.
  static const int b = 1; // Warning.
  namespace inner {
    static int c = 1; // Warning.
  }
}
```

The check will apply a fix by removing the redundant **static** qualifier.

readability-string-compare

Finds string comparisons using the compare method.

A common mistake is to use the string's **compare** method instead of using the equality or inequality operators. The compare method is intended for sorting functions and thus returns a negative number, a positive number or zero depending on the lexicographical relationship between the strings compared. If an equality or inequality check can suffice, that is recommended. This is recommended to avoid the risk of incorrect interpretation of the return value and to simplify the code. The string equality and inequality operators can also be faster than the **compare** method due to early termination.

Examples:

```
std::string str1{"a"};
std::string str2{"b"};

// use str1 != str2 instead.
if (str1.compare(str2)) {
}

// use str1 == str2 instead.
if (!str1.compare(str2)) {
```

```
}

// use str1 == str2 instead.
if (str1.compare(str2) == 0) {
}

// use str1 != str2 instead.
if (str1.compare(str2) != 0) {
}

// use str1 == str2 instead.
if (0 == str1.compare(str2)) {
}

// use str1 != str2 instead.
if (0 != str1.compare(str2)) {
}

// Use str1 == "foo" instead.
if (str1.compare("foo") == 0) {
}
```

The above code examples show the list of if-statements that this check will give a warning for. All of them uses **compare** to check if equality or inequality of two strings instead of using the correct operators.

readability-suspicious-call-argument

Finds function calls where the arguments passed are provided out of order, based on the difference between the argument name and the parameter names of the function.

Given a function call **f(foo, bar);** and a function signature **void f(T tvar, U uvar)**, the arguments **foo** and **bar** are swapped if **foo** (the argument name) is more similar to **uvar** (the other parameter) than **tvar** (the parameter it is currently passed to) **and bar** is more similar to **tvar** than **uvar**.

Warnings might indicate either that the arguments are swapped, or that the names' cross-similarity might hinder code comprehension.

Heuristics

The following heuristics are implemented in the check. If **any** of the enabled heuristics deem the arguments to be provided out of order, a warning will be issued.

The heuristics themselves are implemented by considering pairs of strings, and are symmetric, so in the following there is no distinction on which string is the argument name and which string is the parameter name.

Equality

The most trivial heuristic, which compares the two strings for case-insensitive equality.

Abbreviation

Common abbreviations can be specified which will deem the strings similar if the abbreviated and the abbreviation stand together. For example, if **src** is registered as an abbreviation for **source**, then the following code example will be warned about.

```
void foo(int source, int x);

foo(b, src);
```

The abbreviations to recognise can be configured with the *Abbreviations* check option. This heuristic is case-insensitive.

Prefix

The *prefix* heuristic reports if one of the strings is a sufficiently long prefix of the other string, e.g. **target** to **targetPtr**. The similarity percentage is the length ratio of the prefix to the longer string, in the previous example, it would be *6 / 9 = 66.66...*%.

This heuristic can be configured with *bounds*. The default bounds are: below *25*% dissimilar and above *30*% similar. This heuristic is case-insensitive.

Suffix

Analogous to the *Prefix* heuristic. In the case of **oldValue** and **value** compared, the similarity percentage is *8 / 5 = 62.5*%.

This heuristic can be configured with *bounds*. The default bounds are: below *25*% dissimilar and above *30*% similar. This heuristic is case-insensitive.

Substring

The substring heuristic combines the prefix and the suffix heuristic, and tries to find the *longest common substring* in the two strings provided. The similarity percentage is the ratio of the found longest common substring against the *longer* of the two input strings. For example, given **val** and **rvalue**, the similarity is *3 / 6 = 50*%. If no characters are common in the two string, *0*%.

This heuristic can be configured with *bounds*. The default bounds are: below *40*% dissimilar and above *50*% similar. This heuristic is case-insensitive.

Levenshtein distance (as *Levenshtein*)

The *Levenshtein distance* describes how many single-character changes (additions, changes, or removals) must be applied to transform one string into another.

The Levenshtein distance is translated into a similarity percentage by dividing it with the length of the *longer* string, and taking its complement with regards to *100*%. For example, given **something** and **anything**, the distance is *4* edits, and the similarity percentage is *100*% - *4 / 9 = 55.55*...%.

This heuristic can be configured with *bounds*. The default bounds are: below *50*% dissimilar and above *66*% similar. This heuristic is case-sensitive.

Jaro--Winkler distance (as *JaroWinkler*)

The *Jaro--Winkler distance* is an edit distance like the Levenshtein distance. It is calculated from the amount of common characters that are sufficiently close to each other in position, and to-be-changed characters. The original definition of Jaro has been extended by Winkler to weigh prefix similarities more. The similarity percentage is expressed as an average of the common and non-common characters against the length of both strings.

This heuristic can be configured with *bounds*. The default bounds are: below *75*% dissimilar and above *85*% similar. This heuristic is case-insensitive.

Sǿrensen--Dice coefficient (as *Dice*)

The *Sǿrensen--Dice coefficient* was originally defined to measure the similarity of two sets. Formally, the coefficient is calculated by dividing *2 * #(intersection)* with *#(set1) + #(set2)*, where *#()* is the cardinality function of sets. This metric is applied to strings by creating bigrams (substring sequences of length 2) of the two strings and using the set of bigrams for the two strings as the two sets.

This heuristic can be configured with *bounds*. The default bounds are: below *60*% dissimilar and above *70*% similar. This heuristic is case-insensitive.

Options

MinimumIdentifierNameLength

Sets the minimum required length the argument and parameter names need to have. Names shorter than this length will be ignored. Defaults to *3*.

Abbreviations

For the **Abbreviation** heuristic (*see here*), this option configures the abbreviations in the *"abbreviation=abbreviated_value"* format. The option is a string, with each value joined by *";"*.

By default, the following abbreviations are set:

- *addr=address*
- *arr=array*
- *attr=attribute*
- *buf=buffer*
- *cl=client*
- *cnt=count*
- *col=column*
- *cpy=copy*
- *dest=destination*
- *dist=distance*
- *dst=distance*
- *elem=element*
- *hght=height*
- *i=index*
- *idx=index*
- *len=length*
- *ln=line*
- *lst=list*

- *nr=number*
- *num=number*
- *pos=position*
- *ptr=pointer*
- *ref=reference*
- *src=source*
- *srv=server*
- *stmt=statement*
- *str=string*
- *val=value*
- *var=variable*
- *vec=vector*
- *wdth=width*

The configuration options for each implemented heuristic (see above) is constructed dynamically. In the following, <*HeuristicName*> refers to one of the keys from the heuristics implemented.

<HeuristicName>

True or *False*, whether a particular heuristic, such as *Equality* or *Levenshtein* is enabled.

Defaults to *True* for every heuristic.

<HeuristicName>DissimilarBelow, <HeuristicName>SimilarAbove

A value between *0* and *100*, expressing a percentage. The bounds set what percentage of similarity the heuristic must deduce for the two identifiers to be considered similar or dissimilar by the check.

Given arguments **arg1** and **arg2** passed to **param1** and **param2**, respectively, the bounds check is performed in the following way: If the similarity of the currently passed argument order (**arg1** to **param1**) is **below** the *DissimilarBelow* threshold, and the similarity of the suggested swapped order (**arg1** to **param2**) is **above** the *SimilarAbove* threshold, the swap is reported.

For the defaults of each heuristic, *see above*.

Name synthesis

When comparing the argument names and parameter names, the following logic is used to gather the names for comparison:

Parameter names are the identifiers as written in the source code.

Argument names are:

- If a variable is passed, the variable's name.
- If a subsequent function call's return value is used as argument, the called function's name.
- Otherwise, empty string.

Empty argument or parameter names are ignored by the heuristics.

readability-uniqueptr-delete-release

Replace **delete <unique_ptr>.release()** with **<unique_ptr> = nullptr**. The latter is shorter, simpler and does not require use of raw pointer APIs.

```
std::unique_ptr<int> P;
delete P.release();

// becomes

std::unique_ptr<int> P;
P = nullptr;
```

Options

PreferResetCall

If *true*, refactor by calling the reset member function instead of assigning to **nullptr**. Default value is *false*.

```
std::unique_ptr<int> P;
delete P.release();

// becomes

std::unique_ptr<int> P;
P.reset();
```

readability-uppercase-literal-suffix

cert-dcl16-c redirects here as an alias for this check. By default, only the suffixes that begin with **l** (**l**, **ll**, **lu**, **llu**, but not **u**, **ul**, **ull**) are diagnosed by that alias.

hicpp-uppercase-literal-suffix redirects here as an alias for this check.

Detects when the integral literal or floating point (decimal or hexadecimal) literal has a non-uppercase suffix and provides a fix-it hint with the uppercase suffix.

All valid combinations of suffixes are supported.

```
auto x = 1;  // OK, no suffix.

auto x = 1u; // warning: integer literal suffix 'u' is not upper-case

auto x = 1U; // OK, suffix is uppercase.

...
```

Options

NewSuffixes

Optionally, a list of the destination suffixes can be provided. When the suffix is found, a case-insensitive lookup in that list is made, and if a replacement is found that is different from the current suffix, then the diagnostic is issued. This allows for fine-grained control of what suffixes to consider and what their replacements should be.

Example

Given a list *L;uL*:

- **l -> L**

- **L** will be kept as is.
- **ul** -> **uL**
- **Ul** -> **uL**
- **UL** -> **uL**
- **uL** will be kept as is.
- **ull** will be kept as is, since it is not in the list
- and so on.

IgnoreMacros
If this option is set to *true* (default is *true*), the check will not warn about literal suffixes inside macros.

readability-use-anyofallof
Finds range-based for loops that can be replaced by a call to **std::any_of** or **std::all_of**. In C++ 20 mode, suggests **std::ranges::any_of** or **std::ranges::all_of**.

Example:

```
bool all_even(std::vector<int> V) {
  for (int I : V) {
    if (I % 2)
      return false;
  }
  return true;
  // Replace loop by
  // return std::ranges::all_of(V, [](int I) { return I % 2 == 0; });
}
```

zircon-temporary-objects
Warns on construction of specific temporary objects in the Zircon kernel. If the object should be flagged, If the object should be flagged, the fully qualified type name must be explicitly passed to the check.

For example, given the list of classes "Foo" and "NS::Bar", all of the following will trigger the

warning:

```
Foo();
Foo F = Foo();
func(Foo());

namespace NS {

Bar();

}
```

With the same list, the following will not trigger the warning:

```
Foo F;               // Non-temporary construction okay
Foo F(param);        // Non-temporary construction okay
Foo *F = new Foo();  // New construction okay

Bar();               // Not NS::Bar, so okay
NS::Bar B;           // Non-temporary construction okay
```

Note that objects must be explicitly specified in order to be flagged, and so objects that inherit a specified object will not be flagged.

This check matches temporary objects without regard for inheritance and so a prohibited base class type does not similarly prohibit derived class types.

```
class Derived : Foo {} // Derived is not explicitly disallowed
Derived();           // and so temporary construction is okay
```

Options

Names

A semi-colon-separated list of fully-qualified names of C++ classes that should not be constructed as temporaries. Default is empty.

Name	Offers fixes

abseil-cleanup-ctad	Yes
abseil-duration-addition	Yes
abseil-duration-comparison	Yes
abseil-duration-conversion-cast	Yes
abseil-duration-division	Yes
abseil-duration-factory-float	Yes
abseil-duration-factory-scale	Yes
abseil-duration-subtraction	Yes
abseil-duration-unnecessary-conversion	Yes
abseil-faster-strsplit-delimiter	Yes
abseil-no-internal-dependencies	
abseil-no-namespace	
abseil-redundant-strcat-calls	Yes
abseil-str-cat-append	Yes
abseil-string-find-startswith	Yes
abseil-string-find-str-contains	Yes
abseil-time-comparison	Yes
abseil-time-subtraction	Yes
abseil-upgrade-duration-conversions	Yes
altera-id-dependent-backward-branch	

altera-kernel-name-restriction	
altera-single-work-item-barrier	
altera-struct-pack-align	Yes
altera-unroll-loops	
android-cloexec-accept	Yes
android-cloexec-accept4	Yes
android-cloexec-creat	Yes
android-cloexec-dup	Yes
android-cloexec-epoll-create	Yes
android-cloexec-epoll-create1	Yes
android-cloexec-fopen	Yes
android-cloexec-inotify-init	Yes
android-cloexec-inotify-init1	Yes
android-cloexec-memfd-create	Yes
android-cloexec-open	Yes
android-cloexec-pipe	Yes
android-cloexec-pipe2	Yes
android-cloexec-socket	Yes
android-comparison-in-temp-failure-retry	
boost-use-to-string	Yes

bugprone-argument-comment	Yes
bugprone-assert-side-effect	
bugprone-assignment-in-if-condition	
bugprone-bad-signal-to-kill-thread	
bugprone-bool-pointer-implicit-conversion	Yes
bugprone-branch-clone	
bugprone-copy-constructor-init	Yes
bugprone-dangling-handle	
bugprone-dynamic-static-initializers	
bugprone-easily-swappable-parameters	
bugprone-exception-escape	
bugprone-fold-init-type	
bugprone-forward-declaration-namespace	
bugprone-forwarding-reference-overload	
bugprone-implicit-widening-of-multiplication-result	Yes
bugprone-inaccurate-erase	Yes
bugprone-incorrect-roundings	
bugprone-infinite-loop	
bugprone-integer-division	
bugprone-lambda-function-name	

bugprone-macro-parentheses	Yes
bugprone-macro-repeated-side-effects	
bugprone-misplaced-operator-in-strlen-in-alloc	Yes
bugprone-misplaced-pointer-arithmetic-in-alloc	Yes
bugprone-misplaced-widening-cast	
bugprone-move-forwarding-reference	Yes
bugprone-multiple-statement-macro	
bugprone-no-escape	
bugprone-not-null-terminated-result	Yes
bugprone-parent-virtual-call	Yes
bugprone-posix-return	Yes
bugprone-redundant-branch-condition	Yes
bugprone-reserved-identifier	Yes
bugprone-shared-ptr-array-mismatch	Yes
bugprone-signal-handler	
bugprone-signed-char-misuse	
bugprone-sizeof-container	
bugprone-sizeof-expression	
bugprone-spuriously-wake-up-functions	
bugprone-string-constructor	Yes

bugprone-string-integer-assignment	Yes
bugprone-string-literal-with-embedded-nul	
bugprone-stringview-nullptr	Yes
bugprone-suspicious-enum-usage	
bugprone-suspicious-include	
bugprone-suspicious-memory-comparison	
bugprone-suspicious-memset-usage	Yes
bugprone-suspicious-missing-comma	
bugprone-suspicious-semicolon	Yes
bugprone-suspicious-string-compare	Yes
bugprone-swapped-arguments	Yes
bugprone-terminating-continue	Yes
bugprone-throw-keyword-missing	
bugprone-too-small-loop-variable	
bugprone-unchecked-optional-access	
bugprone-undefined-memory-manipulation	
bugprone-undelegated-constructor	
bugprone-unhandled-exception-at-new	
bugprone-unhandled-self-assignment	
bugprone-unused-raii	Yes

bugprone-unused-return-value	
bugprone-use-after-move	
bugprone-virtual-near-miss	Yes
cert-dcl21-cpp	Yes
cert-dcl50-cpp	
cert-dcl58-cpp	
cert-env33-c	
cert-err33-c	
cert-err34-c	
cert-err52-cpp	
cert-err58-cpp	
cert-err60-cpp	
cert-flp30-c	
cert-mem57-cpp	
cert-msc50-cpp	
cert-msc51-cpp	
cert-oop57-cpp	
cert-oop58-cpp	
clang-analyzer-core.DynamicTypePropagation	
clang-analyzer-core.uninitialized.CapturedBlockVariable	

clang-analyzer-cplusplus.InnerPointer	
clang-analyzer-nullability.NullableReturnedFromNonnull	
clang-analyzer-optin.osx.OSObjectCStyleCast	
clang-analyzer-optin.performance.GCDAntipattern	
clang-analyzer-optin.performance.Padding	
clang-analyzer-optin.portability.UnixAPI	
clang-analyzer-osx.MIG	
clang-analyzer-osx.NumberObjectConversion	
clang-analyzer-osx.OSObjectRetainCount	
clang-analyzer-osx.ObjCProperty	
clang-analyzer-osx.cocoa.AutoreleaseWrite	
clang-analyzer-osx.cocoa.Loops	
clang-analyzer-osx.cocoa.MissingSuperCall	
clang-analyzer-osx.cocoa.NonNilReturnValue	
clang-analyzer-osx.cocoa.RunLoopAutoreleaseLeak	
clang-analyzer-valist.CopyToSelf	
clang-analyzer-valist.Uninitialized	
clang-analyzer-valist.Unterminated	
concurrency-mt-unsafe	
concurrency-thread-canceltype-asynchronous	

cppcoreguidelines-avoid-goto	
cppcoreguidelines-avoid-non-const-global-variables	
cppcoreguidelines-init-variables	Yes
cppcoreguidelines-interfaces-global-init	
cppcoreguidelines-macro-usage	
cppcoreguidelines-narrowing-conversions	
cppcoreguidelines-no-malloc	
cppcoreguidelines-owning-memory	
cppcoreguidelines-prefer-member-initializer	Yes
cppcoreguidelines-pro-bounds-array-to-pointer-decay	
cppcoreguidelines-pro-bounds-constant-array-index	Yes
cppcoreguidelines-pro-bounds-pointer-arithmetic	
cppcoreguidelines-pro-type-const-cast	
cppcoreguidelines-pro-type-cstyle-cast	Yes
cppcoreguidelines-pro-type-member-init	Yes
cppcoreguidelines-pro-type-reinterpret-cast	
cppcoreguidelines-pro-type-static-cast-downcast	Yes
cppcoreguidelines-pro-type-union-access	
cppcoreguidelines-pro-type-vararg	
cppcoreguidelines-slicing	

cppcoreguidelines-special-member-functions	
cppcoreguidelines-virtual-class-destructor	Yes
darwin-avoid-spinlock	
darwin-dispatch-once-nonstatic	Yes
fuchsia-default-arguments-calls	
fuchsia-default-arguments-declarations	Yes
fuchsia-multiple-inheritance	
fuchsia-overloaded-operator	
fuchsia-statically-constructed-objects	
fuchsia-trailing-return	
fuchsia-virtual-inheritance	
google-build-explicit-make-pair	
google-build-namespaces	
google-build-using-namespace	
google-default-arguments	
google-explicit-constructor	Yes
google-global-names-in-headers	
google-objc-avoid-nsobject-new	
google-objc-avoid-throwing-exception	
google-objc-function-naming	

google-objc-global-variable-declaration	
google-readability-avoid-underscore-in-googletest-name	
google-readability-casting	
google-readability-todo	
google-runtime-int	
google-runtime-operator	
google-upgrade-googletest-case	Yes
hicpp-avoid-goto	
hicpp-exception-baseclass	
hicpp-multiway-paths-covered	
hicpp-no-assembler	
hicpp-signed-bitwise	
linuxkernel-must-use-errs	
llvm-header-guard	
llvm-include-order	Yes
llvm-namespace-comment	
llvm-prefer-isa-or-dyn-cast-in-conditionals	Yes
llvm-prefer-register-over-unsigned	Yes
llvm-twine-local	Yes
llvmlibc-callee-namespace	

llvmlibc-implementation-in-namespace	
llvmlibc-restrict-system-libc-headers	Yes
misc-confusable-identifiers	
misc-const-correctness	Yes
misc-definitions-in-headers	Yes
misc-misleading-bidirectional	
misc-misleading-identifier	
misc-misplaced-const	
misc-new-delete-overloads	
misc-no-recursion	
misc-non-copyable-objects	
misc-non-private-member-variables-in-classes	
misc-redundant-expression	Yes
misc-static-assert	Yes
misc-throw-by-value-catch-by-reference	
misc-unconventional-assign-operator	
misc-uniqueptr-reset-release	Yes
misc-unused-alias-decls	Yes
misc-unused-parameters	Yes
misc-unused-using-decls	Yes

modernize-avoid-bind	Yes
modernize-avoid-c-arrays	
modernize-concat-nested-namespaces	Yes
modernize-deprecated-headers	Yes
modernize-deprecated-ios-base-aliases	Yes
modernize-loop-convert	Yes
modernize-macro-to-enum	Yes
modernize-make-shared	Yes
modernize-make-unique	Yes
modernize-pass-by-value	Yes
modernize-raw-string-literal	Yes
modernize-redundant-void-arg	Yes
modernize-replace-auto-ptr	Yes
modernize-replace-disallow-copy-and-assign-macro	Yes
modernize-replace-random-shuffle	Yes
modernize-return-braced-init-list	Yes
modernize-shrink-to-fit	Yes
modernize-unary-static-assert	Yes
modernize-use-auto	Yes
modernize-use-bool-literals	Yes

modernize-use-default-member-init	Yes
modernize-use-emplace	Yes
modernize-use-equals-default	Yes
modernize-use-equals-delete	Yes
modernize-use-nodiscard	Yes
modernize-use-noexcept	Yes
modernize-use-nullptr	Yes
modernize-use-override	Yes
modernize-use-trailing-return-type	Yes
modernize-use-transparent-functors	Yes
modernize-use-uncaught-exceptions	Yes
modernize-use-using	Yes
mpi-buffer-deref	Yes
mpi-type-mismatch	Yes
objc-assert-equals	Yes
objc-avoid-nserror-init	
objc-dealloc-in-category	
objc-forbidden-subclassing	
objc-missing-hash	
objc-nsinvocation-argument-lifetime	Yes

objc-property-declaration	Yes
objc-super-self	Yes
openmp-exception-escape	
openmp-use-default-none	
performance-faster-string-find	Yes
performance-for-range-copy	Yes
performance-implicit-conversion-in-loop	
performance-inefficient-algorithm	Yes
performance-inefficient-string-concatenation	
performance-inefficient-vector-operation	Yes
performance-move-const-arg	Yes
performance-move-constructor-init	
performance-no-automatic-move	
performance-no-int-to-ptr	
performance-noexcept-move-constructor	Yes
performance-trivially-destructible	Yes
performance-type-promotion-in-math-fn	Yes
performance-unnecessary-copy-initialization	Yes
performance-unnecessary-value-param	Yes
portability-restrict-system-includes	Yes

portability-simd-intrinsics	
portability-std-allocator-const	
readability-avoid-const-params-in-decls	Yes
readability-braces-around-statements	Yes
readability-const-return-type	Yes
readability-container-contains	Yes
readability-container-data-pointer	Yes
readability-container-size-empty	Yes
readability-convert-member-functions-to-static	Yes
readability-delete-null-pointer	Yes
readability-duplicate-include	Yes
readability-else-after-return	Yes
readability-function-cognitive-complexity	
readability-function-size	
readability-identifier-length	
readability-identifier-naming	Yes
readability-implicit-bool-conversion	Yes
readability-inconsistent-declaration-parameter-name	Yes
readability-isolate-declaration	Yes
readability-magic-numbers	

readability-make-member-function-const	Yes
readability-misleading-indentation	
readability-misplaced-array-index	Yes
readability-named-parameter	Yes
readability-non-const-parameter	Yes
readability-qualified-auto	Yes
readability-redundant-access-specifiers	Yes
readability-redundant-control-flow	Yes
readability-redundant-declaration	Yes
readability-redundant-function-ptr-dereference	Yes
readability-redundant-member-init	Yes
readability-redundant-preprocessor	
readability-redundant-smartptr-get	Yes
readability-redundant-string-cstr	Yes
readability-redundant-string-init	Yes
readability-simplify-boolean-expr	Yes
readability-simplify-subscript-expr	Yes
readability-static-accessed-through-instance	Yes
readability-static-definition-in-anonymous-namespace	Yes
readability-string-compare	Yes

readability-suspicious-call-argument	
readability-uniqueptr-delete-release	Yes
readability-uppercase-literal-suffix	Yes
readability-use-anyofallof	
zircon-temporary-objects	

Aliases..

Name	Redirect
bugprone-narrowing-conversions	*cppcoreguidelines-narrowing-conversion*
cert-con36-c	*bugprone-spuriously-wake-up-functions*
cert-con54-cpp	*bugprone-spuriously-wake-up-functions*
cert-dcl03-c	*misc-static-assert*
cert-dcl16-c	*readability-uppercase-literal-suffix*
cert-dcl37-c	*bugprone-reserved-identifier*
cert-dcl51-cpp	*bugprone-reserved-identifier*
cert-dcl54-cpp	*misc-new-delete-overloads*
cert-dcl59-cpp	*google-build-namespaces*
cert-err09-cpp	*misc-throw-by-value-catch-by-reference*
cert-err61-cpp	*misc-throw-by-value-catch-by-reference*
cert-exp42-c	*bugprone-suspicious-memory-comparison*

cert-fio38-c	*misc-non-copyable-objects*
cert-flp37-c	*bugprone-suspicious-memory-comparison*
cert-msc30-c	*cert-msc50-cpp*
cert-msc32-c	*cert-msc51-cpp*
cert-oop11-cpp	*performance-move-constructor-init*
cert-oop54-cpp	*bugprone-unhandled-self-assignment*
cert-pos44-c	*bugprone-bad-signal-to-kill-thread*
cert-pos47-c	*concurrency-thread-canceltype-asynchro*
cert-sig30-c	*bugprone-signal-handler*
cert-str34-c	*bugprone-signed-char-misuse*
clang-analyzer-core.CallAndMessage	*Clang Static Analyzer core.CallAndMessage*
clang-analyzer-core.DivideZero	*Clang Static Analyzer core.DivideZero*
clang-analyzer-core.NonNullParamChecker	*Clang Static Analyzer core.NonNullParamChecker*
clang-analyzer-core.NullDereference	*Clang Static Analyzer core.NullDereference*
clang-analyzer-core.StackAddressEscape	*Clang Static Analyzer core.StackAddressEscape*
clang-analyzer-core.UndefinedBinaryOperatorResult	*Clang Static Analyzer core.UndefinedBinaryOperatorResult*
clang-analyzer-core.VLASize	*Clang Static Analyzer*

	core.VLASize
clang-analyzer-core.uninitialized.ArraySubscript	*Clang Static Analyzer core.uninitialized.ArraySubscript*
clang-analyzer-core.uninitialized.Assign	*Clang Static Analyzer core.uninitialized.Assign*
clang-analyzer-core.uninitialized.Branch	*Clang Static Analyzer core.uninitialized.Branch*
clang-analyzer-core.uninitialized.UndefReturn	*Clang Static Analyzer core.uninitialized.UndefReturn*
clang-analyzer-cplusplus.Move	*Clang Static Analyzer cplusplus.Move*
clang-analyzer-cplusplus.NewDelete	*Clang Static Analyzer cplusplus.NewDelete*
clang-analyzer-cplusplus.NewDeleteLeaks	*Clang Static Analyzer cplusplus.NewDeleteLeaks*
clang-analyzer-deadcode.DeadStores	*Clang Static Analyzer deadcode.DeadStores*
clang-analyzer-nullability.NullPassedToNonnull	*Clang Static Analyzer nullability.NullPassedToNonnull*
clang-analyzer-nullability.NullReturnedFromNonnull	*Clang Static Analyzer nullability.NullReturnedFromNonnull*
clang-analyzer-nullability.NullableDereferenced	*Clang Static Analyzer nullability.NullableDereferenced*
clang-analyzer-nullability.NullablePassedToNonnull	*Clang Static Analyzer nullability.NullablePassedToNonnull*
clang-analyzer-optin.cplusplus.UninitializedObject	*Clang Static Analyzer optin.cplusplus.UninitializedObject*

clang-analyzer-optin.cplusplus.VirtualCall	*Clang Static Analyzer optin.cplusplus.VirtualCall*
clang-analyzer-optin.mpi.MPI-Checker	*Clang Static Analyzer optin.mpi.MPI-Checker*
clang-analyzer-optin.osx.cocoa.localizability.EmptyLocalizationContextChecker	*Clang Static Analyzer optin.osx.cocoa.localizability.EmptyLoca*
clang-analyzer-optin.osx.cocoa.localizability.NonLocalizedStringChecker	*Clang Static Analyzer optin.osx.cocoa.localizability.NonLocaliz*
clang-analyzer-osx.API	*Clang Static Analyzer osx.API*
clang-analyzer-osx.SecKeychainAPI	*Clang Static Analyzer osx.SecKeychainAPI*
clang-analyzer-osx.cocoa.AtSync	*Clang Static Analyzer osx.cocoa.AtSync*
clang-analyzer-osx.cocoa.ClassRelease	*Clang Static Analyzer osx.cocoa.ClassRelease*
clang-analyzer-osx.cocoa.Dealloc	*Clang Static Analyzer osx.cocoa.Dealloc*
clang-analyzer-osx.cocoa.IncompatibleMethodTypes	*Clang Static Analyzer osx.cocoa.IncompatibleMethodTypes*
clang-analyzer-osx.cocoa.NSAutoreleasePool	*Clang Static Analyzer osx.cocoa.NSAutoreleasePool*
clang-analyzer-osx.cocoa.NSError	*Clang Static Analyzer osx.cocoa.NSError*
clang-analyzer-osx.cocoa.NilArg	*Clang Static Analyzer osx.cocoa.NilArg*

clang-analyzer-osx.cocoa.ObjCGenerics	*Clang Static Analyzer osx.cocoa.ObjCGenerics*
clang-analyzer-osx.cocoa.RetainCount	*Clang Static Analyzer osx.cocoa.RetainCount*
clang-analyzer-osx.cocoa.SelfInit	*Clang Static Analyzer osx.cocoa.SelfInit*
clang-analyzer-osx.cocoa.SuperDealloc	*Clang Static Analyzer osx.cocoa.SuperDealloc*
clang-analyzer-osx.cocoa.UnusedIvars	*Clang Static Analyzer osx.cocoa.UnusedIvars*
clang-analyzer-osx.cocoa.VariadicMethodTypes	*Clang Static Analyzer osx.cocoa.VariadicMethodTypes*
clang-analyzer-osx.coreFoundation.CFError	*Clang Static Analyzer osx.coreFoundation.CFError*
clang-analyzer-osx.coreFoundation.CFNumber	*Clang Static Analyzer osx.coreFoundation.CFNumber*
clang-analyzer-osx.coreFoundation.CFRetainRelease	*Clang Static Analyzer osx.coreFoundation.CFRetainRelease*
clang-analyzer-osx.coreFoundation.containers.OutOfBounds	*Clang Static Analyzer osx.coreFoundation.containers.OutOfBou*
clang-analyzer-osx.coreFoundation.containers.PointerSizedValues	*Clang Static Analyzer osx.coreFoundation.containers.PointerSi*
clang-analyzer-security.FloatLoopCounter	*Clang Static Analyzer security.FloatLoopCounter*
clang-analyzer-security.insecureAPI.DeprecatedOrUnsafeBufferHandling	*Clang Static Analyzer security.insecureAPI.DeprecatedOrUnsa*
clang-analyzer-security.insecureAPI.UncheckedReturn	*Clang Static Analyzer*

| | |*security.insecureAPI.UncheckedReturn*
+---+---+
|*clang-analyzer-security.insecureAPI.bcmp* |*Clang Static Analyzer*
| |*security.insecureAPI.bcmp*
+---+---+
|*clang-analyzer-security.insecureAPI.bcopy* |*Clang Static Analyzer*
| |*security.insecureAPI.bcopy*
+---+---+
|*clang-analyzer-security.insecureAPI.bzero* |*Clang Static Analyzer*
| |*security.insecureAPI.bzero*
+---+---+
|*clang-analyzer-security.insecureAPI.getpw* |*Clang Static Analyzer*
| |*security.insecureAPI.getpw*
+---+---+
|*clang-analyzer-security.insecureAPI.gets* |*Clang Static Analyzer*
| |*security.insecureAPI.gets*
+---+---+
|*clang-analyzer-security.insecureAPI.mkstemp* |*Clang Static Analyzer*
| |*security.insecureAPI.mkstemp*
+---+---+
|*clang-analyzer-security.insecureAPI.mktemp* |*Clang Static Analyzer*
| |*security.insecureAPI.mktemp*
+---+---+
|*clang-analyzer-security.insecureAPI.rand* |*Clang Static Analyzer*
| |*security.insecureAPI.rand*
+---+---+
|*clang-analyzer-security.insecureAPI.strcpy* |*Clang Static Analyzer*
| |*security.insecureAPI.strcpy*
+---+---+
|*clang-analyzer-security.insecureAPI.vfork* |*Clang Static Analyzer*
| |*security.insecureAPI.vfork*
+---+---+
|*clang-analyzer-unix.API* |*Clang Static Analyzer*
| |*unix.API*
+---+---+
|*clang-analyzer-unix.Malloc* |*Clang Static Analyzer*
| |*unix.Malloc*
+---+---+
|*clang-analyzer-unix.MallocSizeof* |*Clang Static Analyzer*
| |*unix.MallocSizeof*

clang-analyzer-unix.MismatchedDeallocator	*Clang Static Analyzer unix.MismatchedDeallocator*
clang-analyzer-unix.Vfork	*Clang Static Analyzer unix.Vfork*
clang-analyzer-unix.cstring.BadSizeArg	*Clang Static Analyzer unix.cstring.BadSizeArg*
clang-analyzer-unix.cstring.NullArg	*Clang Static Analyzer unix.cstring.NullArg*
cppcoreguidelines-avoid-c-arrays	*modernize-avoid-c-arrays*
cppcoreguidelines-avoid-magic-numbers	*readability-magic-numbers*
cppcoreguidelines-c-copy-assignment-signature	*misc-unconventional-assign-operator*
cppcoreguidelines-explicit-virtual-functions	*modernize-use-override*
cppcoreguidelines-macro-to-enum	*modernize-macro-to-enum*
cppcoreguidelines-non-private-member-variables-in-classes	*misc-non-private-member-variables-in-c*
fuchsia-header-anon-namespaces	*google-build-namespaces*
google-readability-braces-around-statements	*readability-braces-around-statements*
google-readability-function-size	*readability-function-size*
google-readability-namespace-comments	*llvm-namespace-comment*
hicpp-avoid-c-arrays	*modernize-avoid-c-arrays*
hicpp-braces-around-statements	*readability-braces-around-statements*
hicpp-deprecated-headers	*modernize-deprecated-headers*
hicpp-explicit-conversions	*google-explicit-constructor*

hicpp-function-size	*readability-function-size*
hicpp-invalid-access-moved	*bugprone-use-after-move*
hicpp-member-init	*cppcoreguidelines-pro-type-member-init*
hicpp-move-const-arg	*performance-move-const-arg*
hicpp-named-parameter	*readability-named-parameter*
hicpp-new-delete-operators	*misc-new-delete-overloads*
hicpp-no-array-decay	*cppcoreguidelines-pro-bounds-array-to-p*
hicpp-no-malloc	*cppcoreguidelines-no-malloc*
hicpp-noexcept-move	*performance-noexcept-move-constructor*
hicpp-special-member-functions	*cppcoreguidelines-special-member-functi*
hicpp-static-assert	*misc-static-assert*
hicpp-undelegated-constructor	*bugprone-undelegated-constructor*
hicpp-uppercase-literal-suffix	*readability-uppercase-literal-suffix*
hicpp-use-auto	*modernize-use-auto*
hicpp-use-emplace	*modernize-use-emplace*
hicpp-use-equals-default	*modernize-use-equals-default*
hicpp-use-equals-delete	*modernize-use-equals-delete*
hicpp-use-noexcept	*modernize-use-noexcept*
hicpp-use-nullptr	*modernize-use-nullptr*
hicpp-use-override	*modernize-use-override*

hicpp-vararg	*cppcoreguidelines-pro-type-vararg*
llvm-else-after-return	*readability-else-after-return*
llvm-qualified-auto	*readability-qualified-auto*

Clang-tidy IDE/Editor Integrations

Apart from being a standalone tool, **clang-tidy** is integrated into various IDEs, code analyzers, and editors. We recommend using *clangd* which integrates **clang-tidy** and *is available* in most major editors through plugins (Vim, Emacs, Visual Studio Code, Sublime Text and *more*).

The following table shows the most well-known **clang-tidy** integrations in detail.

Tool	Feature: On-the-fly inspection	Check list configuration (GUI)	Options to checks (GUI)	Configuration via **.clang-tidy** files	Custom clang-tidy binary
A.L.E. for Vim	+	-	-	-	+
Clang Power Tools for Visual Studio	-	+	-	+	-
Clangd	+	-	-	+	-
CLion IDE	+	+	+	+	+
CodeChecker	-	-	-	-	+
CPPCheck	-	-	-	-	-
CPPDepend	-	-	-	-	-

Flycheck for Emacs	+	-	-	+	+
KDevelop IDE	-	+	+	+	+
Qt Creator IDE	+	+	-	+	+
ReSharper C++ for Visual Studio	+	+	-	+	+
Syntastic for Vim	+	-	-	-	+
Visual Assist for Visual Studio	+	+	-	-	-

IDEs

CLion 2017.2 and later *integrates clang-tidy* as an extension to the built-in code analyzer. Starting from 2018.2 EAP, CLion allows using **clang-tidy** via Clangd. Inspections and applicable quick-fixes are performed on the fly, and checks can be configured in standard command line format. In this integration, you can switch to the **clang-tidy** binary different from the bundled one, pass the configuration in **.clang-tidy** files instead of using the IDE settings, and configure options for particular checks.

KDevelop with the *kdev-clang-tidy* plugin, starting from version 5.1, performs static analysis using **clang-tidy**. The plugin launches the **clang-tidy** binary from the specified location and parses its output to provide a list of issues.

QtCreator 4.6 integrates **clang-tidy** warnings into the editor diagnostics under the *Clang Code Model*. To employ **clang-tidy** inspection in QtCreator, you need to create a copy of one of the presets and choose the checks to be performed. Since QtCreator 4.7 project-wide analysis is possible with the *Clang Tools* analyzer.

MS Visual Studio has a native *clang-tidy-vs* plugin and also can integrate **clang-tidy** by means of three other tools. The *ReSharper C++* extension, version 2017.3 and later, provides seamless **clang-tidy** integration: checks and quick-fixes run alongside native inspections. Apart from that, ReSharper C++ incorporates **clang-tidy** as a separate step of its code clean-up process. *Visual Assist* build 2210 includes a subset of **clang-tidy** checklist to inspect the code as you edit. Another way to bring **clang-tidy** functionality to Visual Studio is the *Clang Power Tools* plugin, which includes most of the **clang-tidy** checks and runs them during compilation or as a separate step of code analysis.

Editors

Emacs24, when expanded with the *Flycheck* plugin, incorporates the **clang-tidy** inspection into the syntax analyzer. For *Vim*, you can use *Syntastic*, which includes **clang-tidy**, or *A.L.E.*, a lint engine that applies **clang-tidy** along with other linters.

Analyzers

clang-tidy is integrated in *CPPDepend* starting from version 2018.1 and *CPPCheck* 1.82. CPPCheck integration lets you import Visual Studio solutions and run the **clang-tidy** inspection on them. The *CodeChecker* application of version 5.3 or later, which also comes as a *plugin* for Eclipse, supports **clang-tidy** as a static analysis instrument and allows to use a custom **clang-tidy** binary.

Getting Involved

clang-tidy has several own checks and can run Clang static analyzer checks, but its power is in the ability to easily write custom checks.

Checks are organized in modules, which can be linked into **clang-tidy** with minimal or no code changes in **clang-tidy**.

Checks can plug into the analysis on the preprocessor level using *PPCallbacks* or on the AST level using *AST Matchers*. When an error is found, checks can report them in a way similar to how Clang diagnostics work. A fix-it hint can be attached to a diagnostic message.

The interface provided by **clang-tidy** makes it easy to write useful and precise checks in just a few lines of code. If you have an idea for a good check, the rest of this document explains how to do this.

There are a few tools particularly useful when developing clang-tidy checks:

- **add_new_check.py** is a script to automate the process of adding a new check, it will create the check, update the CMake file and create a test;

- **rename_check.py** does what the script name suggests, renames an existing check;
- **pp-trace** logs method calls on *PPCallbacks* for a source file and is invaluable in understanding the preprocessor mechanism;
- **clang-query** is invaluable for interactive prototyping of AST matchers and exploration of the Clang AST;
- *clang-check* with the **-ast-dump** (and optionally **-ast-dump-filter**) provides a convenient way to dump AST of a C++ program.

If CMake is configured with **CLANG_TIDY_ENABLE_STATIC_ANALYZER=NO**, **clang-tidy** will not be built with support for the **clang-analyzer-*** checks or the **mpi-*** checks.

Choosing the Right Place for your Check

If you have an idea of a check, you should decide whether it should be implemented as a:

- *Clang diagnostic*: if the check is generic enough, targets code patterns that most probably are bugs (rather than style or readability issues), can be implemented effectively and with extremely low false positive rate, it may make a good Clang diagnostic.
- *Clang static analyzer check*: if the check requires some sort of control flow analysis, it should probably be implemented as a static analyzer check.
- *clang-tidy check* is a good choice for linter-style checks, checks that are related to a certain coding style, checks that address code readability, etc.

Preparing your Workspace

If you are new to LLVM development, you should read the *Getting Started with the LLVM System*, *Using Clang Tools* and *How To Setup Clang Tooling For LLVM* documents to check out and build LLVM, Clang and Clang Extra Tools with CMake.

Once you are done, change to the **llvm/clang-tools-extra** directory, and let's start!

When you *configure the CMake build*, make sure that you enable the **clang** and **clang-tools-extra** projects to build **clang-tidy**. Because your new check will have associated documentation, you will also want to install *Sphinx* and enable it in the CMake configuration. To save build time of the core Clang libraries you may want to only enable the **X86** target in the CMake configuration.

The Directory Structure

clang-tidy source code resides in the **llvm/clang-tools-extra** directory and is structured as follows:

```
clang-tidy/                    # Clang-tidy core.
|-- ClangTidy.h                # Interfaces for users.
|-- ClangTidyCheck.h           # Interfaces for checks.
|-- ClangTidyModule.h          # Interface for clang-tidy modules.
|-- ClangTidyModuleRegistry.h  # Interface for registering of modules.
   ...
|-- google/                    # Google clang-tidy module.
|-+
  |-- GoogleTidyModule.cpp
  |-- GoogleTidyModule.h
       ...
|-- llvm/                      # LLVM clang-tidy module.
|-+
  |-- LLVMTidyModule.cpp
  |-- LLVMTidyModule.h
       ...
|-- objc/                      # Objective-C clang-tidy module.
|-+
  |-- ObjCTidyModule.cpp
  |-- ObjCTidyModule.h
       ...
|-- tool/                      # Sources of the clang-tidy binary.
       ...
test/clang-tidy/               # Integration tests.
   ...
unittests/clang-tidy/          # Unit tests.
|-- ClangTidyTest.h
|-- GoogleModuleTest.cpp
|-- LLVMModuleTest.cpp
|-- ObjCModuleTest.cpp
   ...
```

Writing a clang-tidy Check

So you have an idea of a useful check for **clang-tidy**.

First, if you're not familiar with LLVM development, read through the *Getting Started with LLVM* document for instructions on setting up your workflow and the *LLVM Coding Standards* document to familiarize yourself with the coding style used in the project. For code reviews we mostly use *LLVM*

Phabricator.

Next, you need to decide which module the check belongs to. Modules are located in subdirectories of *clang-tidy/* and contain checks targeting a certain aspect of code quality (performance, readability, etc.), certain coding style or standard (Google, LLVM, CERT, etc.) or a widely used API (e.g. MPI). Their names are the same as the user-facing check group names described *above*.

After choosing the module and the name for the check, run the **clang-tidy/add_new_check.py** script to create the skeleton of the check and plug it to **clang-tidy**. It's the recommended way of adding new checks.

If we want to create a *readability-awesome-function-names*, we would run:

```
$ clang-tidy/add_new_check.py readability awesome-function-names
```

The add_new_check.py script will:

- create the class for your check inside the specified module's directory and register it in the module and in the build system;
- create a lit test file in the **test/clang-tidy/** directory;
- create a documentation file and include it into the **docs/clang-tidy/checks/list.rst**.

Let's see in more detail at the check class definition:

```
...

#include "../ClangTidyCheck.h"

namespace clang {
namespace tidy {
namespace readability {

...
class AwesomeFunctionNamesCheck : public ClangTidyCheck {
public:
  AwesomeFunctionNamesCheck(StringRef Name, ClangTidyContext *Context)
     : ClangTidyCheck(Name, Context) {}
  void registerMatchers(ast_matchers::MatchFinder *Finder) override;
```

```
  void check(const ast_matchers::MatchFinder::MatchResult &Result) override;
};

} // namespace readability
} // namespace tidy
} // namespace clang

...
```

Constructor of the check receives the **Name** and **Context** parameters, and must forward them to the **ClangTidyCheck** constructor.

In our case the check needs to operate on the AST level and it overrides the **registerMatchers** and **check** methods. If we wanted to analyze code on the preprocessor level, we'd need instead to override the **registerPPCallbacks** method.

In the **registerMatchers** method we create an AST Matcher (see *AST Matchers* for more information) that will find the pattern in the AST that we want to inspect. The results of the matching are passed to the **check** method, which can further inspect them and report diagnostics.

```
using namespace ast_matchers;

void AwesomeFunctionNamesCheck::registerMatchers(MatchFinder *Finder) {
  Finder->addMatcher(functionDecl().bind("x"), this);
}

void AwesomeFunctionNamesCheck::check(const MatchFinder::MatchResult &Result) {
  const auto *MatchedDecl = Result.Nodes.getNodeAs<FunctionDecl>("x");
  if (!MatchedDecl->getIdentifier() || MatchedDecl->getName().startswith("awesome_"))
    return;
  diag(MatchedDecl->getLocation(), "function %0 is insufficiently awesome")
      << MatchedDecl
      << FixItHint::CreateInsertion(MatchedDecl->getLocation(), "awesome_");
}
```

(If you want to see an example of a useful check, look at *clang-tidy/google/ExplicitConstructorCheck.h* and *clang-tidy/google/ExplicitConstructorCheck.cpp*).

If you need to interact with macros or preprocessor directives, you will want to override the

method **registerPPCallbacks**. The **add_new_check.py** script does not generate an override for this method in the starting point for your new check.

If your check applies only under a specific set of language options, be sure to override the method **isLanguageVersionSupported** to reflect that.

Check development tips

Writing your first check can be a daunting task, particularly if you are unfamiliar with the LLVM and Clang code bases. Here are some suggestions for orienting yourself in the codebase and working on your check incrementally.

Guide to useful documentation

Many of the support classes created for LLVM are used by Clang, such as *StringRef* and *SmallVector*. These and other commonly used classes are described in the *Important and useful LLVM APIs* and *Picking the Right Data Structure for the Task* sections of the *LLVM Programmer's Manual*. You don't need to memorize all the details of these classes; the generated *doxygen documentation* has everything if you need it. In the header *LLVM/ADT/STLExtras.h* you'll find useful versions of the STL algorithms that operate on LLVM containers, such as *llvm::all_of*.

Clang is implemented on top of LLVM and introduces its own set of classes that you will interact with while writing your check. When a check issues diagnostics and fix-its, these are associated with locations in the source code. Source code locations, source files, ranges of source locations and the *SourceManager* class provide the mechanisms for describing such locations. These and other topics are described in the *"Clang" CFE Internals Manual*. Whereas the doxygen generated documentation serves as a reference to the internals of Clang, this document serves as a guide to other developers. Topics in that manual of interest to a check developer are:

- *The Clang "Basic" Library* for information about diagnostics, fix-it hints and source locations.

- *The Lexer and Preprocessor Library* for information about tokens, lexing (transforming characters into tokens) and the preprocessor.

- *The AST Library* for information about how C++ source statements are represented as an abstract syntax tree (AST).

  Most checks will interact with C++ source code via the AST. Some checks will interact with the preprocessor. The input source file is lexed and preprocessed and then parsed into the AST. Once the AST is fully constructed, the check is run by applying the check's registered AST matchers against the AST and invoking the check with the set of matched nodes from the AST. Monitoring the actions of the preprocessor is detached from the AST construction, but a check

can collect information during preprocessing for later use by the check when nodes are matched by the AST.

Every syntactic (and sometimes semantic) element of the C++ source code is represented by different classes in the AST. You select the portions of the AST you're interested in by composing AST matcher functions. You will want to study carefully the *AST Matcher Reference* to understand the relationship between the different matcher functions.

Using the Transformer library

The Transformer library allows you to write a check that transforms source code by expressing the transformation as a **RewriteRule**. The Transformer library provides functions for composing edits to source code to create rewrite rules. Unless you need to perform low-level source location manipulation, you may want to consider writing your check with the Transformer library. The *Clang Transformer Tutorial* describes the Transformer library in detail.

To use the Transformer library, make the following changes to the code generated by the **add_new_check.py** script:

- Include **../utils/TransformerClangTidyCheck.h** instead of **../ClangTidyCheck.h**
- Change the base class of your check from **ClangTidyCheck** to **TransformerClangTidyCheck**
- Delete the override of the **registerMatchers** and **check** methods in your check class.
- Write a function that creates the **RewriteRule** for your check.
- Call the function in your check's constructor to pass the rewrite rule to **TransformerClangTidyCheck**'s constructor.

Developing your check incrementally

The best way to develop your check is to start with the simple test cases and increase complexity incrementally. The test file created by the **add_new_check.py** script is a starting point for your test cases. A rough outline of the process looks like this:

- Write a test case for your check.
- Prototype matchers on the test file using **clang-query**.
- Capture the working matchers in the **registerMatchers** method.

- Issue the necessary diagnostics and fix-its in the **check** method.
- Add the necessary **CHECK-MESSAGES** and **CHECK-FIXES** annotations to your test case to validate the diagnostics and fix-its.
- Build the target **check-clang-tool** to confirm the test passes.
- Repeat the process until all aspects of your check are covered by tests.

  The quickest way to prototype your matcher is to use **clang-query** to interactively build up your matcher. For complicated matchers, build up a matching expression incrementally and use **clang-query**'s **let** command to save named matching expressions to simplify your matcher. Just like breaking up a huge function into smaller chunks with intention-revealing names can help you understand a complex algorithm, breaking up a matcher into smaller matchers with intention-revealing names can help you understand a complicated matcher. Once you have a working matcher, the C++ API will be virtually identical to your interactively constructed matcher. You can use local variables to preserve your intention-revealing names that you applied to nested matchers.

Creating private matchers

Sometimes you want to match a specific aspect of the AST that isn't provided by the existing AST matchers. You can create your own private matcher using the same infrastructure as the public matchers. A private matcher can simplify the processing in your **check** method by eliminating complex hand-crafted AST traversal of the matched nodes. Using the private matcher allows you to select the desired portions of the AST directly in the matcher and refer to it by a bound name in the **check** method.

Unit testing helper code

Private custom matchers are a good example of auxiliary support code for your check that can be tested with a unit test. It will be easier to test your matchers or other support classes by writing a unit test than by writing a **FileCheck** integration test. The **ASTMatchersTests** target contains unit tests for the public AST matcher classes and is a good source of testing idioms for matchers.

You can build the Clang-tidy unit tests by building the **ClangTidyTests** target. Test targets in LLVM and Clang are excluded from the "build all" style action of IDE-based CMake generators, so you need to explicitly build the target for the unit tests to be built.

Making your check robust

Once you've covered your check with the basic "happy path" scenarios, you'll want to torture your check with as many edge cases as you can cover in order to ensure your check is robust. Running your

check on a large code base, such as Clang/LLVM, is a good way to catch things you forgot to account for in your matchers. However, the LLVM code base may be insufficient for testing purposes as it was developed against a particular set of coding styles and quality measures. The larger the corpus of code the check is tested against, the higher confidence the community will have in the check's efficacy and false positive rate.

Some suggestions to ensure your check is robust:

- Create header files that contain code matched by your check.
- Validate that fix-its are properly applied to test header files with **clang-tidy**. You will need to perform this test manually until automated support for checking messages and fix-its is added to the **check_clang_tidy.py** script.
- Define macros that contain code matched by your check.
- Define template classes that contain code matched by your check.
- Define template specializations that contain code matched by your check.
- Test your check under both Windows and Linux environments.
- Watch out for high false positive rates. Ideally, a check would have no false positives, but given that matching against an AST is not control- or data flow- sensitive, a number of false positives are expected. The higher the false positive rate, the less likely the check will be adopted in practice. Mechanisms should be put in place to help the user manage false positives.
- There are two primary mechanisms for managing false positives: supporting a code pattern which allows the programmer to silence the diagnostic in an ad hoc manner and check configuration options to control the behavior of the check.
- Consider supporting a code pattern to allow the programmer to silence the diagnostic whenever such a code pattern can clearly express the programmer's intent. For example, allowing an explicit cast to **void** to silence an unused variable diagnostic.
- Consider adding check configuration options to allow the user to opt into more aggressive checking behavior without burdening users for the common high-confidence cases.

Documenting your check

The **add_new_check.py** script creates entries in the *release notes*, the list of checks and a new file for

the check documentation itself. It is recommended that you have a concise summation of what your check does in a single sentence that is repeated in the release notes, as the first sentence in the doxygen comments in the header file for your check class and as the first sentence of the check documentation. Avoid the phrase "this check" in your check summation and check documentation.

If your check relates to a published coding guideline (C++ Core Guidelines, MISRA, etc.) or style guide, provide links to the relevant guideline or style guide sections in your check documentation.

Provide enough examples of the diagnostics and fix-its provided by the check so that a user can easily understand what will happen to their code when the check is run. If there are exceptions or limitations to your check, document them thoroughly. This will help users understand the scope of the diagnostics and fix-its provided by the check.

Building the target **docs-clang-tools-html** will run the Sphinx documentation generator and create documentation HTML files in the tools/clang/tools/extra/docs/html directory in your build tree. Make sure that your check is correctly shown in the release notes and the list of checks. Make sure that the formatting and structure of your check's documentation looks correct.

Registering your Check

(The **add_new_check.py** script takes care of registering the check in an existing module. If you want to create a new module or know the details, read on.)

The check should be registered in the corresponding module with a distinct name:

```
class MyModule : public ClangTidyModule {
 public:
  void addCheckFactories(ClangTidyCheckFactories &CheckFactories) override {
    CheckFactories.registerCheck<ExplicitConstructorCheck>(
        "my-explicit-constructor");
  }
};
```

Now we need to register the module in the **ClangTidyModuleRegistry** using a statically initialized variable:

```
static ClangTidyModuleRegistry::Add<MyModule> X("my-module",
                                    "Adds my lint checks.");
```

When using LLVM build system, we need to use the following hack to ensure the module is linked into the **clang-tidy** binary:

Add this near the **ClangTidyModuleRegistry::Add<MyModule>** variable:

```
// This anchor is used to force the linker to link in the generated object file
// and thus register the MyModule.
volatile int MyModuleAnchorSource = 0;
```

And this to the main translation unit of the **clang-tidy** binary (or the binary you link the **clang-tidy** library in) **clang-tidy/tool/ClangTidyMain.cpp**:

```
// This anchor is used to force the linker to link the MyModule.
extern volatile int MyModuleAnchorSource;
static int MyModuleAnchorDestination = MyModuleAnchorSource;
```

Configuring Checks

If a check needs configuration options, it can access check-specific options using the **Options.get<Type>("SomeOption", DefaultValue)** call in the check constructor. In this case the check should also override the **ClangTidyCheck::storeOptions** method to make the options provided by the check discoverable. This method lets **clang-tidy** know which options the check implements and what the current values are (e.g. for the **-dump-config** command line option).

```
class MyCheck : public ClangTidyCheck {
  const unsigned SomeOption1;
  const std::string SomeOption2;

public:
  MyCheck(StringRef Name, ClangTidyContext *Context)
    : ClangTidyCheck(Name, Context),
      SomeOption(Options.get("SomeOption1", -1U)),
      SomeOption(Options.get("SomeOption2", "some default")) {}

  void storeOptions(ClangTidyOptions::OptionMap &Opts) override {
    Options.store(Opts, "SomeOption1", SomeOption1);
    Options.store(Opts, "SomeOption2", SomeOption2);
  }
  ...
```

Assuming the check is registered with the name "my-check", the option can then be set in a **.clang-tidy** file in the following way:

```
CheckOptions:
```

```
my-check.SomeOption1: 123
my-check.SomeOption2: 'some other value'
```

If you need to specify check options on a command line, you can use the inline YAML format:

```
$ clang-tidy -config="{CheckOptions: {a: b, x: y}}" ...
```

Testing Checks

To run tests for **clang-tidy**, build the **check-clang-tools** target. For instance, if you configured your CMake build with the ninja project generator, use the command:

```
$ ninja check-clang-tools
```

clang-tidy checks can be tested using either unit tests or *lit* tests. Unit tests may be more convenient to test complex replacements with strict checks. *Lit* tests allow using partial text matching and regular expressions which makes them more suitable for writing compact tests for diagnostic messages.

The **check_clang_tidy.py** script provides an easy way to test both diagnostic messages and fix-its. It filters out **CHECK** lines from the test file, runs **clang-tidy** and verifies messages and fixes with two separate *FileCheck* invocations: once with FileCheck's directive prefix set to **CHECK-MESSAGES**, validating the diagnostic messages, and once with the directive prefix set to **CHECK-FIXES**, running against the fixed code (i.e., the code after generated fix-its are applied). In particular, **CHECK-FIXES:** can be used to check that code was not modified by fix-its, by checking that it is present unchanged in the fixed code. The full set of *FileCheck* directives is available (e.g., **CHECK-MESSAGES-SAME:**, **CHECK-MESSAGES-NOT:**), though typically the basic **CHECK** forms (**CHECK-MESSAGES** and **CHECK-FIXES**) are sufficient for clang-tidy tests. Note that the *FileCheck* documentation mostly assumes the default prefix (**CHECK**), and hence describes the directive as **CHECK:**, **CHECK-SAME:**, **CHECK-NOT:**, etc. Replace **CHECK** by either **CHECK-FIXES** or **CHECK-MESSAGES** for clang-tidy tests.

An additional check enabled by **check_clang_tidy.py** ensures that if *CHECK-MESSAGES:* is used in a file then every warning or error must have an associated CHECK in that file. Or, you can use **CHECK-NOTES:** instead, if you want to **also** ensure that all the notes are checked.

To use the **check_clang_tidy.py** script, put a .cpp file with the appropriate **RUN** line in the **test/clang-tidy** directory. Use **CHECK-MESSAGES:** and **CHECK-FIXES:** lines to write checks against diagnostic messages and fixed code.

It’s advised to make the checks as specific as possible to avoid checks matching to incorrect parts of the input. Use **[[@LINE+X]]/[[@LINE-X]]** substitutions and distinct function and variable names in the test code.

Here’s an example of a test using the **check_clang_tidy.py** script (the full source code is at *test/clang-tidy/checkers/google/readability-casting.cpp*):

```
// RUN: %check_clang_tidy %s google-readability-casting %t

void f(int a) {
  int b = (int)a;
  // CHECK-MESSAGES: :[[@LINE-1]]:11: warning: redundant cast to the same type [google-readability-casting]
  // CHECK-FIXES: int b = a;
}
```

To check more than one scenario in the same test file use **-check-suffix=SUFFIX-NAME** on **check_clang_tidy.py** command line or **-check-suffixes=SUFFIX-NAME-1,SUFFIX-NAME-2,...**. With **-check-suffix[es]=SUFFIX-NAME** you need to replace your **CHECK-*** directives with **CHECK-MESSAGES-SUFFIX-NAME** and **CHECK-FIXES-SUFFIX-NAME**.

Here’s an example:

```
// RUN: %check_clang_tidy -check-suffix=USING-A %s misc-unused-using-decls %t -- -- -DUSING_A
// RUN: %check_clang_tidy -check-suffix=USING-B %s misc-unused-using-decls %t -- -- -DUSING_B
// RUN: %check_clang_tidy %s misc-unused-using-decls %t
...
// CHECK-MESSAGES-USING-A: :[[@LINE-8]]:10: warning: using decl 'A' {{.*}}
// CHECK-MESSAGES-USING-B: :[[@LINE-7]]:10: warning: using decl 'B' {{.*}}
// CHECK-MESSAGES: :[[@LINE-6]]:10: warning: using decl 'C' {{.*}}
// CHECK-FIXES-USING-A-NOT: using a::A;$
// CHECK-FIXES-USING-B-NOT: using a::B;$
// CHECK-FIXES-NOT: using a::C;$
```

There are many dark corners in the C++ language, and it may be difficult to make your check work perfectly in all cases, especially if it issues fix-it hints. The most frequent pitfalls are macros and templates:

1. code written in a macro body/template definition may have a different meaning depending on the macro expansion/template instantiation;

2. multiple macro expansions/template instantiations may result in the same code being inspected by the check multiple times (possibly, with different meanings, see 1), and the same warning (or a slightly different one) may be issued by the check multiple times; **clang-tidy** will deduplicate _identical_ warnings, but if the warnings are slightly different, all of them will be shown to the user (and used for applying fixes, if any);

3. making replacements to a macro body/template definition may be fine for some macro expansions/template instantiations, but easily break some other expansions/instantiations.

If you need multiple files to exercise all the aspects of your check, it is recommended you place them in a subdirectory named for the check under the **Inputs** directory for the module containing your check. This keeps the test directory from getting cluttered.

If you need to validate how your check interacts with system header files, a set of simulated system header files is located in the **checkers/Inputs/Headers** directory. The path to this directory is available in a lit test with the variable **%clang_tidy_headers**.

Out-of-tree check plugins

Developing an out-of-tree check as a plugin largely follows the steps outlined above. The plugin is a shared library whose code lives outside the clang-tidy build system. Build and link this shared library against LLVM as done for other kinds of Clang plugins.

The plugin can be loaded by passing *-load* to *clang-tidy* in addition to the names of the checks to enable.

```
$ clang-tidy --checks=-*,my-explicit-constructor -list-checks -load myplugin.so
```

There is no expectations regarding ABI and API stability, so the plugin must be compiled against the version of clang-tidy that will be loading the plugin.

The plugins can use threads, TLS, or any other facilities available to in-tree code which is accessible from the external headers.

Running clang-tidy on LLVM

To test a check it's best to try it out on a larger code base. LLVM and Clang are the natural targets as you already have the source code around. The most convenient way to run **clang-tidy** is with a compile command database; CMake can automatically generate one, for a description of how to enable it see *How To Setup Clang Tooling For LLVM*. Once **compile_commands.json** is in place and a working version of **clang-tidy** is in **PATH** the entire code base can be analyzed with **clang-tidy/tool/run-clang-tidy.py**. The script executes **clang-tidy** with the default set of checks on every

translation unit in the compile command database and displays the resulting warnings and errors. The script provides multiple configuration flags.

- The default set of checks can be overridden using the **-checks** argument, taking the identical format as **clang-tidy** does. For example **-checks=-*,modernize-use-override** will run the **modernize-use-override** check only.

- To restrict the files examined you can provide one or more regex arguments that the file names are matched against. **run-clang-tidy.py clang-tidy/.*Check\.cpp** will only analyze clang-tidy checks. It may also be necessary to restrict the header files that warnings are displayed from using the **-header-filter** flag. It has the same behavior as the corresponding **clang-tidy** flag.

- To apply suggested fixes **-fix** can be passed as an argument. This gathers all changes in a temporary directory and applies them. Passing **-format** will run clang-format over changed lines.

On checks profiling

clang-tidy can collect per-check profiling info, and output it for each processed source file (translation unit).

To enable profiling info collection, use the **-enable-check-profile** argument. The timings will be output to **stderr** as a table. Example output:

```
$ clang-tidy -enable-check-profile -checks=-*,readability-function-size source.cpp
===-------------------------------------------------------------------------===
                          clang-tidy checks profiling
===-------------------------------------------------------------------------===
  Total Execution Time: 1.0282 seconds (1.0258 wall clock)

    ---User Time---   --System Time--   --User+System--   ---Wall Time---  --- Name ---
    0.9136 (100.0%)   0.1146 (100.0%)   1.0282 (100.0%)   1.0258 (100.0%)  readability-function-size
    0.9136 (100.0%)   0.1146 (100.0%)   1.0282 (100.0%)   1.0258 (100.0%)  Total
```

It can also store that data as JSON files for further processing. Example output:

```
$ clang-tidy -enable-check-profile -store-check-profile=. -checks=-*,readability-function-size source.cpp
$ # Note that there won't be timings table printed to the console.
$ ls /tmp/out/
20180516161318717446360-source.cpp.json
$ cat 20180516161318717446360-source.cpp.json
{
```

"file": "/path/to/source.cpp",
"timestamp": "2018-05-16 16:13:18.717446360",
"profile": {
"time.clang-tidy.readability-function-size.wall": 1.0421266555786133e+00,
"time.clang-tidy.readability-function-size.user": 9.208840000005421e-01,
"time.clang-tidy.readability-function-size.sys": 1.2418899999999974e-01
}
}

There is only one argument that controls profile storage:

- **-store-check-profile=<prefix>**

By default reports are printed in tabulated format to stderr. When this option is passed, these per-TU profiles are instead stored as JSON. If the prefix is not an absolute path, it is considered to be relative to the directory from where you have run **clang-tidy**. All **.** and **..** patterns in the path are collapsed, and symlinks are resolved.

Example: Let's suppose you have a source file named **example.cpp**, located in the **/source** directory. Only the input filename is used, not the full path to the source file. Additionally, it is prefixed with the current timestamp.

- If you specify **-store-check-profile=/tmp**, then the profile will be saved to **/tmp/<ISO8601-like timestamp>-example.cpp.json**

- If you run **clang-tidy** from within **/foo** directory, and specify **-store-check-profile=.**, then the profile will still be saved to **/foo/<ISO8601-like timestamp>-example.cpp.json**

clang-tidy is a clang-based C++ "linter" tool. Its purpose is to provide an extensible framework for diagnosing and fixing typical programming errors, like style violations, interface misuse, or bugs that can be deduced via static analysis. **clang-tidy** is modular and provides a convenient interface for writing new checks.

Using clang-tidy

clang-tidy is a *LibTooling*-based tool, and it's easier to work with if you set up a compile command database for your project (for an example of how to do this, see *How To Setup Tooling For LLVM*). You can also specify compilation options on the command line after --:

```
$ clang-tidy test.cpp -- -Imy_project/include -DMY_DEFINES ...
```

clang-tidy has its own checks and can also run Clang Static Analyzer checks. Each check has a name and the checks to run can be chosen using the **-checks=** option, which specifies a comma-separated list of positive and negative (prefixed with **-**) globs. Positive globs add subsets of checks, and negative globs remove them. For example,

```
$ clang-tidy test.cpp -checks=-*,clang-analyzer-*,-clang-analyzer-cplusplus*
```

will disable all default checks (**-***) and enable all **clang-analyzer-*** checks except for **clang-analyzer-cplusplus*** ones.

The **-list-checks** option lists all the enabled checks. When used without **-checks=**, it shows checks enabled by default. Use **-checks=*** to see all available checks or with any other value of **-checks=** to see which checks are enabled by this value.

There are currently the following groups of checks:

Name prefix	Description
abseil-	Checks related to Abseil library.
altera-	Checks related to OpenCL programming for FPGAs.
android-	Checks related to Android.
boost-	Checks related to Boost library.
bugprone-	Checks that target bug-prone code constructs.
cert-	Checks related to CERT Secure Coding Guidelines.
clang-analyzer-	Clang Static Analyzer checks.

concurrency-	Checks related to concurrent programming (including threads, fibers, coroutines, etc.).
cppcoreguidelines-	Checks related to C++ Core Guidelines.
darwin-	Checks related to Darwin coding conventions.
fuchsia-	Checks related to Fuchsia coding conventions.
google-	Checks related to Google coding conventions.
hicpp-	Checks related to High Integrity C++ Coding Standard.
linuxkernel-	Checks related to the Linux Kernel coding conventions.
llvm-	Checks related to the LLVM coding conventions.
llvmlibc-	Checks related to the LLVM-libc coding standards.
misc-	Checks that we didn’t have a better category for.
modernize-	Checks that advocate usage of modern (currently "modern" means "C++11") language constructs.
mpi-	Checks related to MPI (Message Passing Interface).

objc-	Checks related to Objective-C coding conventions.
openmp-	Checks related to OpenMP API.
performance-	Checks that target performance-related issues.
portability-	Checks that target portability-related issues that don’t relate to any particular coding style.
readability-	Checks that target readability-related issues that don’t relate to any particular coding style.
zircon-	Checks related to Zircon kernel coding conventions.

Clang diagnostics are treated in a similar way as check diagnostics. Clang diagnostics are displayed by **clang-tidy** and can be filtered out using the **-checks=** option. However, the **-checks=** option does not affect compilation arguments, so it cannot turn on Clang warnings which are not already turned on in the build configuration. The **-warnings-as-errors=** option upgrades any warnings emitted under the **-checks=** flag to errors (but it does not enable any checks itself).

Clang diagnostics have check names starting with **clang-diagnostic-**. Diagnostics which have a corresponding warning option, are named **clang-diagnostic-<warning-option>**, e.g. Clang warning controlled by **-Wliteral-conversion** will be reported with check name **clang-diagnostic-literal-conversion**.

The **-fix** flag instructs **clang-tidy** to fix found errors if supported by corresponding checks.

An overview of all the command-line options:

```
$ clang-tidy --help
```

USAGE: clang-tidy [options] <source0> [... <sourceN>]

OPTIONS:

Generic Options:

--help - Display available options (--help-hidden for more)
--help-list - Display list of available options (--help-list-hidden for more)
--version - Display the version of this program

clang-tidy options:

--checks=<string> -
Comma-separated list of globs with optional '-'
prefix. Globs are processed in order of
appearance in the list. Globs without '-'
prefix add checks with matching names to the
set, globs with the '-' prefix remove checks
with matching names from the set of enabled
checks. This option's value is appended to the
value of the 'Checks' option in .clang-tidy
file, if any.
--config=<string> -
Specifies a configuration in YAML/JSON format:
-config="{Checks: '*',
CheckOptions: {x, y}}"
When the value is empty, clang-tidy will
attempt to find a file named .clang-tidy for
each source file in its parent directories.
--config-file=<string> -
Specify the path of .clang-tidy or custom config file:
e.g. --config-file=/some/path/myTidyConfigFile
This option internally works exactly the same way as
--config option after reading specified config file.
Use either --config-file or --config, not both.
--dump-config -
Dumps configuration in the YAML format to
stdout. This option can be used along with a
file name (and '--' if the file is outside of a
project with configured compilation database).

The configuration used for this file will be printed.
Use along with -checks=* to include configuration of all checks.

--enable-check-profile -
Enable per-check timing profiles, and print a report to stderr.

--explain-config -
For each enabled check explains, where it is enabled, i.e. in clang-tidy binary, command line or a specific configuration file.

--export-fixes=<filename> -
YAML file to store suggested fixes in. The stored fixes can be applied to the input source code with clang-apply-replacements.

--extra-arg=<string> - Additional argument to append to the compiler command line.
Can be used several times.

--extra-arg-before=<string> - Additional argument to prepend to the compiler command line.
Can be used several times.

--fix -
Apply suggested fixes. Without -fix-errors clang-tidy will bail out if any compilation errors were found.

--fix-errors -
Apply suggested fixes even if compilation errors were found. If compiler errors have attached fix-its, clang-tidy will apply them as well.

--fix-notes -
If a warning has no fix, but a single fix can be found through an associated diagnostic note, apply the fix.
Specifying this flag will implicitly enable the '--fix' flag.

--format-style=<string> -
Style for formatting code around applied fixes:
- 'none' (default) turns off formatting
- 'file' (literally 'file', not a placeholder) uses .clang-format file in the closest parent directory

- '{ <json> }' specifies options inline, e.g.
-format-style='{BasedOnStyle: llvm, IndentWidth: 8}'
- 'llvm', 'google', 'webkit', 'mozilla'
See clang-format documentation for the up-to-date
information about formatting styles and options.
This option overrides the 'FormatStyle' option in
.clang-tidy file, if any.

--header-filter=<string> -
Regular expression matching the names of the
headers to output diagnostics from. Diagnostics
from the main file of each translation unit are
always displayed.
Can be used together with -line-filter.
This option overrides the 'HeaderFilterRegex'
option in .clang-tidy file, if any.

--line-filter=<string> -
List of files with line ranges to filter the
warnings. Can be used together with
-header-filter. The format of the list is a
JSON array of objects:
[
{"name":"file1.cpp","lines":[[1,3],[5,7]]},
{"name":"file2.h"}
]

--list-checks -
List all enabled checks and exit. Use with
-checks=* to list all available checks.

-load=<plugin> -
Load the dynamic object ''plugin''. This
object should register new static analyzer
or clang-tidy passes. Once loaded, the
object will add new command line options
to run various analyses. To see the new
complete list of passes, use the
:option:'--list-checks' and
:option:'-load' options together.

-p <string> - Build path

--quiet -
Run clang-tidy in quiet mode. This suppresses
printing statistics about ignored warnings and

warnings treated as errors if the respective
options are specified.

--store-check-profile=<prefix> -
By default reports are printed in tabulated
format to stderr. When this option is passed,
these per-TU profiles are instead stored as JSON.

--system-headers - Display the errors from system headers.

--use-color -
Use colors in diagnostics. If not set, colors
will be used if the terminal connected to
standard output supports colors.
This option overrides the 'UseColor' option in
.clang-tidy file, if any.

--verify-config -
Check the config files to ensure each check and
option is recognized.

--vfsoverlay=<filename> -
Overlay the virtual filesystem described by file
over the real file system.

--warnings-as-errors=<string> -
Upgrades warnings to errors. Same format as
'-checks'.
This option's value is appended to the value of
the 'WarningsAsErrors' option in .clang-tidy
file, if any.

-p <build-path> is used to read a compile command database.

For example, it can be a CMake build directory in which a file named compile_commands.json exists (use -DCMAKE_EXPORT_COMPILE_COMMANDS=ON CMake option to get this output). When no build path is specified, a search for compile_commands.json will be attempted through all parent paths of the first input file . See: https://clang.llvm.org/docs/HowToSetupToolingForLLVM.html for an example of setting up Clang Tooling on a source tree.

<source0> ... specify the paths of source files. These paths are looked up in the compile command database. If the path of a file is absolute, it needs to point into CMake's source tree. If the path is relative, the current working directory needs to be in the CMake

source tree and the file must be in a subdirectory of the current working directory. "./" prefixes in the relative files will be automatically removed, but the rest of a relative path must be a suffix of a path in the compile command database.

Configuration files:

clang-tidy attempts to read configuration for each source file from a .clang-tidy file located in the closest parent directory of the source file. If InheritParentConfig is true in a config file, the configuration file in the parent directory (if any exists) will be taken and current config file will be applied on top of the parent one. If any configuration options have a corresponding command-line option, command-line option takes precedence. The effective configuration can be inspected using -dump-config:

```
$ clang-tidy -dump-config
---
Checks:              '-*,some-check'
WarningsAsErrors:    ''
HeaderFilterRegex:   ''
FormatStyle:         none
InheritParentConfig: true
User:                user
CheckOptions:
  some-check.SomeOption: 'some value'
...
```

Suppressing Undesired Diagnostics

clang-tidy diagnostics are intended to call out code that does not adhere to a coding standard, or is otherwise problematic in some way. However, if the code is known to be correct, it may be useful to silence the warning. Some clang-tidy checks provide a check-specific way to silence the diagnostics, e.g. *bugprone-use-after-move* can be silenced by re-initializing the variable after it has been moved out, *bugprone-string-integer-assignment* can be suppressed by explicitly casting the integer to **char**, *readability-implicit-bool-conversion* can also be suppressed by using explicit casts, etc.

If a specific suppression mechanism is not available for a certain warning, or its use is not desired for some reason, **clang-tidy** has a generic mechanism to suppress diagnostics using **NOLINT**, **NOLINTNEXTLINE**, and **NOLINTBEGIN** ... **NOLINTEND** comments.

The **NOLINT** comment instructs **clang-tidy** to ignore warnings on the *same line* (it doesn't apply to a

function, a block of code or any other language construct; it applies to the line of code it is on). If introducing the comment on the same line would change the formatting in an undesired way, the **NOLINTNEXTLINE** comment allows suppressing clang-tidy warnings on the *next line*. The **NOLINTBEGIN** and **NOLINTEND** comments allow suppressing clang-tidy warnings on *multiple lines* (affecting all lines between the two comments).

All comments can be followed by an optional list of check names in parentheses (see below for the formal syntax). The list of check names supports globbing, with the same format and semantics as for enabling checks. Note: negative globs are ignored here, as they would effectively re-activate the warning.

For example:

```
class Foo {
  // Suppress all the diagnostics for the line
  Foo(int param); // NOLINT

  // Consider explaining the motivation to suppress the warning
  Foo(char param); // NOLINT: Allow implicit conversion from ‘char‘, because <some valid reason>

  // Silence only the specified checks for the line
  Foo(double param); // NOLINT(google-explicit-constructor, google-runtime-int)

  // Silence all checks from the ‘google‘ module
  Foo(bool param); // NOLINT(google*)

  // Silence all checks ending with ‘-avoid-c-arrays‘
  int array[10]; // NOLINT(*-avoid-c-arrays)

  // Silence only the specified diagnostics for the next line
  // NOLINTNEXTLINE(google-explicit-constructor, google-runtime-int)
  Foo(bool param);

  // Silence all checks from the ‘google‘ module for the next line
  // NOLINTNEXTLINE(google*)
  Foo(bool param);

  // Silence all checks ending with ‘-avoid-c-arrays‘ for the next line
  // NOLINTNEXTLINE(*-avoid-c-arrays)
  int array[10];
```

```
  // Silence only the specified checks for all lines between the BEGIN and END
  // NOLINTBEGIN(google-explicit-constructor, google-runtime-int)
  Foo(short param);
  Foo(long param);
  // NOLINTEND(google-explicit-constructor, google-runtime-int)

  // Silence all checks from the ‘google‘ module for all lines between the BEGIN and END
  // NOLINTBEGIN(google*)
  Foo(bool param);
  // NOLINTEND(google*)

  // Silence all checks ending with ‘-avoid-c-arrays‘ for all lines between the BEGIN and END
  // NOLINTBEGIN(*-avoid-c-arrays)
  int array[10];
  // NOLINTEND(*-avoid-c-arrays)
};
```

The formal syntax of **NOLINT**, **NOLINTNEXTLINE**, and **NOLINTBEGIN** ... **NOLINTEND** is the following:

lint-comment:
  lint-command
  lint-command lint-args

lint-args:
  (check-name-list **)**

check-name-list:
  check-name
  check-name-list **,** *check-name*

lint-command:
  NOLINT
  NOLINTNEXTLINE
  NOLINTBEGIN
  NOLINTEND

Note that whitespaces between **NOLINT**/**NOLINTNEXTLINE**/**NOLINTBEGIN**/**NOLINTEND** and the opening parenthesis are not allowed (in this case the comment will be treated just as **NOLINT**/**NOLINTNEXTLINE**/**NOLINTBEGIN**/**NOLINTEND**), whereas in the check names

list (inside the parentheses), whitespaces can be used and will be ignored.

All **NOLINTBEGIN** comments must be paired by an equal number of **NOLINTEND** comments. Moreover, a pair of comments must have matching arguments -- for example, **NOLINTBEGIN(check-name)** can be paired with **NOLINTEND(check-name)** but not with **NOLINTEND** *(zero arguments)*. **clang-tidy** will generate a **clang-tidy-nolint** error diagnostic if any **NOLINTBEGIN**/**NOLINTEND** comment violates these requirements.

CLANG-INCLUDE-FIXER

Contents

One of the major nuisances of C++ compared to other languages is the manual management of **#include** directives in any file. **clang-include-fixer** addresses one aspect of this problem by providing an automated way of adding **#include** directives for missing symbols in one translation unit.

While inserting missing **#include**, **clang-include-fixer** adds missing namespace qualifiers to all instances of an unidentified symbol if the symbol is missing some prefix namespace qualifiers.

Setup

To use **clang-include-fixer** two databases are required. Both can be generated with existing tools.

- Compilation database. Contains the compiler commands for any given file in a project and can be generated by CMake, see *How To Setup Tooling For LLVM*.
- Symbol index. Contains all symbol information in a project to match a given identifier to a header file.

Ideally both databases (**compile_commands.json** and **find_all_symbols_db.yaml**) are linked into the root of the source tree they correspond to. Then the **clang-include-fixer** can automatically pick them up if called with a source file from that tree. Note that by default **compile_commands.json** as generated by CMake does not include header files, so only implementation files can be handled by tools.

Creating a Symbol Index From a Compilation Database

The include fixer contains **find-all-symbols**, a tool to create a symbol database in YAML format from a compilation database by parsing all source files listed in it. The following list of commands shows how to set up a database for LLVM, any project built by CMake should follow similar steps.

```
$ cd path/to/llvm-build
$ ninja find-all-symbols // build find-all-symbols tool.
$ ninja clang-include-fixer // build clang-include-fixer tool.
$ ls compile_commands.json # Make sure compile_commands.json exists.
  compile_commands.json
$ path/to/llvm/source/clang-tools-extra/clang-include-fixer/find-all-symbols/tool/run-find-all-symbols.py
  ... wait as clang indexes the code base ...
$ ln -s $PWD/find_all_symbols_db.yaml path/to/llvm/source/ # Link database into the source tree.
$ ln -s $PWD/compile_commands.json path/to/llvm/source/ # Also link compilation database if it's not there alread
$ cd path/to/llvm/source
$ /path/to/clang-include-fixer -db=yaml path/to/file/with/missing/include.cpp
  Added #include "foo.h"
```

Integrate with Vim

To run *clang-include-fixer* on a potentially unsaved buffer in Vim. Add the following key binding to your **.vimrc**:

```
noremap <leader>cf :pyf path/to/llvm/source/clang-tools-extra/clang-include-fixer/tool/clang-include-fixer.py<cr>
```

This enables *clang-include-fixer* for NORMAL and VISUAL mode. Change *<leader>cf* to another binding if you need clang-include-fixer on a different key. The *<leader> key* is a reference to a specific key defined by the mapleader variable and is bound to backslash by default.

Make sure vim can find **clang-include-fixer**:

- Add the path to **clang-include-fixer** to the PATH environment variable.
- Or set **g:clang_include_fixer_path** in vimrc: **let**

g:clang_include_fixer_path=path/to/clang-include-fixer

You can customize the number of headers being shown by setting **let g:clang_include_fixer_maximum_suggested_headers=5**

Customized settings in *.vimrc*:

- **let g:clang_include_fixer_path = "clang-include-fixer"**

  Set clang-include-fixer binary file path.

- **let g:clang_include_fixer_maximum_suggested_headers = 3**

  Set the maximum number of **#includes** to show. Default is 3.

- **let g:clang_include_fixer_increment_num = 5**

  Set the increment number of #includes to show every time when pressing **m**. Default is 5.

- **let g:clang_include_fixer_jump_to_include = 0**

  Set to 1 if you want to jump to the new inserted **#include** line. Default is 0.

- **let g:clang_include_fixer_query_mode = 0**

  Set to 1 if you want to insert **#include** for the symbol under the cursor. Default is 0. Compared to normal mode, this mode won't parse the source file and only search the symbol from database, which is faster than normal mode.

  See **clang-include-fixer.py** for more details.

Integrate with Emacs

To run *clang-include-fixer* on a potentially unsaved buffer in Emacs. Ensure that Emacs finds **clang-include-fixer.el** by adding the directory containing the file to the **load-path** and requiring the *clang-include-fixer* in your **.emacs**:

```
(add-to-list 'load-path "path/to/llvm/source/clang-tools-extra/clang-include-fixer/tool/"
(require 'clang-include-fixer)
```

Within Emacs the tool can be invoked with the command **M-x clang-include-fixer**. This will

insert the header that defines the first undefined symbol; if there is more than one header that would define the symbol, the user is prompted to select one.

To include the header that defines the symbol at point, run **M-x clang-include-fixer-at-point**.

Make sure Emacs can find **clang-include-fixer**:

- Either add the parent directory of **clang-include-fixer** to the PATH environment variable, or customize the Emacs user option **clang-include-fixer-executable** to point to the file name of the program.

How it Works

To get the most information out of Clang at parse time, **clang-include-fixer** runs in tandem with the parse and receives callbacks from Clang's semantic analysis. In particular it reuses the existing support for typo corrections. Whenever Clang tries to correct a potential typo it emits a callback to the include fixer which then looks for a corresponding file. At this point rich lookup information is still available, which is not available in the AST at a later stage.

The identifier that should be typo corrected is then sent to the database, if a header file is returned it is added as an include directive at the top of the file.

Currently **clang-include-fixer** only inserts a single include at a time to avoid getting caught in follow-up errors. If multiple *#include* additions are desired the program can be rerun until a fix-point is reached.

MODULARIZE USER'S MANUAL

Modularize Usage

modularize [<modularize-options>] [<module-map>|<include-files-list>]* [<front-end-options>...]

<modularize-options> is a place-holder for options specific to modularize, which are described below in *Modularize Command Line Options*.

<module-map> specifies the path of a file name for an existing module map. The module map must be well-formed in terms of syntax. Modularize will extract the header file names from the map. Only normal headers are checked, assuming headers marked "private", "textual", or "exclude" are not to be checked as a top-level include, assuming they either are included by other headers which are checked, or they are not suitable for modules.

<include-files-list> specifies the path of a file name for a file containing the newline-separated list of headers to check with respect to each other. Lines beginning with '#' and empty lines are ignored. Header file names followed by a colon and other space-separated file names will include those extra

files as dependencies. The file names can be relative or full paths, but must be on the same line. For example:

```
header1.h
header2.h
header3.h: header1.h header2.h
```

Note that unless a **-prefix (header path)** option is specified, non-absolute file paths in the header list file will be relative to the header list file directory. Use -prefix to specify a different directory.

<front-end-options> is a place-holder for regular Clang front-end arguments, which must follow the <include-files-list>. Note that by default, modularize assumes .h files contain C++ source, so if you are using a different language, you might need to use a **-x** option to tell Clang that the header contains another language, i.e.: **-x c**

Note also that because modularize does not use the clang driver, you will likely need to pass in additional compiler front-end arguments to match those passed in by default by the driver.

Modularize Command Line Options

-prefix=<header-path>
Prepend the given path to non-absolute file paths in the header list file. By default, headers are assumed to be relative to the header list file directory. Use **-prefix** to specify a different directory.

-module-map-path=<module-map-path>
Generate a module map and output it to the given file. See the description in *Module Map Generation*.

-problem-files-list=<problem-files-list-file-name>
For use only with module map assistant. Input list of files that have problems with respect to modules. These will still be included in the generated module map, but will be marked as "excluded" headers.

-root-module=<root-name>
Put modules generated by the -module-map-path option in an enclosing module with the given name. See the description in *Module Map Generation*.

-block-check-header-list-only
Limit the #include-inside-extern-or-namespace-block check to only those headers explicitly listed

in the header list. This is a work-around for avoiding error messages for private includes that purposefully get included inside blocks.

-no-coverage-check
Don't do the coverage check for a module map.

-coverage-check-only
Only do the coverage check for a module map.

-display-file-lists
Display lists of good files (no compile errors), problem files, and a combined list with problem files preceded by a '#'. This can be used to quickly determine which files have problems. The latter combined list might be useful in starting to modularize a set of headers. You can start with a full list of headers, use -display-file-lists option, and then use the combined list as your intermediate list, uncommenting-out headers as you fix them.

modularize is a standalone tool that checks whether a set of headers provides the consistent definitions required to use modules. For example, it detects whether the same entity (say, a NULL macro or size_t typedef) is defined in multiple headers or whether a header produces different definitions under different circumstances. These conditions cause modules built from the headers to behave poorly, and should be fixed before introducing a module map.

modularize also has an assistant mode option for generating a module map file based on the provided header list. The generated file is a functional module map that can be used as a starting point for a module.map file.

Getting Started

To build from source:

1. Read *Getting Started with the LLVM System* and *Clang Tools Documentation* for information on getting sources for LLVM, Clang, and Clang Extra Tools.

2. *Getting Started with the LLVM System* and *Building LLVM with CMake* give directions for how to build. With sources all checked out into the right place the LLVM build will build Clang Extra Tools and their dependencies automatically.

   - If using CMake, you can also use the **modularize** target to build just the modularize tool and its dependencies.

Before continuing, take a look at *Modularize Usage* to see how to invoke modularize.

What Modularize Checks

Modularize will check for the following:

- Duplicate global type and variable definitions
- Duplicate macro definitions
- Macro instances, ’defined(macro)’, or #if, #elif, #ifdef, #ifndef conditions that evaluate differently in a header
- #include directives inside ’extern "C/C++" {}’ or ’namespace (name) {}’ blocks
- Module map header coverage completeness (in the case of a module map input only)

Modularize will do normal C/C++ parsing, reporting normal errors and warnings, but will also report special error messages like the following:

```
error: ’(symbol)’ defined at multiple locations:
  (file):(row):(column)
  (file):(row):(column)

error: header ’(file)’ has different contents depending on how it was included
```

The latter might be followed by messages like the following:

```
note: ’(symbol)’ in (file) at (row):(column) not always provided
```

Checks will also be performed for macro expansions, defined(macro) expressions, and preprocessor conditional directives that evaluate inconsistently, and can produce error messages like the following:

```
 (...)/SubHeader.h:11:5:
#if SYMBOL == 1
    ^
error: Macro instance ’SYMBOL’ has different values in this header,
       depending on how it was included.
  ’SYMBOL’ expanded to: ’1’ with respect to these inclusion paths:
    (...)/Header1.h
      (...)/SubHeader.h
(...)/SubHeader.h:3:9:
```

```
#define SYMBOL 1
      ^
Macro defined here.
 'SYMBOL' expanded to: '2' with respect to these inclusion paths:
   (...)/Header2.h
      (...)/SubHeader.h
(...)/SubHeader.h:7:9:
#define SYMBOL 2
      ^
Macro defined here.
```

Checks will also be performed for '#include' directives that are nested inside 'extern "C/C++" {}' or 'namespace (name) {}' blocks, and can produce error message like the following:

```
IncludeInExtern.h:2:3:
#include "Empty.h"
^
error: Include directive within extern "C" {}.
IncludeInExtern.h:1:1:
extern "C" {
^
The "extern "C" {}" block is here.
```

Module Map Coverage Check

The coverage check uses the Clang library to read and parse the module map file. Starting at the module map file directory, or just the include paths, if specified, it will collect the names of all the files it considers headers (no extension, .h, or .inc--if you need more, modify the isHeader function). It then compares the headers against those referenced in the module map, either explicitly named, or implicitly named via an umbrella directory or umbrella file, as parsed by the ModuleMap object. If headers are found which are not referenced or covered by an umbrella directory or file, warning messages will be produced, and this program will return an error code of 1. If no problems are found, an error code of 0 is returned.

Note that in the case of umbrella headers, this tool invokes the compiler to preprocess the file, and uses a callback to collect the header files included by the umbrella header or any of its nested includes. If any front end options are needed for these compiler invocations, these can be included on the command line after the module map file argument.

Warning message have the form:

```
warning: module.modulemap does not account for file: Level3A.h
```

Note that for the case of the module map referencing a file that does not exist, the module map parser in Clang will (at the time of this writing) display an error message.

To limit the checks **modularize** does to just the module map coverage check, use the **-coverage-check-only option**.

For example:

```
modularize -coverage-check-only module.modulemap
```

Module Map Generation

If you specify the **-module-map-path=<module map file>**, **modularize** will output a module map based on the input header list. A module will be created for each header. Also, if the header in the header list is a partial path, a nested module hierarchy will be created in which a module will be created for each subdirectory component in the header path, with the header itself represented by the innermost module. If other headers use the same subdirectories, they will be enclosed in these same modules also.

For example, for the header list:

```
SomeTypes.h
SomeDecls.h
SubModule1/Header1.h
SubModule1/Header2.h
SubModule2/Header3.h
SubModule2/Header4.h
SubModule2.h
```

The following module map will be generated:

```
// Output/NoProblemsAssistant.txt
// Generated by: modularize -module-map-path=Output/NoProblemsAssistant.txt \
    -root-module=Root NoProblemsAssistant.modularize

module SomeTypes {
  header "SomeTypes.h"
  export *
}
module SomeDecls {
  header "SomeDecls.h"
  export *
```

```
}
module SubModule1 {
  module Header1 {
    header "SubModule1/Header1.h"
    export *
  }
  module Header2 {
    header "SubModule1/Header2.h"
    export *
  }
}
module SubModule2 {
  module Header3 {
    header "SubModule2/Header3.h"
    export *
  }
  module Header4 {
    header "SubModule2/Header4.h"
    export *
  }
  header "SubModule2.h"
  export *
}
```

An optional **-root-module=<root-name>** option can be used to cause a root module to be created which encloses all the modules.

An optional **-problem-files-list=<problem-file-name>** can be used to input a list of files to be excluded, perhaps as a temporary stop-gap measure until problem headers can be fixed.

For example, with the same header list from above:

```
// Output/NoProblemsAssistant.txt
// Generated by: modularize -module-map-path=Output/NoProblemsAssistant.txt \
     -root-module=Root NoProblemsAssistant.modularize

module Root {
  module SomeTypes {
    header "SomeTypes.h"
    export *
```

```
 }
 module SomeDecls {
  header "SomeDecls.h"
  export *
 }
 module SubModule1 {
  module Header1 {
   header "SubModule1/Header1.h"
   export *
  }
  module Header2 {
   header "SubModule1/Header2.h"
   export *
  }
 }
 module SubModule2 {
  module Header3 {
   header "SubModule2/Header3.h"
   export *
  }
  module Header4 {
   header "SubModule2/Header4.h"
   export *
  }
  header "SubModule2.h"
  export *
 }
}
```

Note that headers with dependents will be ignored with a warning, as the Clang module mechanism doesn't support headers the rely on other headers to be included first.

The module map format defines some keywords which can't be used in module names. If a header has one of these names, an underscore ('_') will be prepended to the name. For example, if the header name is **header.h**, because **header** is a keyword, the module name will be **_header**. For a list of the module map keywords, please see: *Lexical structure*

PP-TRACE USER'S MANUAL

pp-trace is a standalone tool that traces preprocessor activity. It's also used as a test of Clang's PPCallbacks interface. It runs a given source file through the Clang preprocessor, displaying selected

information from callback functions overridden in a *PPCallbacks* derivation. The output is in a high-level YAML format, described in *pp-trace Output Format*.

pp-trace Usage

Command Line Format

pp-trace [<pp-trace-options>] <source-file> [-- <front-end-options>]

<pp-trace-options> is a place-holder for options specific to pp-trace, which are described below in *Command Line Options*.

<source-file> specifies the source file to run through the preprocessor.

<front-end-options> is a place-holder for regular *Clang Compiler Options*, which must follow the <source-file>.

Command Line Options

-callbacks <comma-separated-globs>

This option specifies a comma-separated list of globs describing the list of callbacks that should be traced. Globs are processed in order of appearance. Positive globs add matched callbacks to the set, netative globs (those with the '-' prefix) remove callacks from the set.

- FileChanged
- FileSkipped
- InclusionDirective
- moduleImport
- EndOfMainFile
- Ident
- PragmaDirective
- PragmaComment
- PragmaDetectMismatch

- PragmaDebug
- PragmaMessage
- PragmaDiagnosticPush
- PragmaDiagnosticPop
- PragmaDiagnostic
- PragmaOpenCLExtension
- PragmaWarning
- PragmaWarningPush
- PragmaWarningPop
- MacroExpands
- MacroDefined
- MacroUndefined
- Defined
- SourceRangeSkipped
- If
- Elif
- Ifdef
- Ifndef
- Else
- Endif

-output <output-file>

By default, pp-trace outputs the trace information to stdout. Use this option to output the trace information to a file.

pp-trace Output Format

The pp-trace output is formatted as YAML. See *https://yaml.org/* for general YAML information. It's arranged as a sequence of information about the callback call, including the callback name and argument information, for example::

```
---
- Callback: Name
  Argument1: Value1
  Argument2: Value2
(etc.)
...
```

With real data::

```
---
- Callback: FileChanged
  Loc: "c:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-include.cpp:1:1"
  Reason: EnterFile
  FileType: C_User
  PrevFID: (invalid)
  (etc.)
- Callback: FileChanged
  Loc: "D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-include.cpp:5:1"
  Reason: ExitFile
  FileType: C_User
  PrevFID: "D:/Clang/llvm/clang-tools-extra/test/pp-trace/Input/Level1B.h"
- Callback: EndOfMainFile
...
```

In all but one case (MacroDirective) the "Argument" scalars have the same name as the argument in the corresponding PPCallbacks callback function.

Callback Details

The following sections describe the purpose and output format for each callback.

Click on the callback name in the section heading to see the Doxygen documentation for the callback.

The argument descriptions table describes the callback argument information displayed.

The Argument Name field in most (but not all) cases is the same name as the callback function parameter.

The Argument Value Syntax field describes the values that will be displayed for the argument value. It uses an ad hoc representation that mixes literal and symbolic representations. Enumeration member symbols are shown as the actual enum member in a (member1|member2|...) form. A name in parentheses can either represent a place holder for the described value, or confusingly, it might be a literal, such as (null), for a null pointer. Locations are shown as quoted only to avoid confusing the documentation generator.

The Clang C++ Type field is the type from the callback function declaration.

The description describes the argument or what is displayed for it.

Note that in some cases, such as when a structure pointer is an argument value, only some key member or members are shown to represent the value, instead of trying to display all members of the structure.

FileChanged **Callback**

FileChanged is called when the preprocessor enters or exits a file, both the top level file being compiled, as well as any #include directives. It will also be called as a result of a system header pragma or in internal renaming of a file.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
Loc	"(file):(line):(col)"	SourceLocation	The location of the directive.
Reason	(EnterFile|ExitFile|SystemHeaderPragma|RenameFile)	PPCallbacks::FileChangeReason	Reason for change.
FileType	(C_User|C_System|C_ExternCSystem)	SrcMgr::CharacteristicKind	Include type.
PrevFID	((file)|(invalid))	FileID	Previous file, if

| | | | |any. |
+---+---+---+---

Example::

```
- Callback: FileChanged
  Loc: "D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-include.cpp:1:1"
  Reason: EnterFile
  FileType: C_User
  PrevFID: (invalid)
```

FileSkipped Callback

FileSkipped is called when a source file is skipped as the result of header guard optimization.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
ParentFile	("(file)" or (null))	const FileEntry	The file that #included the skipped file.
FilenameTok	(token)	const Token	The token in ParentFile that indicates the skipped file.
FileType	(C_User|C_System|C_ExternCSystem)	SrcMgr::CharacteristicKind	The file type.

Example::

```
- Callback: FileSkipped
  ParentFile: "/path/filename.h"
  FilenameTok: "filename.h"
  FileType: C_User
```

InclusionDirective **Callback**

InclusionDirective is called when an inclusion directive of any kind (#include</code>, #import</code>, etc.) has been processed, regardless of whether the inclusion will actually result in an inclusion.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
HashLoc	"(file):(line):(col)"	SourceLocation	The location of the '#' that starts the inclusion directive.
IncludeTok	(token)	const Token	The token that indicates the kind of inclusion directive, e.g., 'include' or 'import'.
FileName	"(file)"	StringRef	The name of the file being included, as written in the source code.
IsAngled	(true|false)	bool	Whether the file name was enclosed in angle brackets; otherwise, it was enclosed in quotes.
FilenameRange	"(file)"	CharSourceRange	The character range of the quotes or angle brackets for the written file name.
File	"(file)"	const	The actual file that

		FileEntry	may be included by this inclusion directive.
SearchPath	"(path)"	StringRef	Contains the search path which was used to find the file in the file system.
RelativePath	"(path)"	StringRef	The path relative to SearchPath, at which the include file was found.
Imported	((module name)|(null))	const Module	The module, whenever an inclusion directive was automatically turned into a module import or null otherwise.

Example::

```
- Callback: InclusionDirective
  HashLoc: "D:/Clang/llvmnewmod/clang-tools-extra/test/pp-trace/pp-trace-include.cpp:4:1"
  IncludeTok: include
  FileName: "Input/Level1B.h"
  IsAngled: false
  FilenameRange: "Input/Level1B.h"
  File: "D:/Clang/llvmnewmod/clang-tools-extra/test/pp-trace/Input/Level1B.h"
  SearchPath: "D:/Clang/llvmnewmod/clang-tools-extra/test/pp-trace"
  RelativePath: "Input/Level1B.h"
  Imported: (null)
```

moduleImport **Callback**

moduleImport is called when there was an explicit module-import syntax.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
ImportLoc	"(file):(line):(col)"	SourceLocation	The location of import directive token.
Path	"(path)"	ModuleIdPath	The identifiers (and their locations) of the module "path".
Imported	((module name)|(null))	const Module	The imported module; can be null if importing failed.

Example::

```
- Callback: moduleImport
  ImportLoc: "d:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-modules.cpp:4:2"
  Path: [{Name: Level1B, Loc: "d:/Clang/llvmnewmod/clang-tools-extra/test/pp-trace/pp-trace-modules.cpp:4:9"},
  Imported: Level2B
```

EndOfMainFile **Callback**

EndOfMainFile is called when the end of the main file is reached.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
(no arguments)			

Example::

```
- Callback: EndOfMainFile
```

Ident **Callback**

Ident is called when a #ident or #sccs directive is read.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
Loc	"(file):(line):(col)"	SourceLocation	The location of the directive.
str	(name)	const std::string	The text of the directive.

Example::

```
- Callback: Ident
  Loc: "D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-ident.cpp:3:1"
  str: "$Id$"
```

PragmaDirective **Callback**

PragmaDirective is called when start reading any pragma directive.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
Loc	"(file):(line):(col)"	SourceLocation	The location of the directive.
Introducer	(PIK_HashPragma|PIK__Pragma|PIK___pragma)	PragmaIntroducerKind	The type of the pragma directive.

Example::

```
- Callback: PragmaDirective
  Loc: "D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-pragma.cpp:3:1"
  Introducer: PIK_HashPragma
```

PragmaComment **Callback**

PragmaComment is called when a #pragma comment directive is read.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
Loc	"(file):(line):(col)"	SourceLocation	The location of the directive.
Kind	((name)|(null))	const IdentifierInfo	The comment kind symbol.
Str	(message directive)	const std::string	The comment message directive.

Example::

```
- Callback: PragmaComment
  Loc: "D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-pragma.cpp:3:1"
  Kind: library
  Str: kernel32.lib
```

PragmaDetectMismatch **Callback**

PragmaDetectMismatch is called when a #pragma detect_mismatch directive is read.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description

Loc	"(file):(line):(col)"	SourceLocation	The location of the directive.
Name	"(name)"	const std::string	The name.
Value	(string)	const std::string	The value.

Example::

```
- Callback: PragmaDetectMismatch
  Loc: "D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-pragma.cpp:3:1"
  Name: name
  Value: value
```

PragmaDebug **Callback**

PragmaDebug is called when a #pragma clang __debug directive is read.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
Loc	"(file):(line):(col)"	SourceLocation	The location of the directive.
DebugType	(string)	StringRef	Indicates type of debug message.

Example::

```
- Callback: PragmaDebug
  Loc: "D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-pragma.cpp:3:1"
  DebugType: warning
```

PragmaMessage **Callback**

PragmaMessage is called when a #pragma message directive is read.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
Loc	"(file):(line):(col)"	SourceLocation	The location of the directive.
Namespace	(name)	StringRef	The namespace of the message directive.
Kind	(PMK_Message|PMK_Warning|PMK_Error)	PPCallbacks::PragmaMessageKind	The type of the message directive.
Str	(string)	StringRef	The text of the message directive.

Example::

```
- Callback: PragmaMessage
  Loc: "D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-pragma.cpp:3:1"
  Namespace: "GCC"
  Kind: PMK_Message
  Str: The message text.
```

PragmaDiagnosticPush **Callback**

PragmaDiagnosticPush is called when a #pragma gcc diagnostic push directive is read.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
Loc	"(file):(line):(col)"	SourceLocation	The location of

			the directive.
Namespace	(name)	StringRef	Namespace name.

Example::

```
- Callback: PragmaDiagnosticPush
  Loc: "D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-pragma.cpp:3:1"
  Namespace: "GCC"
```

PragmaDiagnosticPop **Callback**

PragmaDiagnosticPop is called when a #pragma gcc diagnostic pop directive is read.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
Loc	"(file):(line):(col)"	SourceLocation	The location of the directive.
Namespace	(name)	StringRef	Namespace name.

Example::

```
- Callback: PragmaDiagnosticPop
  Loc: "D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-pragma.cpp:3:1"
  Namespace: "GCC"
```

PragmaDiagnostic **Callback**

PragmaDiagnostic is called when a #pragma gcc diagnostic directive is read.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
Loc	"(file):(line):(col)"	SourceLocation	The location of the directive.
Namespace	(name)	StringRef	Namespace name.
mapping	(0|MAP_IGNORE|MAP_WARNING|MAP_ERROR|MAP_FATAL)	diag::Severity	Mapping type.
Str	(string)	StringRef	Warning/error name.

Example::

```
- Callback: PragmaDiagnostic
  Loc: "D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-pragma.cpp:3:1"
  Namespace: "GCC"
  mapping: MAP_WARNING
  Str: WarningName
```

PragmaOpenCLExtension **Callback**

PragmaOpenCLExtension is called when OpenCL extension is either disabled or enabled with a pragma.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
NameLoc	"(file):(line):(col)"	SourceLocation	The location of the name.
Name	(name)	const IdentifierInfo	Name symbol.

StateLoc	"(file):(line):(col)"	SourceLocation	The location of the state.
State	(1|0)	unsigned	Enabled/disabled state.

Example::

```
- Callback: PragmaOpenCLExtension
  NameLoc: "D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-pragma.cpp:3:10"
  Name: Name
  StateLoc: "D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-pragma.cpp:3:18"
  State: 1
```

PragmaWarning **Callback**

PragmaWarning is called when a #pragma warning directive is read.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
Loc	"(file):(line):(col)"	SourceLocation	The location of the directive.
WarningSpec	(string)	StringRef	The warning specifier.
Ids	[(number)[, ...]]	ArrayRef<int>	The warning numbers.

Example::

```
- Callback: PragmaWarning
  Loc: "D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-pragma.cpp:3:1"
  WarningSpec: disable
  Ids: 1,2,3
```

PragmaWarningPush **Callback**
PragmaWarningPush is called when a #pragma warning(push) directive is read.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
Loc	"(file):(line):(col)"	SourceLocation	The location of the directive.
Level	(number)	int	Warning level.

Example::

```
- Callback: PragmaWarningPush
  Loc: "D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-pragma.cpp:3:1"
  Level: 1
```

PragmaWarningPop **Callback**
PragmaWarningPop is called when a #pragma warning(pop) directive is read.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
Loc	"(file):(line):(col)"	SourceLocation	The location of the directive.

Example::

```
- Callback: PragmaWarningPop
  Loc: "D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-pragma.cpp:3:1"
```

MacroExpands **Callback**

MacroExpands is called when ::HandleMacroExpandedIdentifier when a macro invocation is found.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
MacroNameTok	(token)	const Token	The macro name token.
MacroDirective	(MD_Define|MD_Undefine|MD_Visibility)	const MacroDirective	The kind of macro directive from the MacroDirective structure.
Range	["(file):(line):(col)", "(file):(line):(col)"]	SourceRange	The source range for the expansion.
Args	[(name)|(number)|<(token name)>[, ...]]	const MacroArgs	The argument tokens. Names and numbers are literal, everything else is of the form '<' tokenName '>'.

Example::

```
- Callback: MacroExpands
  MacroNameTok: X_IMPL
  MacroDirective: MD_Define
  Range: [(nonfile), (nonfile)]
  Args: [a <plus> y, b]
```

MacroDefined **Callback**

MacroDefined is called when a macro definition is seen.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
MacroNameTok	(token)	const Token	The macro name token.
MacroDirective	(MD_Define|MD_Undefine|MD_Visibility)	const MacroDirective	The kind of macro directive from the MacroDirective structure.

Example::

```
- Callback: MacroDefined
  MacroNameTok: X_IMPL
  MacroDirective: MD_Define
```

MacroUndefined **Callback**

MacroUndefined is called when a macro #undef is seen.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
MacroNameTok	(token)	const Token	The macro name token.
MacroDirective	(MD_Define|MD_Undefine|MD_Visibility)	const MacroDirective	The kind of macro directive from the MacroDirective structure.

Example::

```
- Callback: MacroUndefined
  MacroNameTok: X_IMPL
  MacroDirective: MD_Define
```

Defined **Callback**

Defined is called when the ’defined’ operator is seen.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
MacroNameTok	(token)	const Token	The macro name token.
MacroDirective	(MD_Define|MD_Undefine|MD_Visibility)	const MacroDirective	The kind of macro directive from the MacroDirective structure.
Range	["(file):(line):(col)", "(file):(line):(col)"]	SourceRange	The source range for the directive.

Example::

```
- Callback: Defined
  MacroNameTok: MACRO
  MacroDirective: (null)
  Range: ["D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-macro.cpp:8:5", "D:/Clang/llvm/clang-tools-extra
```

SourceRangeSkipped **Callback**

SourceRangeSkipped is called when a source range is skipped.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
Range	["(file):(line):(col)","(file):(line):(col)"]	SourceRange	The source range skipped.

Example::

```
- Callback: SourceRangeSkipped
  Range: [":/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-macro.cpp:8:2", ":/Clang/llvm/clang-tools-extra/tes
```

If **Callback**

If is called when an #if is seen.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
Loc	"(file):(line):(col)"	SourceLocation	The location of the directive.
ConditionRange	["(file):(line):(col)","(file):(line):(col)"]	SourceRange	The source range for the condition.
ConditionValue	(true|false)	bool	The condition value.

Example::

```
- Callback: If
  Loc: "D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-macro.cpp:8:2"
  ConditionRange: ["D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-macro.cpp:8:4", "D:/Clang/llvm/clang-
  ConditionValue: false
```

Elif **Callback**

Elif is called when an #elif is seen.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
Loc	"(file):(line):(col)"	SourceLocation	The location of the directive.
ConditionRange	["(file):(line):(col)", "(file):(line):(col)"]	SourceRange	The source range for the condition.
ConditionValue	(true|false)	bool	The condition value.
IfLoc	"(file):(line):(col)"	SourceLocation	The location of the directive.

Example::

```
- Callback: Elif
  Loc: "D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-macro.cpp:10:2"
  ConditionRange: ["D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-macro.cpp:10:4", "D:/Clang/llvm/clang
  ConditionValue: false
  IfLoc: "D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-macro.cpp:8:2"
```

Ifdef **Callback**

Ifdef is called when an #ifdef is seen.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
Loc	"(file):(line):(col)"	SourceLocation	The location of the directive.

MacroNameTok	(token)	const Token	The macro name token.
MacroDirective	(MD_Define|MD_Undefine|MD_Visibility)	const MacroDirective	The kind of macro directive from the MacroDirective structure.

Example::

```
- Callback: Ifdef
  Loc: "D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-conditional.cpp:3:1"
  MacroNameTok: MACRO
  MacroDirective: MD_Define
```

Ifndef **Callback**

Ifndef is called when an #ifndef is seen.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
Loc	"(file):(line):(col)"	SourceLocation	The location of the directive.
MacroNameTok	(token)	const Token	The macro name token.
MacroDirective	(MD_Define|MD_Undefine|MD_Visibility)	const MacroDirective	The kind of macro directive from the MacroDirective structure.

Example::

```
- Callback: Ifndef
  Loc: "D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-conditional.cpp:3:1"
  MacroNameTok: MACRO
  MacroDirective: MD_Define
```

Else **Callback**

Else is called when an #else is seen.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
Loc	"(file):(line):(col)"	SourceLocation	The location of the else directive.
IfLoc	"(file):(line):(col)"	SourceLocation	The location of the if directive.

Example::

```
- Callback: Else
  Loc: "D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-macro.cpp:10:2"
  IfLoc: "D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-macro.cpp:8:2"
```

Endif **Callback**

Endif is called when an #endif is seen.

Argument descriptions:

Argument Name	Argument Value Syntax	Clang C++ Type	Description
Loc	"(file):(line):(col)"	SourceLocation	The location of the endif

			directive.
IfLoc	"(file):(line):(col)"	SourceLocation	The location of the if directive.

Example::

```
- Callback: Endif
  Loc: "D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-macro.cpp:10:2"
  IfLoc: "D:/Clang/llvm/clang-tools-extra/test/pp-trace/pp-trace-macro.cpp:8:2"
```

Building pp-trace

To build from source:

1. Read *Getting Started with the LLVM System* and *Clang Tools Documentation* for information on getting sources for LLVM, Clang, and Clang Extra Tools.

2. *Getting Started with the LLVM System* and *Building LLVM with CMake* give directions for how to build. With sources all checked out into the right place the LLVM build will build Clang Extra Tools and their dependencies automatically.

   - If using CMake, you can also use the **pp-trace** target to build just the pp-trace tool and its dependencies.

CLANG-RENAME

Contents

See also:

clang-rename is a C++ refactoring tool. Its purpose is to perform efficient renaming actions in large-scale projects such as renaming classes, functions, variables, arguments, namespaces etc.

The tool is in a very early development stage, so you might encounter bugs and crashes. Submitting reports with information about how to reproduce the issue to *the LLVM bugtracker* will definitely help the project. If you have any ideas or suggestions, you might want to put a feature request there.

Using Clang-Rename

clang-rename is a *LibTooling*-based tool, and it's easier to work with if you set up a compile command database for your project (for an example of how to do this see *How To Setup Tooling For LLVM*). You can also specify compilation options on the command line after --:

```
$ clang-rename -offset=42 -new-name=foo test.cpp -- -Imy_project/include -DMY_DEFINES ...
```

To get an offset of a symbol in a file run

```
$ grep -FUbo 'foo' file.cpp
```

The tool currently supports renaming actions inside a single translation unit only. It is planned to extend the tool's functionality to support multi-TU renaming actions in the future.

clang-rename also aims to be easily integrated into popular text editors, such as Vim and Emacs, and improve the workflow of users.

Although a command line interface exists, it is highly recommended to use the text editor interface instead for better experience.

You can also identify one or more symbols to be renamed by giving the fully qualified name:

```
$ clang-rename -qualified-name=foo -new-name=bar test.cpp
```

Renaming multiple symbols at once is supported, too. However, **clang-rename** doesn't accept both *-offset* and *-qualified-name* at the same time. So, you can either specify multiple *-offset* or *-qualified-name*.

```
$ clang-rename -offset=42 -new-name=bar1 -offset=150 -new-name=bar2 test.cpp
```

or

```
$ clang-rename -qualified-name=foo1 -new-name=bar1 -qualified-name=foo2 -new-name=bar2 test.cpp
```

Alternatively, {offset | qualified-name} / new-name pairs can be put into a YAML file:

```
---
- Offset:       42
  NewName:      bar1
- Offset:       150
  NewName:      bar2
...
```

or

```
---
- QualifiedName:  foo1
  NewName:      bar1
- QualifiedName:  foo2
  NewName:      bar2
...
```

That way you can avoid spelling out all the names as command line arguments:

```
$ clang-rename -input=test.yaml test.cpp
```

clang-rename offers the following options:

```
$ clang-rename --help
USAGE: clang-rename [subcommand] [options] <source0> [... <sourceN>]

OPTIONS:

Generic Options:

  -help                 - Display available options (-help-hidden for more)
  -help-list            - Display list of available options (-help-list-hidden for more)
  -version              - Display the version of this program

clang-rename common options:

  -export-fixes=<filename>   - YAML file to store suggested fixes in.
  -extra-arg=<string>        - Additional argument to append to the compiler command line
                               Can be used several times.
  -extra-arg-before=<string> - Additional argument to prepend to the compiler command line
                               Can be used several times.
```

-force - Ignore nonexistent qualified names.
-i - Overwrite edited <file>s.
-input=<string> - YAML file to load oldname-newname pairs from.
-new-name=<string> - The new name to change the symbol to.
-offset=<uint> - Locates the symbol by offset as opposed to <line>:<column>.
-p <string> - Build path
-pl - Print the locations affected by renaming to stderr.
-pn - Print the found symbol's name prior to renaming to stderr.
-qualified-name=<string> - The fully qualified name of the symbol.

Vim Integration

You can call **clang-rename** directly from Vim! To set up **clang-rename** integration for Vim see *clang/tools/clang-rename/clang-rename.py*.

Please note that **you have to save all buffers, in which the replacement will happen before running the tool**.

Once installed, you can point your cursor to symbols you want to rename, press *<leader>cr* and type new desired name. The *<leader> key* is a reference to a specific key defined by the mapleader variable and is bound to backslash by default.

Emacs Integration

You can also use **clang-rename** while using Emacs! To set up **clang-rename** integration for Emacs see *clang-rename/tool/clang-rename.el*.

Once installed, you can point your cursor to symbols you want to rename, press *M-X*, type *clang-rename* and new desired name.

Please note that **you have to save all buffers, in which the replacement will happen before running the tool**.

CLANG-DOC

Contents

- *Configuration*
  - *Options*

clang-doc is a tool for generating C and C++ documentation from source code and comments.

The tool is in a very early development stage, so you might encounter bugs and crashes. Submitting reports with information about how to reproduce the issue to *the LLVM bug tracker* will definitely help the project. If you have any ideas or suggestions, please to put a feature request there.

Use

clang-doc is a *LibTooling*-based tool, and so requires a compile command database for your project (for an example of how to do this see *How To Setup Tooling For LLVM*).

By default, the tool will run on all files listed in the given compile commands database:

```
$ clang-doc /path/to/compile_commands.json
```

The tool can also be used on a single file or multiple files if a build path is passed with the **-p** flag.

```
$ clang-doc /path/to/file.cpp -p /path/to/build
```

Output

clang-doc produces a directory of documentation. One file is produced for each namespace and record in the project source code, containing all documentation (including contained functions, methods, and enums) for that item.

The top-level directory is configurable through the **output** flag:

```
$ clang-doc -output=output/directory/ compile_commands.json
```

Configuration

Configuration for **clang-doc** is currently limited to command-line options. In the future, it may develop the ability to use a configuration file, but no such efforts are currently in progress.

Options

clang-doc offers the following options:

```
$ clang-doc --help
USAGE: clang-doc [options] <source0> [... <sourceN>]

OPTIONS:

Generic Options:

  -help                - Display available options (-help-hidden for more)
  -help-list           - Display list of available options (-help-list-hidden for more)
  -version             - Display the version of this program

clang-doc options:

  --doxygen              - Use only doxygen-style comments to generate docs.
  --extra-arg=<string>   - Additional argument to append to the compiler command line
                           Can be used several times.
  --extra-arg-before=<string> - Additional argument to prepend to the compiler command line
                           Can be used several times.
  --format=<value>       - Format for outputted docs.
    =yaml                -   Documentation in YAML format.
    =md                  -   Documentation in MD format.
    =html                -   Documentation in HTML format.
  --ignore-map-errors    - Continue if files are not mapped correctly.
  --output=<string>      - Directory for outputting generated files.
  -p <string>            - Build path
  --project-name=<string>  - Name of project.
  --public               - Document only public declarations.
  --repository=<string>  -
                           URL of repository that hosts code.
                           Used for links to definition locations.
  --source-root=<string>  -
                           Directory where processed files are stored.
                           Links to definition locations will only be
                           generated if the file is in this dir.
  --stylesheets=<string>  - CSS stylesheets to extend the default styles.
```

The following flags should only be used if **format** is set to **html**: - **repository** - **source-root** - **stylesheets**

The Doxygen documentation describes the **internal** software that makes up the tools of

clang-tools-extra, not the **external** use of these tools. The Doxygen documentation contains no instructions about how to use the tools, only the APIs that make up the software. For usage instructions, please see the user's guide or reference manual for each tool.

- *Doxygen documentation*

  NOTE:
  This documentation is generated directly from the source code with doxygen. Since the tools of clang-tools-extra are constantly under active development, what you're about to read is out of date!

- *Index*

- *Search Page*

AUTHOR

The Clang Team

COPYRIGHT