Secrets
ABAP
Apex
C
C++
CloudFormation
COBOL
C#
CSS
Flex
Go
HTML
Java
JavaScript
Kotlin
Kubernetes
Objective C
PHP
PL/I
PL/SQL
Python
RPG
Ruby
Scala
Swift
Terraform
Text
TypeScript
T-SQL
VB.NET
VB6
XML

C++ static code analysis

Unique rules to find Bugs, Vulnerabilities, Security Hotspots, and Code Smells in your C++ code

All rules 578 | Vulnerability 13 | Bug 111 | Security Hotspot 18 | Code Smell 436 | Quick Fix 68

Tags

Search by name...

"memset" should not be used to delete sensitive data
Vulnerability

POSIX functions should not be called with arguments that trigger buffer overflows
Vulnerability

XML parsers should not be vulnerable to XXE attacks
Vulnerability

Function-like macros should not be invoked without all of their arguments
Bug

The address of an automatic object should not be assigned to another object that may persist after the first object has ceased to exist
Bug

Assigning to an optional should directly target the optional
Bug

Result of the standard remove algorithms should not be ignored
Bug

"std::scoped_lock" should be created with constructor arguments
Bug

Objects should not be sliced
Bug

Immediately dangling references should not be created
Bug

"pthread_mutex_t" should be unlocked in the reverse order they were locked
Bug

"pthread_mutex_t" should be properly initialized and destroyed
Bug

"pthread_mutex_t" should not be consecutively locked or unlocked twice

"explicit" should be used on single-parameter constructors and conversion operators

Analyze your code

Code Smell | Critical | Quick Fix | cppcoreguidelines based-on-misra

If you invoked a method with arguments of the wrong type, you would typically expect an error at compile time (if not in the IDE). However, when the expected parameter is a class with a single-argument constructor, the compiler will implicitly pass the method argument to that constructor to implicitly create an object of the correct type for the method invocation. Alternately, if the wrong type has a conversion operator to the correct type, the operator will be called to create an object of the needed type.

But just because you *can* do something, that doesn't mean you *should*, and using implicit conversions makes the execution flow difficult to understand. Readers may not notice that a conversion occurs, and if they do notice, it will raise a lot of questions: Is the source type able to convert to the destination type? Is the destination type able to construct an instance from the source? Is it both? And if so, which method is called by the compiler?

Moreover, implicit promotions can lead to unexpected behavior, so they should be prevented by using the `explicit` keyword on single-argument constructors and (C++11) conversion operators. Doing so will prevent the compiler from performing implicit conversions.

Noncompliant Code Example

```
struct Bar {
};

struct Foo {
  Foo(Bar& bar); // Noncompliant; allow implicit conversion f
};

struct Baz {
  operator Foo(); // Noncompliant; allow implicit conversion
};

void func(const Foo& b); // this function needs a 'Foo' not a

int test(Bar& bar, Baz& baz) {
  func(bar); // implicit conversion using Foo::Foo(Bar& bar)
  func(baz); // implicit conversion using Baz::operator Foo()
  func(baz);
}
```

Compliant Solution

```
struct Bar {
};

struct Foo {
  explicit Foo(Bar& bar); // Compliant, using "explicit" keyw
};

struct Baz {
  Foo asFoo();              // Compliant, explicit function
  explicit operator Foo(); // Compliant, using C++11 "explici
};

void func(const Foo& b); // this function needs a 'Foo' not a
```

Bug

"std::move" and "std::forward" should not be confused

Bug

A call to "wait()" on a "std::condition_variable" should have a condition

Bug

A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast

Bug

Functions with "noreturn" attribute should not return

Bug

RAII objects should not be temporary

Bug

"memcmp" should only be called with pointers to trivially copyable types with no padding

Bug

"memcpy", "memmove", and "memset" should only be called with pointers to trivially copyable types

Bug

"std::auto_ptr" should not be used

Bug

Destructors should be "noexcept"

Bug

```
int test(Bar& bar, Baz& baz) {
  func(Foo(bar));              // explicit conversion using F
  func(baz.asFoo());           // explicit conversion using B
  func(static_cast<Foo>(baz)); // explicit conversion using B
}
```

Exceptions

C++20 introduced conditional `explicit(expr)` that allows developers to make a constructor or conversion operator conditionally explicit depending on the value of `expr`. The new syntax allows a constructor or conversion operator declared with an `explicit(expr)` specifier to be implicit when `expr` evaluates to `false`. The issue is not raised in such situation.

Additionally, developers can use `explicit(false)` to mark constructors or conversion operators as intentionally implicit.

See

- MISRA C++:2008, 12-1-3 - All constructors that are callable with a single argument of fundamental type shall be declared `explicit`.
- C++ Core Guidelines C.46 - By default, declare single-argument constructors explicit
- C++ Core Guidelines C.164 - Avoid implicit conversion operators

Available In:

sonarlint | sonarcloud | sonarqube Developer Edition