


-  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

Concise syntax should be used for concatenatable namespaces

Analyze your code

 Code Smell  Minor  Quick Fix  since-c++17  confusing

Namespaces represent a cross-file named scope. They are very useful to organize code and interfaces without cluttering a unique namespace. For instance, they provide a much cleaner way to avoid name collisions than using bad long names.

Namespaces can be nested to provide even more structure to type and symbol names. In that case, namespaces can be nested one inside another like scopes would with curly braces.

In C++17, a new concise syntax was introduced to increase the readability of nested namespaces. It is much less verbose and involves much less curly braces-delimited scopes. Whereas declaring a nested namespace of depth N requires N pairs of curly braces with the original syntax, this new syntax requires only one pair of curly braces. This syntax is much more readable and less error-prone. When possible, non-inlined or inlined (since C++20) named namespaces should be concatenated.

Noncompliant Code Example

```
namespace geometry { // Noncompliant
    namespace common {
        class point {
        };
    }
}

namespace geometry { // Noncompliant since C++20
    inline namespace triangle {
        class edge {
        };
    }
}
```

Compliant Solution

```
namespace geometry::common {
    class point {
    };
}

namespace geometry::inline triangle { // C++20
    class edge {
    };
}

namespace sonarsource { // Compliant: cannot be concatenated
    namespace core {
        class Rule {
        };
    }
    class A {
    };
}
```

Available In:

sonarlint

| sonarcloud

| sonarqube

Developer Edition

 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
"memcpy" 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