

- 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

"volatile" types should not be used in compound operations

Analyze your code

Bug Critical ?

The behavior of volatile types in compound expressions that both read and write the same variable (for instance `a++` or `a+=2`, where `a` is a `volatile int`) is not well defined, and is deprecated since C++20.

Such expressions should be replaced by explicit operations that only read or write the value.

## Noncompliant Code Example

```
void f1(int volatile* p) {
    ++(*p); // Noncompliant
}

void f2(volatile int& in) {
    in += 2; // Noncompliant
}

void f3(volatile int& in) {
    int i = in = 2; // Noncompliant
}
```

## Compliant Solution

```
void f1(int volatile* p) {
    auto val = *p; // One access to read the register
    *p = val + 1; // One access to write to it (and potentially
}

void f2(volatile int& in) {
    auto val = in;
    in = val + 2;
}

void f3(volatile int& in) {
    in = 2;
    int i = in;
}
```

Available In:

sonarlint | sonarcloud | sonarqube Developer Edition

 Bug
<b>"std::move" and "std::forward" should not be confused</b>  Bug
<b>A call to "wait()" on a "std::condition_variable" should have a condition</b>  Bug
<b>A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast</b>  Bug
<b>Functions with "noreturn" attribute should not return</b>  Bug
<b>RAII objects should not be temporary</b>  Bug
<b>"memcmp" should only be called with pointers to trivially copyable types with no padding</b>  Bug
<b>"memcpy", "memmove", and "memset" should only be called with pointers to trivially copyable types</b>  Bug
<b>"std::auto_ptr" should not be used</b>  Bug
<b>Destructors should be "noexcept"</b>  Bug