



SAP ABAP

<sub>АРЕХ</sub> Арех

C C



CloudFormation

COBOL COBOL

C# C#

css

X Flex

**GO** Go

HTML

Java

Js JavaScript

Kotlin

Kubernetes

Objective C

PHP

PL/I

PL/SQL

Python

RPG RPG

Ruby

Scala

Swift

Terraform

**Text** 

TS TypeScript

T-SQL

VB VB.NET

VB6 VB6

XML XML



## C++ static code analysis

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

All 578 rules Vulnerability 13

**R** Bug (111)

Security Hotspot

Code 436

O Quick 68 Fix

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

<table-of-contents> 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 "[\*this]" should be used to capture the current object by copy

Analyze your code

Code Smell

Quick Fix since-c++17 clumsy pitfall

When you are using lambdas in a member function, you can capture this implicitly through [=] or [&] or explicitly through [this]. This will capture the current object pointer by reference or by value, but the underlying object will always be captured by reference (see {rule:cpp:S5019}).

This will become a problem:

- When the lifetime of the lambda exceeds the one of the current object.
- When you want to capture the current state of the object.
- When you want to pass a copy of the object to avoid any concurrency issue.

C++14 provides a workaround to solve this problem. Where you can take the underlying object by copy using the following pattern:

```
auto lam = [copyOfThis = *this] { std::cout << copyOfThis.fie
```

This is verbose and error-prone, as you might implicitly not use the copied object:

```
auto lam = [& , copyOfThis = *this] {
std::cout << field; // implicitly calling "this" captured by
};</pre>
```

C++17 solves this problem by introducing an explicit consistent way to capture this by copy:

```
auto lam = [&, *this] {
std::cout << field // implicitly calling "this" captured by c
};</pre>
```

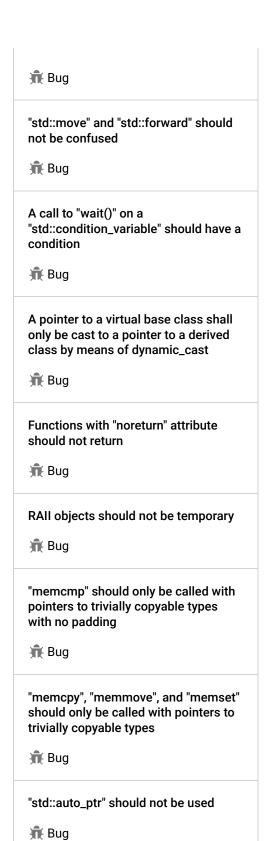
This rule will flag the C++14 way of capturing the current object by copy and suggest to replace it with the C++17 way.

## Noncompliant Code Example

```
struct A {
  int field = 0;
  void memfn() const {
    auto lam = [copyOfThis = *this] { // Noncompliant
       std::cout << copyOfThis.field;
    };
};
</pre>
```

## **Compliant Solution**

```
struct A {
  int field = 0;
  void memfn() const {
    auto lam = [*this] { // Compliant
       std::cout << field;
    };
};
}</pre>
```



Destructors should be "noexcept"

📆 Bug

Available In:

sonarlint sonarcloud sonarqube Developer Edition

© 2008-2022 SonarSource S.A., Switzerland. All content is copyright protected. SONAR, SONARSOURCE, SONARLINT, SONARQUBE and SONARCLOUD are trademarks of SonarSource S.A. All other trademarks and copyrights are the property of their respective owners. All rights are expressly reserved.

Privacy Policy