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

Template parameters should be preferred to "std::function" when configuring behavior at compile time

Analyze your code

Code Smell | Critical | cppcoreguidelines performance bad-practice

To configure an algorithm with a function in C++, you can use one of the following techniques:

- A function pointer (see {rule:cpp:S5205} that explains why it is a bad idea)
- An `std::function`
- A template argument

How do you select between an `std::function` and a template argument?

`std::function` offers the most flexibility. You can store them in a variable, in a container (as `std::map<string, std::function<void(void)>>` for instance... This flexibility is provided by type erasure: A single `std::function` can wrap any kind of functor, as long as the signature is compatible. It also comes with a cost: Due to this type erasure, a compiler will typically not be able to inline a call to a `std::function`.

Template parameters, on the other hand, are less flexible. Each functor has its own type, which prevents storing several of them together even if they all have compatible signatures. But since each template instance knows the type of the functor, calls can be inlined making this a zero-cost abstraction.

As a conclusion, if the functor can be known at compile-time, you should prefer using a template parameter, if it has to be dynamic, `std::function` will give you greater flexibility.

This rule detects function parameters of type `std::function` that would probably benefit from being replaced by a template parameter. It does so by looking if the functor is only called inside the function, or if it participates in other operations.

Noncompliant Code Example

```
using Criterion = std::function<bool(DataPoint const&)>;
void filter(DataSet* data, Criterion criterion) { // Noncompl
  for (auto &dataPoint : data) {
    if (criterion(dataPoint)) {
      data.markForRemoval(dataPoint);
    }
  }
}
```

Compliant Solution

```
template<class Criterion>
void filter(DataSet* data, Criterion criterion) { // Complian
  for (auto &dataPoint : data) {
    if (criterion(dataPoint)) {
      data.markForRemoval(dataPoint);
    }
  }
}
```

Exceptions

This rule ignores virtual functions, that don't work well with templates.

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

See

- C++ Core Guidelines T.49 - Where possible, avoid type-erasure

Available In:

sonarlint | sonarcloud | sonarqube Developer Edition