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

Member data should be initialized in-class or in a constructor initialization list

Analyze your code

Code Smell | Major | cppcoreguidelines performance

There are three ways to initialize a non-static data member in a class:

- With an in-class initializer (since C++11)
- In the initialization list of a constructor
- In the constructor body

You should use those methods in that order of preference. When applicable, in-class initializers are best, because they apply automatically to all constructors of the class (except for default copy/move constructors and constructors where an explicit initialization for this member is provided). But they can only be used for initialization with constant values.

If your member value depends on a parameter, you can initialize it in the constructor's initialization list. If the initialization is complex, you can define a function to compute the value, and use this function in the initializer list.

Initialization in the constructor body has several issues. First, it's not an initialization, but an assignment. Which means it will not work with all data types (const-qualified members, members of reference type, member of a type without default constructor...). And even if it works, the member will first be initialized, then assigned to, which means useless operations will take place. To prevent "use-before-set" errors, it's better to immediately initialize the member with its real value.

It's hard to find a good example where setting the value of a member in the constructor would be appropriate. One case might be when you assign to several data members in one operation. As a consequence constructor bodies are empty in many situations.

This rules raises an issue in two conditions:

- When you assign a value to a member variable in the body of a constructor.
- When you default-initialize in an initializer list a member variable, that would be value-initialized by default
- For C++11 or later, when you initialize a member variable in the initializer list of a constructor, but could have done so directly in the class:
  - The variable has either no in-class initializer, or an in-class initializer with the same value as in the constructor
  - The initial value does not depend on a constructor parameter

Noncompliant Code Example

```
class S {
  int i1;
  int i2;
  int i3;
public:
  S( int halfValue, int i2 = 0) : i2(i2), i3(42) { // Noncomp
    this->i1 = 2*halfValue;
  }
};
```

Compliant Solution

```
class S {
  int i1;
  int i2;
  int i3 = 42; // In-class initializer
public:
  S( int halfValue, int i2 = 0 ) : i1(2*halfValue), i2(i2) {}
```

```
};
```

See

- C++ Core Guidelines C.48 - Prefer in-class initializers to member initializers in constructors for constant initializers
- C++ Core Guidelines C.49 - Prefer initialization to assignment in constructors

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

"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