



Apex

ABAP

С

C++

CloudFormation

COBOL

C#

CSS

Go

Flex

5 HTML

Java

JavaScript

Kotlin

Kubernetes

Objective C

PHP

PL/I

PL/SOL

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

ΑII 578 6 Vulnerability (13) rules

R Bug (111)

• Security Hotspot

(18)

⇔ 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

📆 Bug

Objects should not be sliced

🖷 Bug

Immediately dangling references should not be created

📆 Bug

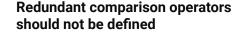
"pthread_mutex_t" should be unlocked in the reverse order they were locked

📆 Bug

"pthread_mutex_t" should be properly initialized and destroyed

📆 Bug

"pthread_mutex_t" should not be consecutively locked or unlocked



Analyze your code

since-c++20 clumsy pitfall

C++20 introduces rewriting rules that enable defining only a few operator overloads in a class to be able to compare class instances in many ways:

- the "spaceship" operator<=> can replace all the other comparison operators in most cases: The code a @ b (where @ is one of the following operators: <, <=, >, or >=) can be implicitly rewritten to use either a<=>b or b<=>a, and its three-way comparison semantics instead.
- If operator == is defined, a!=b can be implicitly rewritten! (a==b)
- If an operator<=> is defined as =default, a matching operator== is automatically generated if it does not already exist.

If you define your own version of any particular comparison operator, e.g., operator< in addition to the operator<=>, it will supersede the compilergenerated version and might result in a surprising behavior with operator< semantics inconsistent with the semantics of other operators defined through operator<=>.

In most cases, you will only have to define the following set of comparison operators in your class (possibly several of those sets, to allow for mixed-type comparison):

- No comparison operator, if the class should not be compared, or
- only operator == for classes that can only be compared for equality (and
- only operator<=>, defined as =default for fully comparable classes that only need to perform comparison member by member, or
- both operator<=> and operator== when the comparison is more complex.

This rule will raise an issue when a class is defined:

- With an operator<=> and any of the four operators <, <=, >, >= defined with the
- With both operator == and operator! = defined for the same types.
- With a defaulted operator<=> and a defaulted operator== with the same argument types defined.
- With two operator<=> or two operator== that are declared with the same argument types in reverse order.

Noncompliant Code Example

Example with redundant operations in the same class:

```
class A {
  int field;
  public:
    auto operator<=>(const A&) const = default;
    bool operator<(const A& other) const { // Noncompliant: t
      return field < other.field;</pre>
    bool operator == (const A&) const = default; // Noncomplian
};
```

Example with equivalent operations in different order:

```
class MyStr {
  friend std::strong_ordering operator<=>(MyStr const &s1, st
  friend std::strong_ordering operator<=>(std::string const &
};
```



"std::move" and "std::forward" should not be confused



A call to "wait()" on a "std::condition_variable" should have a condition



A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast



Functions with "noreturn" attribute should not return



RAII objects should not be temporary



"memcmp" should only be called with pointers to trivially copyable types with no padding



"memcpy", "memmove", and "memset" should only be called with pointers to trivially copyable types



"std::auto_ptr" should not be used

📆 Bug

Destructors should be "noexcept"

```
📆 Bug
```

Compliant Solution

The class has been reduced to a minimal set:

```
class A {
 int field;
 public:
   auto operator<=>(const A&) const = default; // Compliant:
};
// The following code is valid:
void f(A const &a1, A const &a2) {
 bool b1 = a1 == a2; // Uses implicitly generated operator==
 bool b2 = a1 != a2; // Uses implicitly generated operator==
 bool b3 = a1 < a2; // Rewritten as: (a1 <=> a2) < 0
 bool b4 = a1 >= a2; // Uses implicitly generated operator==
 bool b1 = a1 == a2; // Uses implicitly generated operator==
```

Only one order needs to be written

```
class MyStr {
  friend std::strong_ordering operator<=>(MyStr const &s1, st
// The following code is valid
void f(MyStr const &s1, std::string const &s2) {
  bool b1 = s1 < s2; // Rewritten as: (s1<=>s2) < 0
  bool b2 = s2 \Rightarrow s1; // Rewritten as 0 \Rightarrow (s1<=>s2);
```

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**