



ABAP

Apex

C

C++

CloudFormation

COBOL

C#

CSS

Flex

Go **GO**

5 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

578 ΑII **6** Vulnerability (13) rules

R Bug (111)

• Security Hotspot ⊗ Code (436)

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

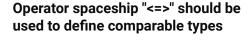
"pthread_mutex_t" should be unlocked in the reverse order they were locked

📆 Bug

"pthread_mutex_t" should be properly initialized and destroyed

📆 Bug

"pthread_mutex_t" should not be consecutively locked or unlocked



Analyze your code

☼ Code Smell ♥ Minor ②

since-c++20 clumsy

C++20 introduces the "spaceship" operator<=> that replaces all the other comparison operators in most cases. When this operator is defined, the compiler can rewrite expressions using <, <=, > and >= to use this operator instead. This presents three advantages:

- Less code to write (and therefore fewer bugs too),
- Guaranteed consistency between all the comparison operators (for instance, in this situation, a < b and !(a >= b) will always return the same value).
- Guaranteed symmetry for comparisons: If you can write a<b, and that operation is resolved through operator<=>, you can also write b<a, and get a consistent result. Achieving the same result with classical comparison operators require to double the number overloads if a and b are of different types.

Additionally, if the operator<=> has the defaulted implementation, the compiler can also implicitly generate a defaulted implementation of operator==, simplifying the class definition one step further.

Before C++20, if was common to provide only operator< for a class, and ask the user of this class to write all his code only using this operator (this is what std::map requires of its key type, for instance). It is still advised in this case to replace the operator with <=>: The quantity of required work is similar, and the user of the class will benefit from a much greater expressivity.

This rule reports user-provided comparison operators (member functions or free functions) <, <=, > and >=.

Noncompliant Code Example

```
class A { // Noncompliant: defines operator< that can be repl</pre>
  int field;
  public:
    bool operator<(const A& other) const {</pre>
      return field < other.field;</pre>
};
class C;
class B { // Noncompliant: defines 12 comparison operators th
  public:
    bool operator==(const C&) const;
    bool operator!=(const C&) const;
    bool operator<=(const C&) const;</pre>
    bool operator<(const C&) const;</pre>
    bool operator>=(const C&) const;
    bool operator>(const C&) const;
    friend bool operator==(const C&, const B&);
    friend bool operator!=(const C&, const B&);
    friend bool operator <= (const C&, const B&);
    friend bool operator<(const C&, const B&);</pre>
    friend bool operator>=(const C&, const B&);
    friend bool operator>(const C&, const B&);
};
enum class MyEnum {low = 1, high = 2};
bool operator<(MyEnum lhs, MyEnum rhs) { // Noncompliant: can
    return static_cast<int>(lhs) < static_cast<int>(rhs);
}
```

Compliant Solution



"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

Rug Bug

"std::auto_ptr" should not be used

🕀 Bug

Destructors should be "noexcept"

```
📆 Bug
```

```
class A {
  int field;
  public:
    auto operator<=>(const A& other) const = default;
    // Note that here, operator == will be implicitly default
};
class B \{ // Compliant. the same comparisons are possible as
  int field;
  public:
    auto operator<=>(const C&) const;
    auto operator==(const C&) const;
};
enum class MyEnum {low = 1, high = 2};
auto operator<=>(MyEnum lhs, MyEnum rhs) {
    return static cast<int>(lhs) <=> static cast<int>(rhs);
```

See

• {rule:cpp:S6186} - Redundant comparison operators should not be defined.

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