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

STL algorithms and range-based for loops should be preferred to traditional for loops

Analyze your code

Code Smell | Minor | cppcoreguidelines | clumsy

`for`-loops are a very powerful and versatile tool that can be used for many purposes. This flexibility comes with drawbacks:

- It is very easy to make a small mistake when writing them,
- They are relatively verbose to write,
- They do not express the intent of the code, the reader has to look at loop details to understand what the loop does.

There are algorithms that encapsulate a `for`-loop and give it some meaning (`std::all_of`, `std::count_if`, `std::remove_if`…). These algorithms are well tested, efficient, and explicit and therefore should be your first choice.

This rule detects loops that go through all consecutive elements of a sequence (eg: containers, objects with begin() and end() member functions), and deal only with the current element without side-effects on the rest of the sequence.

This rule suggests using one of the supported STL algorithm patterns corresponding to your C++ standard when a loop matches it.

Currently, this rule supports:

- `std::all_of` (since C++11) and `std::ranges::all_of` (since C++20): returns `true` if all elements in the given range are matching the given predicate, `false` otherwise
- `std::none_of` (since C++11) and `std::ranges::none_of` (since C++20): returns `true` if no elements in the given range are matching the given predicate, `false` otherwise
- `std::any_of` (since C++11) and `std::ranges::any_of` (since C++20): returns `true` if at least one element in the given range is matching the given predicate, `false` otherwise

This rule suggests two options below when the loop doesn't match any of the supported STL algorithm patterns and you just want to iterate over all elements of a sequence:

- Range-based `for`-loops, which were introduced in C++11 and will run through all elements of a sequence
- `std::for_each`, an algorithm that performs the same operation between two iterators (allowing more flexibility, for instance by using `reverse_iterators`, or with a variant that can loop in parallel on several elements at a time).

Noncompliant Code Example

```
#include <vector>
#include <iostream>

using namespace std;

bool asDesired(const int v);

bool areAllDesired(std::vector<int> values) {
  for (int val : values) { // Noncompliant, replace it by a c
    if (!asDesired(val)) {
      return false;
    }
  }
  return true;
}

int f(vector<int> &v) {
```

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

```
  for (auto it = v.begin(); it != v.end(); ++it) { // Noncomp
    if (*it > 0) {
      cout << "Positive number : " << *it << endl;
    } else {
      cout << "Negative number : " << *it << endl;
    }
  }

  auto sum = 0;
  for (auto it = v.begin(); it != v.end(); ++it) { // Noncomp
    sum += *it;
  }
  return sum;
}
```

Compliant Solution

```
#include <vector>
#include <iostream>
#include <algorithm>

using namespace std;

bool asDesired(const int v);

bool areAllDesired2(std::vector<int> values) {
  return std::all_of(std::begin(values), std::end(values), as
}

bool areAllDesiredCpp20(std::vector<int> values) {
  return std::ranges::all_of(values, asDesired);
}

void displayNumber(int i) {
  if (i > 0) {
    cout << "Positive number : " << i << endl;
  } else {
    cout << "Negative number : " << i << endl;
  }
}

void f(vector<int> &v) {

  std::for_each(v.begin(), v.end(), displayNumber);
  // Or since C++20:
  std::ranges::for_each(v, displayNumber);

  auto sum = 0;
  for (auto elt : v) {
    sum += elt;
  }
  return sum;
  // An even better way to write this would be:
  // return std::accumulate(v.begin(), v.end(), 0);
}
```

See

- C++ Core Guidelines ES.71 - Prefer a range-for-statement to a for-statement when there is a choice
- C++ Core Guidelines P.3 - Express intent

Available In:

sonarlint | sonarcloud | sonarqube Developer Edition