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

Redundant lambda return types should be omitted

Analyze your code

Code Smell | Minor | bad-practice | since-c++11

It is a best practice to make lambda return types implicit whenever possible. First and foremost, doing so avoids implicit conversions, which could result in data or precision loss. Second, omitting the return type often helps future-proof the code.

The issue is raised when explicit return types are used.

Noncompliant Code Example

```
// Noncompliant: the explicit return types are redundant.
[](int i) -> int { return i + 42; }
[]() -> auto { return foo(); }
[](int x) -> std::vector<int> { return std::vector<int>{ x };
```

Compliant Solution

```
// Compliant: no explicit return type.
[](int i) { return i + 42; }
[]() { return foo(); }
[](int x) { return std::vector<int>{ x }; }
```

Exceptions

There are a few exceptions to this rule.

First, no issue is raised when the compiler is not deducing the same type by itself. This can happens when a conversion is requested.

```
// Compliant: the compiler would deduce a different return ty
[](int x) -> double { return x; }
[](float x) -> int { return x; } // Precision loss, see S5276
```

The compiler also deduces a different type when there are no `return` statements and the explicit return type is not `void`.

```
// Compliant: removing these explicit return types would resu
[](int x) -> int { throw std::runtime_error("No more eggs");
[](int x) -> int { std::terminate(); }
```

Another similar situation is when references are involved: instead, the compiler deduces a value without an explicit return type. This can have an impact on both correctness and performance.

```
// Compliant: removing the explicit return type would return
[](std::vector<int>& data) -> auto& {
    std::sort(data.begin(), data.end());
    return data;
}
```

Additionally, no issues are raised when the deduction of the return type is not available. This is the case with C++20 coroutines in their lambda form.

```
// Compliant: coroutine lambdas cannot rely on type deduction
[]() -> Task { co_await std::suspend_always{}; }
```

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

In some other cases, removing the explicit return type would result in ill-formed programs. This is the case when using initializer lists or aggregate initializations.

```
// Compliant: type deduction wouldn't work.
[](int x) -> std::vector<int> { return { x }; }
[]() -> std::array<int, 4> { return { 1, 2, 3, 4 }; }
```

Removing the explicit return type when a lambda has multiple `return` statements of different types would also result in ill-formed programs. Here are two examples where the explicit return type introduces useful implicit conversions.

```
// Compliant: omitting the return type would result in a comp
[](Base* ptr) -> Derived* {
    if (auto* derived = dynamic_cast<Derived*>(ptr)) {
        actOnDerived(derived);
        return derived;
    }

    // "nullptr_t" mismatches the previous return type "Deriv
    return nullptr;
}

// Compliant: omitting the return type would result in a comp
[](std::string_view request) -> std::variant<Error, Data> {
    if (!isRequestValid(request)) {
        return Error{ "invalid request" };
    }

    auto reader = readRequest(request);
    // "Data" mismatches the previous return type "Error".
    return Data{ reader.data(), reader.size() };
}
```

Finally, this rule does not trigger on explicit template-dependent or constrained return types since they can have a use of their own, help readability or improve maintainability.

```
// Compliant: enforce "process()" returns a reference.
[](auto x) -> auto& { return process(x); }

// Compliant: ensure the lambda returns a reference when "f()
[f](auto arg) -> decltype(f(arg)) { return f(arg); }

// Compliant: the return type restricts possible input types.
[](auto x) -> std::enable_if_t<std::is_integral_v<decltype(x)

template <typename T>
T getGlobalProperty(std::string_view name, T defaultValue) {

// Compliant: the return type is constrained (C++20) and can
auto getProperty = [](std::string_view name) -> std::totally_
    return getGlobalProperty(name, 0);
}
```

See

- {rule:cpp:S5276}: Implicit casts should not lower precision

Available In:

sonarlint | sonarcloud | sonarqube Developer Edition