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

## "std::enable_if" should not be used

Analyze your code

Code Smell | Major | since-c++20 clumsy

`std::enable_if` is a very important part of template meta-programming in C++ up to C++17. Based on SFINAE, it can be used to subtly tune the behavior of overload resolution based on properties of types.

However, using `std::enable_if` correctly is not easy, and requires skills and experience, for a resulting code that is not straightforward. Since C++20, new features offer first-class support for what used to require `enable_if` trickery:

- Concepts allow defining named constraints on types, using a terse syntax to specify that a template argument must adhere to a concept;
- `requires` clauses can be directly written for one-shot constraints;
- In some cases, using `if constexpr` (introduced in C++17) may replace an overload set with just one function (see {rule:cpp:S6017}).

Additionally, since those features provide a higher level of abstraction, compilers understand them better and can provide clearer diagnostics when a constraint is violated.

As a consequence, `std::enable_if` is no longer the right tool and should be replaced with those facilities. Note that the replacement is not always mechanical: The expression controlling a `std::enable_if` would probably be acceptable as a `requires` condition, but better alternatives usually exist, for instance reusing an existing concept defined in the standard.

This rule reports the use of `std::enable_if`.

### Noncompliant Code Example

```
template <typename N, class = typename
  std::enable_if<std::is_integral_v<N> && std::is_signed_v<N>
auto negate(N n) { return -n; }
```

### Compliant Solution

```
template <class N> requires std::signed_integral<N>
auto negate(N n) { return -n; }
```

Or

```
template <std::signed_integral N>
auto negate(N n) { return -n; }
```

Or

```
auto negate(std::signed_integral auto n) { return -n; }
```

### See

Why I want Concepts, and why I want them sooner rather than later

### See Also

- {rule:cpp:S6017} to see when `std::enable_if` could be replaced with `if constexpr`.

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