

"pthread_mutex_t" should be properly

"pthread_mutex_t" should not be consecutively locked or unlocked

initialized and destroyed

📆 Bug





"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

📆 Bug

"std::auto_ptr" should not be used

📆 Bug

Destructors should be "noexcept"

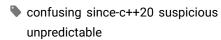
📆 Bug

Thread local variables should not be used in coroutines

Analyze your code

Code Smell





In contrast to normal functions, coroutines can suspend and later resume their execution. Depending on the program, the coroutine may resume on a different thread of execution than the one it was started or run previously on.

Therefore, the access to the "same" variable with thread_local storage may produce different values as illustrated below:

```
thread local std::vector<Decorator> decorators;
lazy<Thingy> doSomething() {
  // evaluation started on thread t1
  /* .... */
  const std::size_t decoratorCount = decorators.size(); // va
  auto result = co_await produceThingy();
  // after co_await, execution resumes on thread t2
  for (std::size_t i = 0; i < decoratorCount; ++i) {</pre>
    decorators[i].modify(result); // access value specific to
    // miss some tasks if t1:decorators.size() < t2:decorator</pre>
    // undefined behavior if t1:decorators.size() > t2:decora
 }
  co_return result;
```

This behavior is surprising and unintuitive compared to normal functions that are always evaluated on a single thread. The same issue can happen for the use of different thread-local variables if their values are interconnected (e.g., one is the address of the buffer, and the other is the number of elements in the buffer).

Moreover, access to thread-local variables defined inside the coroutine may read uninitialized memory. Each such variable is initialized when a specific thread enters the function for the first time, and if the function was never called from a thread on which the coroutine is resumed, it is uninitialized.

This rule raises an issue on the declaration of thread_local variables and access to thread_local variables in coroutines.

Noncompliant Code Example

```
thread_local std::vector<Decorator> decorators;
lazy<Thingy> doSomething() {
  thread local Decorator localDecorator; // Noncompliant
  const std::size_t decoratorCount = decorators.size(); // No
  auto result = co_await produceThingy();
  for (std::size_t i = 0; i < taskCount; ++i) {</pre>
    decorators[i].modify(result);
  localDecorator.modify(result); // Noncompliant
  co_return result;
```

Available In:

sonarlint in 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