C++ static code analysis

Unique rules to find Bugs, Vulnerabilities, Security Hotspots, and Code Smells in your C++ code

Use conditional suspension to resume current coroutine

Analyze your code

Code Smell Minor performance since-c++20

One of the use cases for the coroutines is suspending execution until certain conditions are satisfied (e.g. value is produced, flag/event is triggered). In some situations, the expected result may be already available at the point of the `co_await/co_yield` expression, and the execution can be resumed immediately.

The C++ standard provides an efficient method to suspend the coroutine conditionally. The result of `await_ready` is used to determine whether a coroutine should be suspended. Returning `true` from this function avoids the cost of the coroutine suspension if it is not needed (e.g., the result is already available). Furthermore, the `bool`-returning version of `await_suspend` allows immediate resumption of the current coroutine in the case when `false` is returned (returning `true` indicates that the coroutine should remain suspended). Compared to symmetric transfer, this method provides better optimization opportunities, as the continuation code is known to the compiler - i.e., it is the code of the current coroutine, while in symmetric transfer the handle could point to an arbitrary coroutine.

This rule raises an issue on `await_suspend` that can benefit from using conditional suspension.

Noncompliant Code Example

```
struct WaitForAwaiter {
  Event& event;
  /* .... */
  std::coroutine_handle<> await_suspend(std::coroutine_handle
    bool callback_registered = event.register_callback(curren
    if (!callback_registered) {
      return current;
    } else {
      return std::noop_coroutine();
    }
  }
};

struct ReadBytesAwaiter {
  Socket& socket;
  std::size_t count;
  std::span<std::byte> buffer;
  std::error_code error;
  /* .... */
  void await_suspend(std::coroutine_handle<> current) { // No
    auto callback = [&error_store=error, current](std::error_
      error_store = ec;
      current.resume();
    };

    auto ec = socket.async_read(buffer, count, callback);
    if (ec) {
      error = ec;
      current.resume();
    }
  }
};
```

Compliant Solution

```
struct WaitForAwaiter {
```

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

```
  Event& event;
  /* .... */
  bool await_ready() const {
    return event.is_already_triggered();
  }
  bool await_suspend(std::coroutine_handle<> current) {
    bool callback_registered = event.register_callback(curren
    return callback_registered;
  }
};

struct ReadBytesAwaiter {
  Socket& socket;
  std::size_t count;
  std::span<std::byte> buffer;
  std::error_code error;
  /* .... */

  bool await_ready() const {
    return false; // no way to query before suspension
  }
  bool await_suspend(std::coroutine_handle<> current) {
    auto callback = [&error_store=error, current](std::error_
      error_store = ec;
      current.resume();
    };

    auto ec = socket.async_read(buffer, count, callback);
    if (ec) {
      error = ec;
      return false;
    }

    return true;
  }
};
```

See

{rule:cpp:S6365} - transferring execution to any suspended coroutine

Available In:

sonarlint sonarcloud sonarqube Developer Edition