The Most Vexing Capture: A Treacherous C++ Foot-gun

David Mazières

December, 2025

Introduction

Since deciphering C++ coroutines a few years ago, I've started using them regularly. While I still wish the design were cleaner, I've mostly made peace with it, and at various points have taken advantage of almost every feature they provide. However, there is one particularly pernicious category of bug that I keep introducing into my coroutine code. Somehow, over and over, I've been able to look at 2–3 lines of code, ask myself, "Does this have the bug?" and then convince myself incorrectly that it's fine, only later to discover a use-after-free error with address sanitizer.

I call this problem the *most vexing capture*, or MVC, bug. My hope is that giving it a name and writing about it will make it easier for me to recognize and save other people from similar frustration. Without further ado, let me show you an example of the bug. It requires some coroutine infrastructure that we'll have to build up, but here is the actual code with the bug:

```
// Any function that returns type Detached is a "detached" coroutine
// that self-destroys when done.
struct Detached;

// A queue that lets coroutines suspend themselves and request to
// be resumed in a given amount of time.
struct CoroSleep;

CoroSleep cs;

// Not actually a coroutine, but returns something you can co_await
auto
thing()
{
  std::println("doing a thing");
  return cs.sleep(1s);
}

void
```

```
do_n_things(int n)
{
  [n] -> Detached {   // Serious BUG!
    for (int i = 0; i < n; ++i)
      co_await thing();
  }(); // <-- lambda destroyed here
}
```

At first glance, you may think `n` is captured by value, so this is safe. Unfortunately, `n` is captured into a lambda object, which is a temporary object destroyed at the end of the full expression (at the semicolon right after the function call). Yet the coroutine is suspended by `thing()`, which means it will be resumed later, after the lambda is destroyed, making the access to `n` in the loop condition a use-after-free bug.

For the next part of this blog post, I'll review how C++ coroutines work. Then I'll give an example of how to use coroutines by describing the implementation of `Detached` and `CoroSleep`. People familiar with coroutines can skip over these sections, but for those still learning coroutines, they provide a good reference of the language facilities and an example of how to use them. Finally, I'll explain why the bug occurs and discuss how to work around it. In particular, the surprisingly versatile C++23 deducing this feature provides a clean workaround when you are conscious of the problem and want to avoid it.

You can download the code in this blog post here. Or if you want to cut and paste from the text, you will want these includes:

```
#include <chrono>
#include <condition_variable>
#include <coroutine>
#include <latch>
#include <map>
#include <mutex>
#include <print>
#include <thread>
#include <utility>
```

Coroutine review

If you've never seen C++ coroutines before, I recommend reading my earlier tutorial. This section is more of a C++ coroutine reference. If you are already very familiar with coroutines, you may want to skip straight to the section on MVC bugs.

A *coroutine* is a call frame that does not reside on a stack (other than possibly as an optimization in certain uninteresting corner cases). As a result, the coroutine can be repeatedly resumed and suspended at arbitrary points from different functions and in different threads.

A coroutine is created by invoking a *coroutine function*, which is an ordinary-looking function

whose definition uses at least one of the `co_await`, `co_yield`, or `co_return` keywords. If you disassemble a coroutine function, you generally see a call to `operator new` at the beginning of the function to allocate a call frame and associated state in the heap. That said, you cannot simply use coroutine keywords in arbitrary functions to turn them into coroutines. These keywords are only valid in functions whose signature can be used to determine a *promise type* meeting certain criteria.

If you have a coroutine function with signature `R function(A0, ..., An)` or a coroutine method on a class `A0` with signature `R A0::method(A1, ..., An)`, the coroutine's promise type is `typename std::coroutine_traits<R, A0, ..., An>::type`. If no such type exists, or if the resulting type does not have certain required methods, then the function or method cannot be a coroutine. The default value for `coroutine_traits<R, Args...>::type` is `typename R::promise_type`, meaning the promise type depends only on the return value. When relying on the default, this means a function can use coroutine keywords when its return type contains a nested structure or type alias `promise_type`.

Coroutine state includes an instance of the promise type, known as the *promise object*, and a call frame with the coroutine's local variables. A pointer to this state has type `std::coroutine_handle<promise_type>`, which can also be implicitly converted to the type-erased version `std::coroutine_handle<>` (the default template argument is `void`). A coroutine handle is a dumb pointer that can be trivially copied and destroyed, not a smart pointer like `std::shared_ptr` that does something when it goes out of scope. If `h` is a coroutine handle and the coroutine is suspended, then `h.resume()` (or, equivalently, `h()`) resumes the coroutine in the current thread. `h.destroy()` deallocates the coroutine's state. (Some coroutines deallocate themselves on completion, depending on the promise object; it is an error to call `h.destroy()` on such a self-deallocated coroutine.)

A coroutine gets suspended by evaluating the expression `co_await a;` for an *awaitable* `a` that suspends the coroutine. An awaitable can dynamically decide whether or not to suspend a coroutine, for instance depending on whether some IO or asynchronous result is available. When an awaitable does suspend the coroutine, it obtains a copy of the coroutine's handle so as to be able to resume the coroutine later on.

Promise object reference

The following methods have special meaning on promise objects. Those not tagged optional must be present in valid promise types.

- [optional] **`(constructor)()`** or **`(constructor)(A0, ..., An)`** – The promise object can be constructed by a default constructor (whether explicitly defined or not). Alternatively, it can be initialized from the exact same arguments that were passed to the coroutine function. In the latter case, the arguments to the promise object constructor are initialized from the coroutine function arguments after they have already been copied into the coroutine's call frame. Hence, if the coroutine function takes an argument by value and the promise object constructor takes the same argument by reference, the reference will be to the object in the coroutine frame, not in the invoking function's call frame.

Note that with some care it's possible to add extra arguments with default values. In particular, it can be useful for the constructor to take a final argument `std::source_location loc = std::source_location::current()` that is not an argument of the coroutine function. This makes it possible to trace where coroutines are created.

- **`R get_return_object()`** – A coroutine function is still a function with a return value. The function returns this value when the coroutine is first suspended—often before the coroutine completes. In fact, the `return` keyword is illegal inside coroutines, so a coroutine function's return value must be created independent of the coroutine itself. The promise object therefore bears responsibility for the coroutine function's return value, which it produces in the `get_return_object` method.
- [optional] **`R get_return_object_on_allocation_failure()`** – Barring optimized corner cases, a coroutine's state requires dynamically allocated memory. C++ obtains this memory either from the promise object's `operator new` or, more typically, from the global `::operator new` function. If the promise object has a `get_return_object_on_allocation_failure` method, then C++ uses the `std::nothrow_t` version of `operator new`. If this allocator returns `nullptr`, then no coroutine can be created, and C++ invokes `get_return_object_on_allocation_failure` to produce an error version of the return value.
- **`Awaitable initial_suspend()`** – all coroutine functions begin execution by constructing the promise object (which we'll name by a fictitious reference `promise`) and evaluating `co_await promise.initial_suspend();` before the first real line of code in the coroutine function is actually evaluated. Typically the return type `Awaitable` would be either `std::suspend_always` (to begin execution in a suspended state) or `std::suspend_never` (to begin execution of the coroutine function body before returning from the coroutine function). For example:

```
namespace detail {
struct SuspendedCoroPromise {
  // The return value is the coroutine handle itself
  std::coroutine_handle<SuspendedCoroPromise> get_return_object()
  {
    return std::coroutine_handle<SuspendedCoroPromise>::from_promise(*this);
  }

  // Always suspend the coroutine before executing it
  static std::suspend_always initial_suspend() noexcept { return {}; }

  void return_void() const noexcept {}
  void unhandled_exception() noexcept { std::terminate(); }

  // Don't suspend after co_return, meaning the coroutine will
  // deallocate itself and doesn't need to be destroyed explicitly.
```

```
  static std::suspend_never final_suspend() noexcept { return {}; }
};
} // namespace detail

using Suspended = std::coroutine_handle<detail::SuspendedCoroPromise>;

// Register SuspendedCoroPromise as a promise type for functions
// of arbitrary arguments returning Suspended.
template<typename... Args>
struct std::coroutine_traits<Suspended, Args...> {
  using promise_type = detail::SuspendedCoroPromise;
};

int
main()
{
  std::coroutine_handle<> h = [] -> Suspended {
    std::println("The coroutine has started");
    co_return;
  }();
  // coroutine has been created and immediately suspended

  std::println("About to resume coroutine");
  h.resume();
}

// Prints:
// About to resume coroutine
// The coroutine has started
```

- [optional] **`Awaitable yield_value(T)`** – If this function exists, then the expression `co_yield e;` desugars to `co_await promise.yield_value(e);`, where `promise` is a fictitious reference to the promise object.
- [optional] **`void return_value(T)`** or **`void return_void()`** (but not both). In the former case `co_return e;` invokes `promise.return_value(e);` followed by `co_await promise.final_suspend();`. In the later case, `co_return;` (with no expression) or dropping off the end of the coroutine function invokes `promise.return_void();` followed by `co_await promise.final_suspend();`. It is illegal to define both `return_value` and `return_void` methods. It is legal to define neither for a coroutine that never uses `co_return`. However, if no `return_void` function is defined and execution drops off the end of the coroutine, the result is undefined behavior.
- **`void unhandled_exception()`** – If an exception escapes a coroutine before the coroutine has ever been suspended (and hence before the coroutine's return value has been returned), the exception is propagated to the code that invoked the coroutine func-

tion. Once the coroutine function has returned a value, however, it is not as obvious where to propagate the exception. Hence, C++ expects the promise object to handle the exception and invokes `promise.unhandled_exception();` followed by `co_await promise.final_suspend();`. If `unhandled_exception` itself throws an exception, it will escape `coroutine_handle::resume`, which is not usually someplace you will have enough information to handle the exception cleanly, so I recommend always defining `unhandled_exception` as `noexcept`.

Three reasonable ways for `unhandled_exception` to proceed are 1) to terminate the program with `std::terminate()`, 2) to save the value of `e = std::current_exception()` somewhere, and use `rethrow_exception(e)` when code checks the result of the coroutine (like `std::async`), or 3) to print some message and hope the program is recoverable by just finishing the coroutine as usual. An example of #3 might be:

```
void
promise_type::unhandled_exception() noexcept
{
  try {
    // re-throw current exception
    throw;
  } catch (const std::exception &e) {
    std::println(stderr, "Ignoring unhandled exception in coroutine: {}",
                 e.what());
    // Return as if coroutine had normally co_returned
  } catch (...) {
    std::println(stderr, "non-std::exception thrown");
    // Kill program assuming this was unexpected
    std::terminate();
  }
}
```

- **`Awaitable final_suspend() noexcept`** – The final action of a coroutine, after execution of `return_void`, `return_value`, or `unhandled_exception`, is conceptually to call `co_await promise.final_suspend();`. The method is defined separately from the return and exception handlers so that both paths invoke it.

  If the awaitable returned by `final_suspend` does not suspend the coroutine (e.g., see `SuspendedCoroPromise` above), the coroutine then deallocates itself. Otherwise, some other code must explicitly call `h.destroy()` on the coroutine's handle `h` to destroy the promise object and free the coroutine state. When the coroutine places a result in the promise object, you don't want it to deallocate itself on completion—rather, whatever code consumes the result should destroy the coroutine. In cases where a coroutine doesn't produce a result in the promise object and no one is waiting for the coroutine to complete, it's often useful for the coroutine to delete its own state by not suspending at `final_suspend`.

  Reasonable options to return from `final_suspend` include `std::suspend_never`,

`std::suspend_always`, or a custom awaitable that suspends the current coroutine but immediately resumes another one (by returning another handle from `await_suspend`, discussed below).

- [optional] **`Awaitable2 await_transform(Awaitable1)`** – A customization point that allows a promise object to override the behavior of an awaitable. If `promise.await_transform(a)` is valid, then `co_await a;` becomes equivalent to `co_await promise.await_transform(a);`.

Awaiter reference

The behavior of the expression `co_await a` in a coroutine is governed by `a`, which must be an *awaitable* object. The first thing C++ does is convert the awaitable into an *awaiter* (if it isn't one already). It does so by invoking `a.operator co_await()` or `operator co_await(a)` (using ADL). An Awaiter must supply the following three methods:

- **`bool await_ready()`** – If this method returns `false`, C++ will attempt to suspend the coroutine. If it returns `true` or throws an exception, C++ does not attempt to suspend the coroutine (i.e., it does not invoke `await_suspend`).
- **`R await_suspend(std::coroutine_handle<promise_type>)`** where `R` is `void`, `bool`, or `std::coroutine_handle<>` – This method receives the coroutine handle when C++ attempts to suspend a coroutine. You can have extra parameters with default arguments, for instance

  ```
  void await_suspend(std::coroutine_handle<promise_type> h,
                     std::source_location loc =
                         std::source_location::current())
  ```

  to trace where a coroutine is being suspended, though sadly this broke in gcc 15.2. If `await_suspend` throws an exception, the coroutine is not suspended. Otherwise, the behavior depends on the return type `R`:

  - `void` – the coroutine is suspended
  - `bool` – the coroutine is suspended only if the return value is `true`, not if it is `false`.
  - `std::coroutine_handle<>` – the coroutine is suspended, but then the coroutine corresponding to the returned handle is immediately resumed.

  Note that coroutine code runs in one of two ways: 1) when a coroutine function is initially invoked, or 2) after it has been suspended, when code invokes `h.resume()` (or equivalently `h()`) on the coroutine handle `h`. For the first two return types, the coroutine function or `resume` method returns when the coroutine is suspended (or when the coroutine finishes completely and is deallocated, in the case that `final_suspend()` does not suspend the coroutine).

  For the third return type, if the coroutine returns a `std::coroutine_handle<>`, then the returned coroutine is resumed before the coroutine function or `resume` returns. If

you use this return type and at runtime decide you don't have another coroutine to resume, you can always return `std::noop_coroutine()`, which immediately suspends itself.

- **`T await_resume()`** – This method returns the value of the `co_await` expression in the coroutine (or void when `co_await` does not return a value). The method can also throw an exception if `co_await` should throw.

The two standard library awaiters `std::suspend_always` and `std::suspend_never` return `false` and `true` from `await_ready`, respectively, discard any handle passed to `await_suspend`, and return `void` from `await_resume`.

Coroutine handle reference

Unlike promise objects and awaiters, `std::coroutine_handle` is already comprehensively documented at cppreference.com, so I'll just summarize a few of the most important methods.

- **`bool done() const`** – returns `true` if the coroutine has reached `final_suspend` and then suspended itself. Returns `false` if the coroutine has not yet reached that point. Constitutes a use-after-free bug if the coroutine reached `final_suspend` and did not suspend, because at that point the coroutine deallocated its own state, so the `coroutine_handle` is a dangling pointer.
- **`explicit operator bool() const noexcept`** – Since a coroutine handle is a pointer, it can be NULL. In a boolean context, the handle evaluates to true when non-NULL. Do not confuse this bool conversion with `done()`.
- **`void resume() const`** or the equivalent **`void operator()() const`** – resumes the coroutine after it suspended.
- **`void destroy() const`** – deallocates the coroutine frame, which must be called to avoid leaking memory except for coroutines whose `final_suspend` does not suspend.
- **`promise_type &promise() const`** – returns a reference to the promise object, but only in non-type-erased `std::coroutine_handle<promise_type>`, not in `std::coroutine_handle<>`.
- **`static coroutine_handle from_promise(promise_type &)`** – returns the coroutine handle given the promise object, which is useful for the promise object to obtain its own coroutine handle.

Summary

Here's a summary of the major actions that happen to coroutines:

- A coroutine is *allocated* when you invoke a coroutine function.
- A coroutine is *deallocated* when

  - The coroutine executes `co_return` (or falls off the end of a coroutine with a `return_void()` method) *and* the promise object `final_suspend()` does not suspend the coroutine, or
  - You call `h.destroy()` on the coroutine's handle `h`.
- A coroutine *executes* when
  - You invoke the coroutine function *and* `initial_suspend()` does not suspend the coroutine, or
  - The coroutine is suspended and you invoke `h.resume()` on its handle `h`.
- A coroutine is *suspended*, putting its handle in a state that can be resumed, when
  1. The coroutine evaluates `co_await a` or some expression that desugars to `co_await a` such as `co_yield v`, **and**
  2. The awaitable `a` transforms to an awaiter whose `await_ready()` method returns `false`, **and**
  3. The awaiter's `await_suspend()` method returns `void`, `true`, or a `std::coroutine_handle<>`.

Example infrastructure

For our example MVC bug, we need to create a coroutine, suspend it, and later resume it. To create coroutines, we will define a return type `Detached` with an associated promise type. To suspend coroutines, we will define a type `CoroSleep` that acts as a sleep queue on which coroutines can place themselves for some requested delay time.

`Detached` implementation

Let's define our return type `Detached` that represents a coroutine that doesn't return a value and that you can't wait for—much like a detached thread that can't be joined. Here is an implementation of such a type:

```
struct Detached {
  struct promise_type {
    Detached get_return_object() const noexcept { return {}; }
    std::suspend_never initial_suspend() const noexcept { return {}; }
    void return_void() const noexcept {}
    [[noreturn]] void unhandled_exception() const noexcept { std::terminate(); }
    std::suspend_never final_suspend() const noexcept { return {}; }
  };

  Detached(const Detached &) = delete;

private:
```

```
  Detached() noexcept = default;
};
```

Some notes about the above code. We don't need to specialize `std::coroutine_traits` here, because we just embed the `promise_type` right in the return type. `initial_suspend` never suspends the coroutine, so the coroutine function starts executing immediately and doesn't return until it suspends or co_returns. The coroutine is expected to co_return void (or fall off the end), so `final_suspend` never suspends, allowing the coroutine to be deallocated automatically on completion. Finally, we make the constructor private so that a function can't accidentally create a return value of type `Detached` if the programmer forgot to use the `co_return` operator.

We now have a way to create a coroutine—just invoke a function that returns `Detached`—but we still need something non-trivial to `co_await` within the coroutine.

`CoroSleep` implementation

Let's define an awaitable that suspends the current coroutine and resumes it in a specified amount of time. We'll start with a toy example to get the interface, then show how you might do this in production code.

Toy example

Here's the interface we are going for:

```
ToyCoroSleep tcs;

Detached
mycoro()
{
  std::println("about to sleep");
  co_await tcs.sleep(std::chrono::seconds(1));
  std::println("just slept");
}
```

The above code should print "just slept" one second after printing "about to sleep". To implement `ToyCoroSleep`, we just spawn a new thread that sleeps and then resumes the coroutine.

```
struct ToyCoroSleep {
  using clock = std::chrono::steady_clock;

  struct Awaiter {
    clock::time_point wake_time_;

    bool await_ready() noexcept { return false; }
    void await_suspend(std::coroutine_handle<> h) noexcept
```

```
    {
      std::thread([h, wake = wake_time_] {
        std::this_thread::sleep_until(wake);
        h();
      }).detach();
    }
    // Return success (for compatibility with CoroSleep)
    bool await_resume() noexcept { return true; }
  };
  Awaiter sleep(clock::duration d)
  {
    return Awaiter { .wake_time_ = clock::now() + d };
  }
};
```

The above code is sufficient to understand the most vexing capture bug examples, but you'd never want to use it in production—the overhead of spawning a thread will negate any benefit of having coroutines. In a real implementation, you would want to store coroutine handles in a list or tree sorted by wake time. That raises the problem of what to do when the tree or list is destroyed, because the coroutines will have to be resumed to avoid leaking them. That's why `await_resume()` returns a `bool`—we want to return true usually, but `false` if we may be resuming before the deadline because the queue is being destroyed.

Real code

Let's build a production-quality `CoroSleep` that overcomes the limitations of `ToyCoroSleep` above. We will use a `std::multimap waiters_` to store a mapping of wake times to coroutine handles. We will use a `std::mutex lock_` to protect the multimap. We'll use a `std::atomic_bool done_` to flag when the `CoroSleep` is being destroyed and no more sleepers should be inserted into the multimap. We'll use a `std::condition_variable recheck_` to notify a service thread that the earliest deadline may now be earlier than before. Finally, we'll use a service thread that resumes all the coroutines that need to be resumed.

Note that for the most part, `done_` will be protected by `lock_`, so all accesses can be relaxed. However, it needs to be atomic because `Awaiter::await_resume` reads it without holding the lock (albeit in a context where an indeterminate value is acceptable should there be a concurrent invocation of `CoroSleep`'s destructor). Using an atomic avoids undefined behavior from a data race. Here's the code:

```
class CoroSleep {
public:
  using clock = std::chrono::steady_clock;

  CoroSleep() : service_thread_(&CoroSleep::service_loop, this) {}

  ~CoroSleep()
```

```
  {
    lock_.lock();
    done_.store(true, std::memory_order_relaxed);
    recheck_.notify_all();
    lock_.unlock();

    service_thread_.join();
  }

  struct [[nodiscard]] Awaiter {
    CoroSleep *cs_;
    clock::time_point wake_time_;

    // Always attempt to suspend
    static bool await_ready() noexcept { return false; }

    // Insert coroutine in waiters.  Returns false (meaning don't
    // suspend) when CoroSleep is being destroyed.
    bool await_suspend(std::coroutine_handle<> h) noexcept
    {
      return cs_->insert(wake_time_, h);
    }

    // Return true while the CoroSleep is active, or false if it is
    // being destroyed and the wake time may not have passed.
    bool await_resume() const noexcept
    {
      return !cs_->done_.load(std::memory_order_relaxed);
    }
  };
  Awaiter sleep(clock::duration d)
  {
    return {.cs_ = this, .wake_time_ = clock::now() + d};
  }

private:
  // Return the next wakeup time.  Assumes lock_ is already held.
  clock::time_point next_wake() const
  {
    auto it = waiters_.begin();
    if (it == waiters_.end())
      return clock::time_point::max();
    return it->first;
  }
```

```
// Return a multimap of all coroutines whose wake time has arrived.
// If the CoroSleep is being destroyed, returns all coroutines.
// Assumes lock_ is already held.
std::multimap<clock::time_point, std::coroutine_handle<>> ready_coros()
{
  if (done_.load(std::memory_order_relaxed))
    return std::exchange(waiters_, {});

  decltype(waiters_) ready;
  auto it = waiters_.begin();
  auto end = waiters_.upper_bound(clock::now());
  while (it != end)
    ready.insert(waiters_.extract(it++));
  return ready;
}

// Insert a new waiter.  Fails and returns false if CoroSleep is
// being destroyed.
bool insert(clock::time_point wake_time, std::coroutine_handle<> h)
{
  std::lock_guard _lg(lock_);
  if (done_.load(std::memory_order_relaxed))
    return false;

  bool need_wake = wake_time < next_wake();
  waiters_.emplace(wake_time, h);
  if (need_wake)
    recheck_.notify_all();
  return true;
}

// Service routine for dedicated wakeup thread
void service_loop()
{
  std::unique_lock lk(lock_);
  for (;;)
    if (auto ready = ready_coros(); !ready.empty()) {
      // Release lock so resumed coros can call sleep() without deadlock
      lk.unlock();
      for (auto &[_time, h] : ready)
        h.resume();
      lk.lock();
    }
    else if (!done_.load(std::memory_order_relaxed))
      recheck_.wait_until(lk, next_wake());
```

```
      else
        return;
  }

  // Lock to protect waiters_ and done_.
  std::mutex lock_;

  // True when CoroSleep is being destroyed.  Atomic because
  // Awaiter::await_resume could load done_ concurrently with a store
  // in ~CoroSleep.  Even though we don't care what value load returns
  // in that situation, the spec says any such data race has undefined
  // behavior.
  std::atomic_bool done_;

  // All sleeping coroutines, indexed by wake time
  std::multimap<clock::time_point, std::coroutine_handle<>> waiters_;

  // Notified when inserting a coroutine with an earlier wake_time.
  std::condition_variable recheck_;

  // Dedicated thread to resume coroutines as they become ready
  std::thread service_thread_;
};
```

Most vexing capture bugs

The introduction already gave one example of an MVC bug:

```
void
do_n_things(int n)
{
  [n] -> Detached {   // Serious BUG!
    for (int i = 0; i < n; ++i)
      co_await thing();
  }(); // <-- lambda destroyed here
}
```

The unintuitive aspect of this code is that `n` is captured by value. As C++ programmers, we are constantly worried about capturing things by reference when the original might be destroyed before the reference. So capturing by value seems safe—it doesn't matter if the original `n` goes out of scope, right? In fact, we truly don't care about the original `n`, but thinking about that makes it easy to forget that the lambda is going to be destroyed, too.

The problem is that an unnamed lambda expression such as `[n] -> Detached { ... }()` creates a temporary lambda object (containing the captures) that is destroyed at the end of the function call expression. That's true for both ordinary lambdas and coroutine lambdas.

It's not a problem for ordinary lambdas, because by the time a non-coroutine lambda returns, you no longer need the lambda object. In fact, calling temporary lambdas is a widely used pattern we grow accustomed to seeing. However, a coroutine typically persists beyond the return of the coroutine function that created it. That makes it a serious problem when the coroutine function is a lambda object's `operator()` method, because the lambda object is destroyed on return of the function call, not on completion of the coroutine.

Here's a table of what's happening in `do_n_things` and the anonymous lambda as time progresses:

`do_n_things`	lambda
create coroutine lambda	
invoke coroutine lambda	
	allocate coroutine
	`co_await thing()`
	suspend coroutine
lambda returns `Detached{}`	
destroy lambda	
	1 second elapses
	resume coroutine
	`i < n` (MVC bug)
	⋮
	deallocate coroutine

Think you've got it? Test yourself against the following example and see if you can find any MVC bugs:

```
using namespace std::literals::chrono_literals;

CoroSleep cs;

Detached
print_loop(std::latch &done)
{
  for (int i = 0; i < 3 && co_await cs.sleep(1s); ++i)
    std::println("function {}", i);
  done.count_down();
}

int
main()
{
  std::latch done{4};
```

```
  // 1. Coroutine function
  print_loop(done);

  // 2. Named coroutine lambda
  auto lambda = [&done] -> Detached {
    for (int i = 0; i < 3 && co_await cs.sleep(1s); ++i)
      std::println("named lambda {}", i);
    done.count_down();
  };
  lambda();

  // 3. Anonymous coroutine lambda with no captures
  [](std::latch &done) -> Detached {
    for (int i = 0; i < 3 && co_await cs.sleep(1s); ++i)
      std::println("latch argument {}", i);
    done.count_down();
  }(done);

  // 4. Anonymous coroutine lambda with capture
  [&done] -> Detached {
    for (int i = 0; i < 3 && co_await cs.sleep(1s); ++i)
      std::println("latch capture {}", i);
    done.count_down();
  }();

  done.wait();
}
```

Did you catch which of the above are correct and incorrect? All of 1–3 are correct, while 4 is an MVC bug. As before, the problem with 4 is that the lambda is an unnamed temporary, destroyed at the end of the full expression. So the lambda is destroyed when the *coroutine function* returns an object of type `Detached`, which is before the *coroutine* itself implicitly "co_returns" void by dropping off the end of the function definition. My brain looks at the capture `[&done]` and says, "Uh oh, you are capturing by reference, are you sure `done` won't go out of scope before the coroutine ends? No, it's okay because `done` is a latch that won't be destroyed until after `done.wait()`, which won't happen until all the coroutines are finished accessing `done`." But focusing on the lifetime of `done` is a distraction—the actual issue is the lifetime of the lambda that contains the address of `done`. At the implementation layer, capturing a reference means storing a pointer in the lambda, and that pointer just resides in the lambda object, on the stack, in memory that is available for reuse as soon as the coroutine is first suspended.

Here's another bug from real code. This is glue code intended to allow non-coroutine code to wait for a coroutine awaiter. Can you spot the problem?

```
template<typename A>
```

```
concept is_awaiter = requires(A a) {
  { a.await_ready() } -> std::same_as<bool>;
  a.await_suspend(std::noop_coroutine());
  a.await_resume();
};

// incorrect code:
auto
await(is_awaiter auto &awaiter)
{
  if (!awaiter.await_ready()) {
    std::binary_semaphore wait(0);
    auto h = [&wait] -> Suspended {
      wait.release();
      co_return;
    }();

    try {
      using sus_type = decltype(awaiter.await_suspend(h));
      if constexpr (std::same_as<sus_type, void>)
        awaiter.await_suspend(h);
      else if constexpr (std::same_as<sus_type, bool>) {
        if (!awaiter.await_suspend(h))
          // Since await_suspend didn't suspend, it won't later resume
          // h, either.  For simplicity let's just resume it ourselves.
          // That releases the semaphore and deallocates the coroutine.
          h.resume();
      }
      else {
        // sus_type must be std::coroutine_handle<P>
        auto h2 = awaiter.await_suspend(h);
        h = nullptr;
        h2.resume();
      }
    } catch (...) {
      if (h)
        h.destroy();
      throw;
    }

    wait.acquire();
  }
  return awaiter.await_resume();
}
```

Once again, the lambda capturing `wait` is destroyed before the suspended coroutine is resumed. However, superficially this code looks like 2 (named coroutine lambda) in the previous example. I literally looked at the code, asked myself "is there an MVC problem?" thought about it, and concluded that it was fine since `h` was a named variable. Except obviously `h` here is the result of the lambda, not the lambda itself, meaning the code is incorrect.

Avoiding MVC bugs

There are two ways to avoid the above problem. The first is just not to capture anything in coroutines. You can instead pass any necessary values to the coroutine as parameters, which will be copied into the coroutine frame and persist even if the lambda is destroyed. This is fine, but a bit redundant because it requires naming the capture twice, for instance:

```
void
do_n_things(int n)
{
  [](int n) -> Detached {   // Ok
    for (int i = 0; i < n; ++i)
      co_await thing();
  }(n);
}
```

The second and better option is to copy the lambda itself into the coroutine by means of the versatile C++23 deducing this feature. Briefly, deducing this allows a method to get its object explicitly as the first argument, rather than implicitly through the `this` keyword. To use the feature, you must tag the first parameter of a method with the keyword `this`, which tells the compiler to use that parameter (by convention often called `self`) as the object. Note that while an implicit `this` is a pointer, an explicit `this` parameter has reference semantics, making it possible to infer whether the object is an lvalue or rvalue.

As an example, in the code below, calling `substr()` on an object of type `DerivedString` returns a `DerivedString`, not a BaseString, because `BaseString::substr` deduces the type of `this` as `DerivedString`:

```
struct BaseString {
   //...
   template<typename Self>
   Self substr(this const Self &self, size_t pos,
               size_t count = Self::npos)
   {
     if (count == Self::npos)
       count = self.size() - pos;
     return Self(self.begin() + pos, self.begin() + pos + count);
   }
};
```

```
struct DerivedString : BaseString {
  //...
  using BaseString::BaseString;
};
```

The `this` argument is almost always passed by reference, often as a forwarding reference (`this auto &&self`). However, C++ doesn't require that it be a reference. You can also copy an object into a method by passing `this` by value—which is especially useful in the case of preserving lambda captures:

```
void
do_n_things(int n)
{
  [n](this auto) -> Detached { // OK because this is not a reference
    for (int i = 0; i < n; ++i)
      co_await thing();
  }();
}
```

The parameter `(this auto)` causes the lambda object to be passed by value into the coroutine frame. The lambda captures (including values and references) become part of the coroutine's state, extending their lifetime to match the coroutine's lifetime. That means avoiding MVC is as simple as adding an unnamed `(this auto)` argument to any unnamed lambda coroutine with captures!

One final thing to point out is that when you define a lambda within a method, the `this` argument to the lambda refers to the lambda, while a `this` capture refers to the object. It's perfectly reasonable for a lambda to capture a `this` argument—when captured by reference, it's the easiest way to define a recursive lambda. However, on occasion one of my coworkers has seen internal compiler errors from capturing both `this` arguments, as in `[this](this auto) { /* ... */ }()`. An easy workaround is just to rename the captured `this` to `self`, as in `[&self = *this](this auto) { /* ... */ }()`, at which point you access the object as `self.field_` or `self.method()`.