# C++ coroutines: Snooping in on the coroutine body

devblogs.microsoft.com/oldnewthing/20210428-00

April 28, 2021

Raymond Chen

A coroutine promise can snoop on the coroutine body by implementing a method named `await_transform`. Any time the coroutine body performs a `co_await`, the thing being awaited is passed through the `await_transform` method, and whatever `await_transform` returns is the thing that is *actually* awaited. This is the mysterious "We're not ready to talk about step 1 yet" that kept reappearing in our introduction to awaitable objects.

One way that `await_transform` is used is to provide the coroutine body a way to communicate with the promise, by `co_await`'ing an object with a sentinel type. This is the magic behind secret signals like `co_await get_cancellation_token()`.

Let's use this to allow the coroutine to configure the promise's `unhandled_exception` behavior.

```
namespace async_helpers::details
{
    struct simple_promise_policies
    {
        bool m_terminate_on_unhandled_exception = false;
    };
}

namespace async_helpers
{
    template<typename T>
    struct simple_task;

    struct simple_task_policy
    {
        simple_task_policy(details::simple_promise_policies& policies)
            : m_policies(policies) {}

        bool terminate_on_unhandled_exception(bool value = true)
            const noexcept
        {
            return std::exchange(
                m_policies.m_terminate_on_unhandled_exception,
                value);
        }
    private:
        details::simple_promise_policies& m_policies;
    };

    struct get_simple_task_policy {};
}
```

We start by declaring a private structure `simple_promise_policies` to hold our simple promise policies. So far, the only policy is whether to terminate on unhandled exception. You can imagine adding additional runtime policies here when they occur to you.

We then provide a public structure `simple_task_policy` that wraps the private one. This is what the coroutine itself uses to alter the behavior of the promise.

For now, there is only one method on the policy object, namely `terminate_on_unhandled_exception()` which specifies whether you want the coroutine to terminate if an unhandled exception occurs. The default is `true`, and the method returns the previous setting in case you want to restore it later.

Finally, we define a marker structure `get_simple_task_policy`. The purpose of this structure will become apparent later.

```
namespace async_helpers::details
{
    template<typename T>
    struct simple_promise_base
    {
        ...
        simple_promise_policies m_policies;

        ...

        void unhandled_exception() noexcept
        {
            if (m_policies.m_terminate_on_unhandled_exception)
            {
                std::terminate();
            }
            m_holder.unhandled_exception();
        }
        ...
```

We add a policies object to our `simple_promise_base`, and the `unhandled_exception` method consults the policy to decide whether to terminate when an unhandled exception occurs, or whether to stow the exception in the holder for later rethrowing when the coroutine is `co_await`ed.

```
        // still in struct simple_promise_base<T>
        auto await_transform(get_simple_task_policy) noexcept
        {
            struct awaiter : std::experimental::suspend_never
            {
                simple_promise_policies& policies;
                auto await_resume() const noexcept
                {
                    return simple_task_policies(policies);
                }
            };
            return awaiter{ {}, m_policies };
        }
```

This is where the magic happens, the mysterious step 1.

If the coroutine promise has a method called `await_transform`, then every `co_await` is passed to the `await_transform` method, and the thing it returns is the thing that is actually awaited. This is how the coroutine promise can snoop on all `co_await` activity that occurs inside the coroutine body.

One use of `await_transform` is for the coroutine to inject some code at every potential suspension point. For example, it could do some extra bookkeeping when suspension occurs, and again when the coroutine resumes.

That's not what we're going to use it for, though.

In our case, we have an overload that takes a `get_simple_task_policy` object. Any attempt to `co_await` one of those objects will trigger a call to this overload of `await_transform`, and the overload ignores the parameter and instead returns a custom awaitable whose sole purpose is to return a `simple_task_policies` object that wraps the promise's policy object.

That's what makes `await_transform` special: Your basic awaitable object doesn't know what coroutine is awaiting it. But `await_transform` is a member of the promise, and therefore it can create an awaitable that is in cahoots with its promise.

It is typical for the custom awaiter for these backchannel communications awaitables not to suspend at all and just produce the desired value in the `await_resume`.

A coroutine that uses the `simple_promise` can use this secret signal like this:

```
async_helpers::simple_task<void> Example()
{
    auto policy = co_await
        async_helpers::get_simple_task_policy();
}
```

There appear to be a few schools of thought on how these secret signals should be made.

- One school uses marker structures with default constructors. That's what we did here with `get_simple_task_policy`.
- Another school uses marker structures that are returned by purpose-built functions. That's how C++/WinRT does things.
- A third school of thought uses premade sentinel objects.

An implementation that follow the second school would go like this:

```
struct get_simple_task_policy_t {};
inline constexpr get_simple_task_policy_t
get_simple_task_policy()
{
    return {};
}

async_helpers::simple_task<void> Example()
{
    auto policy = co_await
        async_helpers::get_simple_task_policy();
}
```

The inline function `get_simple_task_policy()` returns an instance of the marker.

An implementation that followed the third school would go like this:

```
struct get_simple_task_policy_t {};
inline constexpr get_simple_task_policy_t get_simple_task_policy;

async_helpers::simple_task<void> Example()
{
    auto policy = co_await
        async_helpers::get_simple_task_policy;
}
```

An advantage of the first two designs is that you can parameterize the secret signal. For example, you could have

```
co_await report_progress(50);
```

to report that you were 50% done.

An advantage of the third design is that it removes a set of pesky parentheses.

Unfortunately, we can't just stop there, because the language rules say that `await_transform` is all-or-nothing. If you have *any* `await_transform` method, then you must handle *all* possible awaitables. In order to soak up the other awaitables and pass them through, we need to add

```
        // in struct simple_promise_base<T>
        template<typename U>
        U&& await_transform(U&& awaitable) noexcept
        {
            return static_cast<U&&>(awaitable);
        }
    }
```

The `await_transform` lets you insert code into every `co_await` operation, but sadly the mechanism to do this is quite cumbersome because wrapping the existing awaitable requires you first to *find* the awaitable by replicating the algorithm used by the compiler. You end up having to do a bunch of SFINAE to look for a `co_await` operator that will produce the awaitable you need to wrap, or give up and use the object as its own awaiter. (And good luck expressing the operator overload conflict resolution algorithm in template metaprogramming.)

Raymond Chen

**Follow**