In C++/WinRT, how can I await multiple coroutines and capture the results?, part 3

devblogs.microsoft.com/oldnewthing/20240112-00

January 12, 2024

Raymond Chen

We saw last time that our `when_all_with_results` function had problems if any of the awaitables completed with a C++ reference or `void`. Let's look at the `void` case first.[1]

One idea is to just remove any `void`-things from the tuple. Though it might look weird at the call site:

```
auto [result1, result3] = co_await when_all_with_results(
        Do1Async(), Do2Async(), Do3Async());
```

"I'm awaiting three things, but somehow only two things came out?"

Another idea would be to transform a `void` to `std::monostate`. That way, you can write

```
auto [result1, ignore, result3] = co_await when_all_with_results(
    co_await when_all_with_results(
        Do1Async(), Do2Async(), Do3Async());
```

Unfortunately, this potentially triggers a "variable never used" error on `ignore`, and the language doesn't currently[2] let you write

```
auto [result1, [[maybe_unused]] ignore, result3] =
    co_await when_all_with_results(
        Do1Async(), Do2Async(), Do3Async());
```

You could receive the tuple and extract the values you want.

```
auto result = co_await when_all_with_results(
        Do1Async(), Do2Async(), Do3Async());
auto&& result1 = std::get<0>(std::move(result));
auto&& result2 = std::get<2>(std::move(result));
```

but that's the sort of syntax only a mother would love.

Or you could use `std::tie`:

```
ResultType1 result1;
ResultType2 result2;

std::tie(result1, std::ignore, result2) = co_await
    co_await when_all_with_results(
        Do1Async(), Do2Async(), Do3Async());
```

This has the benefit of making it clearer that the second awaitable's result is being discarded. It has the downside of requiring the result types to be default-constructible with no unwanted side effects. It also loses references, so you generate extra copies if `DoAsync()` functions completed with references.

Even though there's no good syntax for consuming a tuple with an ignored element, it's still arguably better than just returning a smaller tuple.

Okay, so let's do it. One way is to wrap the awaiter.

```
template<typename Inner>
struct void_to_monostate_awaitable_wrapper
{
    void_to_monostate_awaitable_wrapper(Inner& inner) :
        m_awaiter(awaiter_finder::get_awaiter(std::move(inner))) {}

    typename awaiter_finder::type<Inner> m_awaiter;

    bool await_ready()
    { return m_awaiter.await_ready(); }

    template<typename Handle>
    auto await_suspend(Handle handle)
    { return m_awaiter.await_suspend(handle); }

    using AsyncResult = decltype(m_awaiter.await_resume());
    static constexpr bool is_async_result_void =
        std::is_same_v<AsyncResult, void>;
    using Result = std::conditional_t<
        is_async_result_void, std::monostate, AsyncResult>;

    AsyncResult await_resume() {
        if constexpr (is_async_result_void) {
            m_awaiter.await_resume();
            return std::monostate{};
        } else {
            return m_awaiter.await_resume();
        }
    }
};
```

This awaitable wrapper wraps the original awaiter, but if the original awaiter's `await_resume()` returns `void`, we change it to `std::monostate`.

```
template<typename... Asyncs>
using when_all_result = std::tuple<
    typename void_to_monostate_awaitable_wrapper
        <Asyncs>::Result...>;

template<typename... Asyncs>
wil::task<when_all_result<Asyncs...>>
when_all_with_results(Asyncs... async)
{
    co_return when_all_result<Asyncs...>>(
        co_await void_to_monostate_awaitable_wrapper(async)...);
}
```

Another way is to wrap the awaitable inside another coroutine.

```
template<typename Async>
struct awaitable_traits
{
    using Awaiter = typename awaiter_finder::type<Async>;
    using Result = decltype(std::declval<Awaiter>().await_resume());
    static constexpr bool is_void_result =
        std::is_same_v<Result, void>;
    using TransformedResult = std::conditional_t<
        is_void_result, std::monostate, Result>;
};

template<typename... Asyncs>
using when_all_result = std::tuple<
    typename awaitable_traits<Asyncs>::TransformedResult...>;

template<typename... Asyncs>
wil::task<when_all_result<Asyncs...>>
when_all_with_results(Asyncs... async)
{
    co_return when_all_result<Asyncs...>(
        co_await [async = std::move(async)]()
            -> wil::task<typename awaitable_traits<Asyncs>::TransformedResult> {
            if constexpr (awaitable_traits<Asyncs>::is_void_result) {
                co_await std::move(async);
                co_return std::monostate{};
            } else {
                co_return co_await std::move(async);
            }
        }()...);
}
```

I'm not a fan of this approach because it makes everything a `wil::task`, even if the original awaitable used some other framework.

Either way, that sure was an awful lot of typing, and we haven't even dealt with the case of what to do if one of the awaitables encounters an exception. Do we want to throw the first exception we encounter? Do we want to capture them and return a tuple of `std::variant<T, std::exception_ptr>`s, so the caller can decode which awaitables failed and which succeeded?

As with our lengthy discussion of waiting for multiple C++ coroutines to complete before propagating failure, the answer appears to be "Why are you even bothering to write this function at all?": The `when_all` helper is really just a crutch. You may as well just await them all yourself at the call site.

```
auto op1 = Do1Async();
auto op2 = Do2Async();
auto op3 = Do3Async();

auto result1 = co_await op1;
co_await op2;
auto result3 = co_await op3;
```

Now it's obvious that you're starting three operations, letting them run in parallel (assuming they are hot-start), saving the results of the first and third, and ignoring the results of the second. The exception handling story is also obvious from the code: We report exceptions from `op1` first, then `op2`, and then `op3`.

[1] The stalled Regular Void proposal would have allowed for it.

[2] There is a C++26 proposal for attributes for structured bindings.