Synthesizing a when_all coroutine from pieces you already have

devblogs.microsoft.com/oldnewthing/20200902-00

September 2, 2020

Raymond Chen

C++/WinRT provides a helper function that takes a bunch of awaitable objects and produces an `IAsyncAction` that completes when all of the awaitable objects have completed.

It has a very simple definition.

```
template <typename... T>
Windows::Foundation::IAsyncAction when_all(T... async)
{
    (co_await async, ...);
}
```

Let's take this apart.

The opening `template<typename... T>` says that this is a template that takes an arbitrary number of type parameters.

The function prototype is for a function which takes a parameter list of `T... async`. This means that you can pass as many parameters as you like, of whatever type you prefer, and they are accepted by value. The parameter list is given the name `async`.

The body is `(co_await async, ...)`. This is a *fold expression*. If `async...` represents the parameter list $\texttt{async}_1$, $\texttt{async}_2$, $\texttt{async}_3$, $\texttt{async}_4$, then

```
(co_await async, ...)
```

expands to

$$(\texttt{co\_await async}_1, \texttt{co\_await async}_2, \texttt{co\_await async}_3, \texttt{co\_await async}_4)$$

Usually, fold expressions are used with operators like `+` or `||`:

```
(v + ...)
```

expands to

```
(v₁ + v₂ + v₃ + v₄)
```

and

```
(is_even(v) || ...)
```

expands to

```
(is_even(v₁) || is_even(v₂) || is_even(v₃) || is_even(v₄))
```

for example.[1] Here, we're using the comma operator not for anything interesting; it's just a way to execute a bunch of stuff.

The end result of this all is that if you write `when_all(x, y, z)`, this becomes

```
Windows::Foundation::IAsyncAction when_all(X x, Y y Z z)
{
  (co_await x, co_await y, co_await z);
}
```

This produce a coroutine which awaits `x`, then throws the result away; then awaits `y`, then throws the result away; and finally awaits `z`, then throws the result away. And then the coroutine is finished.

Mid-article bonus chatter: There are some flaws in the above function. We'll look at them next time. **End of bonus chatter**.

A customer wanted to know how they could pass a `std::vector` of `IAsyncAction` objects to the `when_all` function.

It reminds me of the old *Sesame Street* sketch where Grover has no trouble counting blocks, but when asked to count some oranges, Grover freezes up. "I know how to count blocks, but I do not know how to count oranges!"

I have to confess that as I child, I didn't get the joke.

Anyway, we saw how to count blocks (await every object in a parameter list). We just need to count oranges (await every object in a vector).

```
std::vector<IAsyncAction> actions = get_actions();
for (auto&& action : actions) co_await action;
```

We can try to wrap this up in a function:

```
template<typename T>
IAsyncAction when_all(T const& container)
{
  for (auto&& v : container) co_await v;
}
```

This doesn't work because there is an ambiguity in the case where there is one parameter. Are you trying to await all of the awaitables in a list of length 1? Or is the parameter a container, and you want to await all objects within it?

I'll say that if the single parameter has a method named `begin` whose return type is not `void`, then it's a container. (I could try to do better by also accepting a free function `begin`, but I'm feeling lazy.)

```
template<typename T>
auto when_all(T&& container) ->
  std::enable_if_t<sizeof(container.begin()) >= 0, IAsyncAction>
{
  for (auto&& v : container) co_await v;
}
```

I'm using `sizeof` as a way to create a constant `true` value from a dependent type, so it can be tested with `std::enable_if_t`. We know that the container's iterator must be a complete type because we're going to use it in the `for` loop.

We might also want to support a range expressed as two input iterators.[2]

```
template<typename Iter>
std::enable_if_t<
  std::is_convertible_v<
    typename std::iterator_traits<Iter>::iterator_category,
    std::input_iterator_tag>, IAsyncAction>
when_all(Iter begin, Iter end)
{
  for (; begin != end; ++begin) co_await *begin;
}
```

In all of these cases, you need to make sure to keep the container or range alive until after the `co_await when_all(...)` completes.

Whatever way you come up with to express a collection of awaitable objects, you can write a function that accepts that collection and awaits each object in the collection.

Go ahead and count oranges.

[1] More precisely, they expand to

```
(v₁ + (v₂ + (v₃ + v₄)))
```

and

```
(is_even(v₁) || (is_even(v₂) || (is_even(v₃) || is_even(v₄))))
```

If you want the left-associative version, then you need to put the ellipsis on the left.

```
(... + v)
(... || is_even(v))
```

[2] For extra flexibility, we could implicitly convert the second argument to match the first.

```
// C++17
when_all(Iter begin, std::enable_if_t<true, Iter> end)

// C++20
when_all(Iter begin, std::type_identity_t<Iter> end)
```

Raymond Chen

Follow