Creating an already-completed asynchronous activity in C++/WinRT, part 3

devblogs.microsoft.com/oldnewthing/20240711-00

July 11, 2024

Raymond Chen

Last time, we figured out how to create an already-completed asynchronous activity in C++/WinRT. Today we'll try to generalize it to cover the four kinds of Windows Runtime asynchronous activities.

	No progress	Progres
No result	`IAsyncAction`	`IAsyncActionWithProgress<P>`
Result	`IAsyncOperation`	`IAsyncOperationWithProgress<T, P>`

One way to do this is to write four different functions for each category, similar to how we dealt with *cv*-qualifiers before we had deducing this.

```
winrt::Windows::Foundation::IAsyncAction
MakeCompletedAsyncAction()
{
    co_return;
}

template<typename Progress>
winrt::Windows::Foundation::IAsyncActionWithProgress<Progress>
MakeCompletedAsyncActionWithProgress()
{
    co_return;
}

template<typename Result, typename Progress>
winrt::Windows::Foundation::IAsyncOperation<Result>
MakeCompletedAsyncOperation(Result result)
{
    co_return result;
}

template<typename Result, typename Progress>
winrt::Windows::Foundation::IAsyncOperationWithProgress<Result, Progress>
MakeCompletedAsyncOperationWithProgress(Result result)
{
    co_return result;
}

// Sample usage:

winrt::Windows::Foundation::IAsyncOperation<int>
GetHeightAsync()
{
    return MakeCompletedAsyncOperation(42);
}

winrt::Windows::Foundation::
    IAsyncOperationWithProgress<int, HeightProgress>
GetHeightAsync()
{
    return MakeCompletedAsyncOperationWithProgress<
        int, HeightProgress>(42);
}
```

Explicit specialization is required for the `WithProgress` versions, since there is no opportunity to deduce the progress type.

We could combine the four flavors into a single function, though this means that specialization is mandatory.

```
template<typename Async, typename... Result>
Async MakeCompleted(Result... result)
{
    if constexpr (sizeof...(Result) == 0) {
        co_return;
    } else {
        static_assert(sizeof...(Result) == 1);
        co_return std::get<0>(
            std::forward_as_tuple(result...));
    }
}
```

We use a trick in `MakeCompleted` by formally accepting any number of arguments, although we check inside the function body that it is zero or one. In the case where there is one parameter, we use the `forward_as_tuple` + `get` technique to pull a single item from a parameter pack.

Next time, we'll try to write `MakeFailed`.