Making our multiple-interface query more C++-like, part 1

devblogs.microsoft.com/oldnewthing/20220317-00

March 17, 2022

Raymond Chen

The `MULTI_QI` structure we've been looking at is kind of awkward to use, but maybe we can improve on it with some C++ magic.

But before we start, we need to decide what we want. How about making it possible to write this:

```
// Into new variables
auto [widget, objectWithSite, persistFile] =
    CreateInstanceMultiQI<IWidget, IObjectWithSite, IPersistFile>
        (CLSID_Widget, nullptr, CLSCTX_LOCAL_SERVER);

// Into existing variables
wil::com_ptr<IWidget> widget;
wil::com_ptr<IObjectWithSite> objectWithSite;
wil::com_ptr<IPersistFile> persistFile;

std::tie(widget, objectWithSite, persistFile) =
    CreateInstanceMultiQI<IWidget, IObjectWithSite, IPersistFile>
        (CLSID_Widget, nullptr, CLSCTX_LOCAL_SERVER);

// Preloading interfaces, keeping only one for now.
wil::com_ptr<IWidget> widget;

std::tie(widget, std::ignore, std::ignore) =
    CreateInstanceMultiQI<IWidget, IObjectWithSite, IPersistFile>
        (CLSID_Widget, nullptr, CLSCTX_LOCAL_SERVER);
```

with the rule that any interfaces that couldn't be obtained are returned as empty `com_ptr`s. (We'll deal with required interfaces later.)

Here's my idea. We'll start with pseudo code and gradually fill it in.

```
template<typename... Interfaces>
std::tuple<wil::com_ptr<Interfaces>...>
CreateInstanceMultiQI(
    REFCLSID clsid, IUnknown* punkOuter,
    DWORD clsctx)
{
    MULTI_QI mqi[] = {
        { &__uuidof(Interfaces), nullptr, 0 }...
    };

    THROW_IF_FAILED(
        CoCreateInstanceEx(clsid, punkOuter, clsctx,
                           sizeof...(Interfaces), mqi));

    std::tuple<wil::com_ptr<Interfaces>...> t;

    for (Index = 0; Index < sizeof...(Interfaces); Index++) {
        std::get<Index>(t).
            attach(static_cast<Interfaces[Index]*>
                        (mqi[Index].pItf));
    }

    return t;
}
```

First, we build an array of `MULTI_QI` structures initialized with the interface identifiers corresponding to the requested interfaces.

Next, we call `CoCreateInstanceEx` with that array of `MULTI_QI` structures to create the object and query multiple interfaces. If we are unable to create the object, or if the object supports none of the interfaces, then we throw the failure.

Otherwise, we have something to return. Create the output tuple and attach each of the `MULTI_QI` results to the corresponding slot in the tuple. The interfaces in the `MULTI_QI` are all represented as `IUnknown*`, so we `static_cast` them to the requested interface. The `static_cast` also validates that all of the requested interfaces derive from `IUnknown`.[1]

Okay, now that we have this sketch, we have to turn it into real C++.

First, we'll fix the initialization of the `MULTI_QI` array.

```
    MULTI_QI mqi[] = {
        MULTI_QI{ &__uuidof(Interfaces), nullptr, 0 }...
    };
```

You cannot expand a parameter pack into a series of braced initializers, so we make it an explicit aggregate construction of a `MULTI_QI`. That converts the braced initializers into an expression, and expressions support parameter pack expansion. That one was easy to fix.

The harder part is the weird `for` loop that iterates over a parameter pack. For that, we need help from our old friend the index sequence. (Previous series on index sequences.)

```
template<typename... Interfaces, std::size_t... Ints>
auto TupleFromMultiQi(
    MULTI_QI* mqi, std::index_sequence<Ints...>)
{
    std::tuple<com_ptr<Interfaces>...> t;
    ((std::get<Ints>(t).attach(
        static_cast<Interfaces*>(mqi[Ints].pItf))), ...);
    return t;
}
```

There are a few interesting things going on here.

First of all, this template function has *two* parameter packs. This is allowed in template functions if everything after the first template parameter pack can be deduced. In our case, the indices can be deduced from the `index_sequence` parameter.

The second trick here is that we are expanding two parameter packs in a single expansion. If you do this, the packs must be of the same size, and they are expanded in parallel with corresponding elements.

Now we can put everything together.

```
template<typename... Interfaces, std::size_t... Ints>
auto CreateInstanceMultiQIWorker(
    REFCLSID clsid, IUnknown* punkOuter,
    DWORD clsctx, MULTI_QI* mqi,
    std::index_sequence<Ints...>)
{
    THROW_IF_FAILED(
        CoCreateInstanceEx(clsid, punkOuter, clsctx,
                           sizeof...(Interfaces), mqi));

    std::tuple<com_ptr<Interfaces>...> t;
    ((std::get<Ints>(t).attach(
        static_cast<Interfaces*>(mqi[Ints].pItf))), ...);
    return t;
}

template<typename... Interfaces>
std::tuple<wil::com_ptr<Interfaces>...>
CreateInstanceMultiQI(
    REFCLSID clsid, IUnknown* punkOuter,
    DWORD clsctx)
{
    MULTI_QI mqi[] = {
        { &__uuidof(Interfaces), nullptr, 0 }...
    };

    return CreateInstanceMultiQIWorker<Interfaces...>
        (clsid, punkOuter, clsctx, mqi,
         std::index_sequence_for<Interfaces...>{});
}
```

I moved the `CoCreateInstanceEx` into the helper function as well, to facilitate COMDAT folding: All instantiations of `CreateInstanceMultiQIWorker` with the same number of interfaces will expand to the same code, because the `static_cast` does nothing, and the `attach` does the same thing regardless of the type. The only thing that affects the code generation is the number of interfaces. Therefore, all the instantiations will be shared, and only one copy will go into the final binary.

This version treats all interfaces as optional. The only requirement is that at least *one* of them be supported. Next time, we'll look at the case where some interfaces are required and others are optional. For example, you might require `IWidget` support, but `IObjectWithSite` support is optional.

[1] The static cast also succeeds if `Interface` is `void`. Fortunately, `__uuidof(void)` is not defined, so that problem is caught elsewhere.

Raymond Chen

Follow