Making our multiple-interface query more C++-like, part 2

devblogs.microsoft.com/oldnewthing/20220318-00

March 18, 2022

Raymond Chen

Last time, we wrote a C++-style wrapper around `CoCreatInstanceEx` that treats all of the interfaces as optional. But probably, you are issuing the multiple-interface query in the case where some or perhaps even all of the interfaces are required.

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

The `IWidget` is required. Without that, you have nothing. But maybe the `IObjectWithSite` is optional: If the widget doesn't support `IObjectWithSite`, we just won't set a site on it.

We need a way to tell our `CreateInstanceMultiQI` function that some of the interfaces are required and others are optional.

I'm going to pull a dirty trick. and instead of creating my own marker, I'm going to reuse an existing template from the standard library, since it has such a great name: `std::optional`. I'm going to say that all of the interfaces are required by default, but you can wrap them inside a `std::optional` to say that they're optional.

```
auto [widget, objectWithSite, persistFile] =
    CreateInstanceMultiQI<
        IWidget,
        std::optional<IObjectWithSite>,
        IPersistFile>
        (CLSID_Widget, nullptr, CLSCTX_LOCAL_SERVER);
```

Most of the stuff we wrote last time still works. We just need to add a validation step that verifies that all of the required interfaces were successfully obtained.

```
template<typename Interface>
struct multiqi_traits
{
    using type = Interface;
    static constexpr bool is_required = true;
};

template<typename Interface>
struct multiqi_traits<std::optional<Interface>>
{
    using type = Interface;
    static constexpr bool is_required = false;
};

template<typename Interface>
using multiqi_traits_com_ptr =
    wil::com_ptr<typename multiqi_trauts<Interface>::type>;
```

The `multiqi_traits` template traits type assumes that every interface is required. The specialization unwraps any interface that is wrapped inside a `std::optional` and remembers that it is not required. For convenience, we also define a `multiqi_traits_com_ptr` that represents the final `com_ptr` we want to return.

We can use this traits type to modify our existing function to throw if any required interface was not obtained:

```
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<multiqi_traits_com_ptr<Interfaces>...> t;
    ((std::get<Ints>(t).attach(
        static_cast<typename multiqi_traits<Interfaces>::type*>
        (mqi[Ints].pItf))), ...);

    ([&] {
        if constexpr (multiqi_traits<Interfaces>::is_required) {
            THROW_IF_FAILED(mqi[Ints].hr);
        }
    }(), ...);

    return t;
}

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

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

The first change is mechanical: We have to use `multiqi_traits` to unwrap the elements of `Interfaces...`, because some of them may be `std::optional<T>`, and in those cases, we want to reach inside the `std::optional` and extract the `T`.

The new part is the template parameter pack expansion of a lambda invocation. Template parameter pack expansions can expand expressions, but we need to expand an `if` statement. No problem: We wrap the `if` statement in a lambda, and then evaluate the lambda immediately, and then take the lambda evaluation (now an expression!) and make the evaluation the thing that is given to the template parameter pack expansion.

Note that we perform the validation as a separate step after transferring the raw interface pointers into the tuple. This attempted optimization would be incorrect:

```
    ([&] {
        std::get<Ints>(t).attach(
        static_cast<typename multiqi_traits<Interfaces>::type*>
        (mqi[Ints].pItf));

        if constexpr (multiqi_traits<Interfaces>::is_required) {
            THROW_IF_FAILED(mqi[Ints].hr);
        }
    }(), ...);
```

If any of the required interfaces were not found, then the above incorrect version throws an exception immediately, leaking all of the interface pointers that hadn't yet been processed. We need to move the raw pointers into smart pointers first, so they don't leak, and only then can we start throwing exceptions.

Bonus chatter: Note that this code allows you to call `CreateInstanceMultiQI` and specify that all of the interfaces are optional. That's intentional, because you might have this:

```
auto [widget, doodad] =
    CoCreateInstanceMultiQI<
        std::optional<IWidget>,
        std::optional<IDoodad>
    >(clsid, nullptr, CLSCTX_ANY);
```

In this case, both interfaces are tagged as optional. You don't care whether the object is a widget or doodad, but it needs to be one of the two. (Because `CoCreateInstanceEx` will fail if *none* of the interfaces is supported.)

If you want to create the object, with the possibility that it is *neither* a widget nor a doodad, you can specify `IUnknown` as a required interface. Every COM object supports `IUnknown`, so you know that will succeed if the object can be created at all.

Bonus bonus chatter: I put the required checks inside the `CreateInstanceMultiQI-Worker`, which means that you get a separate version of `CreateInstanceMultiQIWorker` for each arity, as well for each pattern of required/optional. To improve code sharing, we could factor out the part of the worker that is independent of the required/optional pattern, so that function could be shared among all uses with the same number of interfaces.

Raymond Chen

Follow