C++/WinRT event handlers that are lambdas with weak pointers to the parent class, part 2

devblogs.microsoft.com/oldnewthing/20230601-00

June 1, 2023

Raymond Chen

Last time, we discovered a hole in C++/WinRT delegates: They don't support creating a delegate from a weak pointer and a lambda. Let's fix that.

Fortunately, the basic idea is simple. After all, the existing helpers are all lambdas, so we just need another lambda for the missing case. Here's a sketch:

```
template<typename T, typename Handler>
auto weak_delegate(
    winrt::weak_ref<T> weak,
    Handler handler)
{
  return [=](auto&&... args)
    {
      if (auto strong = weak.get()) {
        handler(args...);
      }
    };
}
```

We capture the weak pointer and the handler into the lambda, and when the lambda is invoked, we try to resolve the weak reference to a strong reference. If successful, we call the handler.

We can use this helper like this:

```
// Old and busted
void MyClass::RegisterSomething(int otherData)
{
  widget.Something(
    [weak = get_weak(), this, otherData]
    (auto&& sender, auto&& args)
    {
      if (auto strong = weak.get()) {
        DoThing1(sender);
        DoThing2(args);
        DoThing3(otherData);
      }
    });
}

// New hotness
void MyClass::RegisterSomething(int otherData)
{
  widget.Something(weak_delegate(get_weak(),
    [/* weak = get_weak(), */ this, otherData]
    (auto&& sender, auto&& args)
    {
     // if (auto strong = weak.get()) {
      DoThing1(sender);
      DoThing2(args);
      DoThing3(otherData);
     // }
    }));
}
```

The rest is in the details.

For example, the `handler` is copied twice, once from the caller into the parameter, and then again from the parameter into the lambda. This feels like too much copying, particularly since the lambda may capture move-only objects, so copying isn't an option. Similarly, the weak pointer is also double-copied. We should also perfect-forward our arguments into the lambda to avoid extra copies and preserve reference types. And finally, we should add support for mutable lambdas.

```
template<typename T, typename Handler>
auto weak_delegate(
    winrt::weak_ref<T> weak,
    Handler&& handler)
{
  return [weak = std::move(weak),
          handler = std::forward<Handler>(handler)]
    (auto&&... args) mutable
    {
      if (auto strong = weak.get()) {
        handler(std::forward<decltype(args)>(args)...);
      }
    };
}
```

Furthermore, if the lambda is a coroutine, it will probably need to retain a strong reference to the containing class (as well as copy out anything from its captures that it may need later). Since we have a strong reference in our hands, we may as well let them have that too.

```
template<typename T, typename Handler>
auto weak_delegate(
    winrt::weak_ref<T> weak,
    Handler&& handler)
{
  return [weak = std::move(weak), handler = std::move(handler)]
    (auto&&... args) mutable
    {
      if (auto strong = weak.get()) {
        if constexpr (
            std::is_invocable_v<Handler, decltype(strong)&&, decltype(args)...>) {
          handler(std::move(strong), std::forward<decltype(args)>(args)...);
        } else {
          handler(std::forward<decltype(args)>(args)...);
        }
    };
}

void MyClass::RegisterSomething(int otherData)
{
  widget.Something(weak_delegate(get_weak(),
    [otherData]
    (auto strong, auto sender, auto args)
        -> winrt::fire_and_forget
    {
      auto otherDataLocal = otherData;
      co_await strong->DoThing1Async(sender);
      strong->DoThing2(args);
      strong->DoThing3(otherDataLocal);
    }));
}
```

For notational convenience, the work is probably going to be pulled out into a helper function, to avoid having to say `strong->` all the time and having to put the captured lambda variables into locals:

```
winrt::fire_and_forget MyClass::OnSomething(
    winrt::com_ptr<MyClass> strong,
    Widget sender,
    WidgetChangedEventArgs args,
    int otherData)
{
  // Lifetime is being extended by the "strong" parameter
  co_await DoThing1Async(sender);
  DoThing2(args);
  DoThing3(otherData);
}

void MyClass::RegisterSomething(int otherData)
{
  widget.Something(weak_delegate(get_weak(),
    [otherData]
    (auto&& strong, auto&& sender, auto&& args)
    {
        strong->OnSomething(strong, sender, args, otherData);
    }));
}
```

And now, thinking on it a bit more, maybe the feature of providing the strong reference as a bonus parameter isn't that useful after all. It goes against the convention of starting every coroutine with the creation of a strong reference, which will frustrate static analysis tools as well as human beings doing code reviews.

We'll wrap up our investigations next time, as we try to unify this with C++ `weak_ptr`, so that the helper is useful from plain C++ classes as well as C++/WinRT runtime class implementations.

Bonus chatter: An alternative to using lambdas with captures is to do like `std::thread` and allow bonus parameters to be passed to the lambda-maker, which will in turn be passed to the method. Passing them as explicit parameters also puts them in the coroutine frame, which allows them to be carried across coroutine suspensions.

```
template<typename T, typename Method, typename... Params>
auto weak_method_with_params(
    winrt::weak_ref<T> weak,
    Method method,
    Params&&... param)
{
  return [weak = std::move(weak), method,
          ...params = std::forward<Params>(params)]
    (auto&&... args) mutable
    {
      if (auto strong = weak.get()) {
        (strong.get().*method)(std::forward<decltype(args)>(args)..., params...);
      }
    };
}

winrt::fire_and_forget MyClass::OnSomething(
    winrt::com_ptr<MyClass> strong,
    Widget sender,
    WidgetChangedEventArgs args,
    int otherData)
{
  auto lifetime = get_strong();
  co_await DoThing1Async(sender);
  DoThing2(args);
  DoThing3(otherData);
}

void MyClass::RegisterSomething(int otherData)
{
  widget.Something(weak_method_with_params(get_weak(), &MyClass::OnSomething,
otherData));
}
```

The `[...params = std::forward<Params>(params)]` syntax requires C++20, so if we stick to our C++17 baseline, we'll have to hide it in a tuple, and then things get ugly.

```
template<typename T, typename Method, typename... Params>
auto weak_method_with_params(
    winrt::weak_ref<T> weak,
    Method method,
    Params&&... params)
{
  return [weak = std::move(weak), method,
          params = std::make_tuple(std::forward<Params>(params)...)]
    (auto&&... args)
    {
      if (auto strong = weak.get()) {
        std::apply([&] (auto&&... params) {
          (strong.get().*method)(
            std::forward<decltype(args)>(args)...,
            std::forward<decltype(params)>(params)...);
        }(params);
      }
    };
}
```