C++/WinRT event handlers that are lambdas with weak pointers to the parent class, part 3

devblogs.microsoft.com/oldnewthing/20230602-00

June 2, 2023

Raymond Chen

Last time, we created a helper function for creating a delegate from a weak pointer and a lambda. The weak pointer in question was a C++/WinRT weak pointer, but maybe you want to use this from a traditional C++ class, in which case what you have is not a `winrt::weak_ref` but rather a `std::weak_ptr`. Let's adapt our function to work with `std::weak_ptr` too.

Recall that we left off last time with this function:

```
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)...);
      }
    };
}
```

The only difference that matters (for our purposes) between `winrt::weak_ref` and `std::weak_ptr` is that `winrt::weak_ref` uses the `get()` method to promote the weak reference to a strong reference, whereas the corresponding method on `std::weak_ptr` is called `lock()`.

What we want to do is something weird like

```
template<typename Weak, typename Handler>
auto weak_delegate(
    Weak&& weak,
    Handler&& handler)
{
  return [weak = std::forward<Weak>(weak),
          handler = std::forward<Handler>(handler)]
    (auto&&... args) mutable
    {
      if constexpr (is_specialization_v<Weak, std::weak_ptr>) {
        auto strong = weak.lock();
      } else {
        auto strong = weak.get();
      }
      if (strong) {
        handler(std::forward<decltype(args)>(args)...);
      }
    };
}
```

But that doesn't work because the scope of `strong` is wrong.[1]

What we could do is turn it into a lambda:

```
template<typename Weak, typename Handler>
auto weak_delegate(
    Weak&& weak,
    Handler&& handler)
{
  return [weak = std::forward<Weak>(weak),
          handler = std::forward<Handler>(handler)]
    (auto&&... args) mutable
    {
      auto strong = [&] {
        if constexpr (is_specialization_v<Weak, std::weak_ptr>) {
          return weak.lock();
        } else {
          return weak.get();
        }
      }();
      if (strong) {
        handler(std::forward<decltype(args)>(args)...);
      }
    };
}
```

Now, writing that `is_specialization_v` thingie is going to be a bit of an annoyance, but it turns out we don't need to do it. We can let function overloading do the work.

```
template<typename... Extra>
auto try_resolve_weak_pointer_to_strong(std::weak_ptr<Extra...> const& weak)
{
    return weak.lock();
}

template<typename... Extra>
auto try_resolve_weak_pointer_to_strong(winrt::weak_ref<Extra...> const& weak)
{
    return weak.get();
}

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

The `try_resolve_weak_pointer_to_strong` functions use a trick we looked at a little while ago.

Now you can use `weak_delegate` with C++ standard library weak pointers, as well as C++/WinRT weak references.

```
struct MyClass : std::enable_shared_from_this<MyClass>
{
    ...

    void RegisterSomething(int otherData)
    {
      widget.Something(weak_delegate(weak_from_this(),
        [this, otherData]
        (auto&& sender, auto&& args)
        {
          DoThing1(sender);
          DoThing2(args);
          DoThing3(otherData);
        }));
    }
};
```

A benefit of delegating the “resolve a weak pointer to a strong pointer” decision to a function overload is that you can extend support to other weak pointer libraries by adding new overloads.

```
// WRL WeakRef
auto try_resolve_weak_pointer_to_strong(::Microsoft::WRL::WeakRef const& weak)
{
    ::Microsoft::WRL::ComPtr<IInspectable> o;
    weak.As(&o);
    return o;
}

// wil com_weak_ptr
template<typename... Extra>
auto try_resolve_weak_pointer_to_strong(
    wil::com_ptr<IWeakReference, Extra...> const& weak)
{
    return weak.try_copy<IUnknown>();
}

// Unreal Engine TWeakPtr
template<typename... Extra>
auto try_resolve_weak_pointer_to_strong(TWeakPtr<Extra...> const& weak)
{
    return weak.Pin();
}

// Unreal Engine TWeakObjectPtr
template<typename... Extra>
auto try_resolve_weak_pointer_to_strong(TWeakObjectPtr<Extra...> const& weak)
{
    return weak.Get();
}

// Qt QWeakPointer
template<typename... Extra>
auto try_resolve_weak_pointer_to_strong(QWeakPointer<Extra...> const& weak)
{
    return weak.toStrongRef();
}
```

Note that we are using the `Extra...` technique for accepting any specialization of a C++ template type.[2]

The case of `wil` is a little tricky because `com_weak_ptr` is itself a template alias. We have to use the name of the base template type rather than the type alias.

[1] If you decide to deepen the sad history of Visual Studios custom `__if_exists` keyword, you could write

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

But please, don't do that. Use SFINAE instead.

[2] Others have noted that the `Extra...` trick doesn't work for templates that take non-type template parameters or other templates, so maybe SFINAE is the way to go after all.

```
// winrt weak_ref
template<typename T>
auto try_resolve_weak_pointer_to_strong(T const& weak)
-> decltype(weak.get())
{
    return weak.get();
}

// std weak_ptr
template<typename T>
auto try_resolve_weak_pointer_to_strong(T const& weak)
-> decltype(weak.lock())
{
    return weak.lock();
}

// WRL WeakRef (no change)
auto try_resolve_weak_pointer_to_strong(::Microsoft::WRL::WeakRef const& weak)
{
    ::Microsoft::WRL::ComPtr<IInspectable> o;
    weak.As(&o);
    return o;
}

// wil com_weak_ptr
template<typename T>
auto try_resolve_weak_pointer_to_strong(T const& weak)
-> decltype(weak.try_copy<IUnknown>())
{
    return weak.try_copy<IUnknown>();
}

// Unreal Engine TWeakPtr
template<typename T>
auto try_resolve_weak_pointer_to_strong(T const& weak)
-> decltype(weak.Pin())
{
    return weak.Pin();
}

// Unreal Engine TWeakObjectPtr
// (Would also accidentally match WRL if not for the better overload above)
template<typename T>
auto try_resolve_weak_pointer_to_strong(T const& weak)
-> decltype(weak.Get())
{
    return weak.Get();
}

// Qt QWeakPointer
template<typename T>
auto try_resolve_weak_pointer_to_strong(T const& weak)
```

```
-> decltype(weak.toStrongRef())
{
    return weak.toStrongRef();
}
```

One of the risks of using SFINAE for duck typing is that you may detect things that you didn't mean to. For example, here, we have a name collision: Both WRL WeakRef and Unreal Engine TWeakObjectPtr have a `Get` method, but they do different things. The WRL `Get` method returns the wrapped `IWeakReference*` (no resolving happens), whereas the Unreal Engine `Get` method resolves the weak pointer to a strong pointer. We have to make sure that the WRL specialization gets picked up for WRL WeakRef.