Windows Runtime delegates and object lifetime in C++/WinRT

devblogs.microsoft.com/oldnewthing/20190524-00

May 24, 2019

Raymond Chen

In C++/WinRT, there are four ways to create delegates for event handlers:

- As a raw pointer with a method pointer.
- As a strong pointer with a method pointer.
- As a weak pointer with a method pointer.
- As a lambda.

```
// raw pointer with method pointer.
MyButton.Event({ this, &AwesomePage::Button_Click });

// strong pointer with method pointer.
MyButton.Event({ get_strong(), &AwesomePage::Button_Click });

// weak pointer with method pointer.
MyButton.Event({ get_weak(), &AwesomePage::Button_Click });

// lambda.
MyButton.Event([...](auto&& sender, auto&& args) { ... });
```

The first three are all very similar. They call the method on the object. The only difference is how they obtain the object.

- Raw pointer: The method is invoked on the raw pointer.
- Strong pointer: The method is invoked on the strong pointer.
- Weak pointer: The weak pointer is resolved to a strong pointer. If successful, the method is invoked on the strong pointer. If unsuccessful, nothing happens.[1]

The lambda case is the same as always: The lambda is invoked. It is up to the lambda to determine what objects were captured, how they were captured, and what happens when the lambda is invoked.

The tricky part is deciding which of these mechanisms is most appropriate for your use case.

Use the strong pointer if you need the object referenced by the strong pointer to remain alive for as long as the event handler is registered. This is not a common scenario; usually, you are keeping the object alive by other means.

Before discussing the scenarios where you would use the weak or raw pointer, let’s think about the problems these options are trying to solve.

The underlying problem is the event that is raised after the object has been destructed. Can you guarantee that this will not happen? If you can guarantee it, then you can use a raw pointer. If you cannot guarantee it, then you must use a weak pointer. (If you’re not sure, it is always okay to use a weak pointer.)

Under what conditions can you guarantee that the event won’t be raised after the object has been destructed? Well, one obvious requirement is that you have to stop the event handler from being invoked. The standard way of doing this is by unregistering the event handler. A less common way is destroying the event source. (It’s less common because it assumes that nobody else has taken a reference to the event source and is thereby keeping it alive!)

Preventing the event source from calling your handler is completely on you, so let’s assume you remember to do that, and that you do it correctly.

Even if you remember to unregister the event handler, it’s possible to receive an event after unregistering if the event can be raised from another thread, There’s an unavoidable race condition, because the event may be in flight at the time you unregister the handler. The only way to be sure that you won’t receive any events after unregistering is when the event is always raised from the thread doing the unregistering. This guarantee is available only for objects with thread affinity, and in the cases where the event is raised synchronously in response to the event trigger. In practice, this means that you have this guarantee only for UI objects, such as XAML elements.

Okay, so now that we understand the scenarios, we can write the guidance.

If the event source has thread affinity (typical of UI objects), then you have the option of using either the weak pointer or raw pointer version. The raw pointer is more efficient, but it counts on you having done your analysis correctly. And of course when your object is destroyed, you have to remember to stop the event handler from being invoked.

Use the weak pointer for all the other cases where you are choosing between a weak pointer and a raw pointer, namely if the event source does not have thread affinity, which basically means that the event source is not a UI object.

When an event handler is created via XAML markup, the resulting delegate is created with a raw pointer and method pointer.

```
<!-- XAML -->
<Page x:Name="AwesomePage" ...>
  ...
  <Button Click="Button_Click" >
  ...
</Page>
```

When you write the above XAML, the delegate is created as if you had written

```
thatButton.Click({ this, &AwesomePage::Button_Click });
```

The XAML compiler knows that it's safe to do this because it's hooking up the handler to a XAML element, which is a UI object with thread affinity. It therefore knows that it can unregister its handlers at destruction without risk of events coming in late.

Additional reading:

[1] Note that the "nothing happens" also means that it doesn't even return the magic `RPC_E_DISCONNECTED` error code to tell the event source that the handler is permanently dead.

Raymond Chen

Follow