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

devblogs.microsoft.com/oldnewthing/20190523-00

May 23, 2019

Raymond Chen

In C++/CX, there are two ways to create event handlers: As an (object, method) pair, or as a lambda. And the lifetime rules are different depending on how you do it.

When you create a delegate with an object and a method, the object is captured by weak reference. When the delegate is invoked, the runtime first attempts to resolve the weak reference back to a strong reference. If successful, then it calls the method. If not successful, it raises the `Platform:: DisconnectedException`, which we saw earlier is a signal to the event source that the delegate should be auto-unregistered because it will never succeed again.

When you create a delegate with a lambda, you capture objects into your lambda, and those objects remain alive for as long as the delegate exists. For event handlers, the delegate generally[1] continues to exist until the handler is unregistered. In C++/CX, hat pointers are strong references, so if you capture a hat pointer, you captured a strong reference to the object. In particular, in methods on ref classes, `this` is a hat pointer to the enclosing class, so capturing `this` captures a strong reference to the enclosing class. This is called out in the documentation:

> A named function captures the "this" pointer by weak reference, but a lambda captures it by strong reference and creates a circular reference.

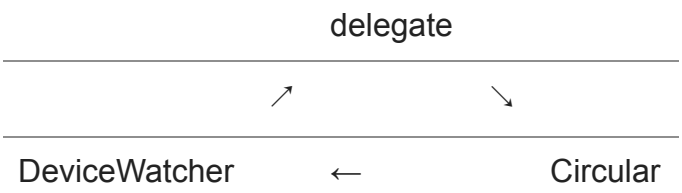
The documentation here is being a bit presumptive that the object captured in the lambda in turn contains a reference to the object that holds the delegate. If that's not the case, then you don't have an (immediate) circular reference.

```
using namespace Windows::Devices::Enumeration;

ref class Circular
{
  DeviceWatcher^ watcher;

public:
  Circular()
  {
    watcher = DeviceInformation::CreateWatcher();
    watcher.Added += ref new TypedEventHandler<
        DeviceWatcher^, DeviceInformation^>(
          [this](DeviceWatcher^ sender, DeviceInformation^ info)
          {
             ...
          });
  }
};
```

The above example creates a C++/CX delegate with a lambda, and that lambda captured `this`. Since this is a ref class, a strong reference was captured into the lambda, and we have created a circular reference:



As with C#, you will have to break this circular reference manually. You can't break the arrow from the delegate to the `Circular` object (since it's captured inside the lambda), so your choices are to unregister the delegate from the event (breaking the arrow from the `DeviceWatcher` to the delegate) or to null out the `Circular` object's reference to the `DeviceWatcher`.

On the other hand, this version creates the delegate with an object and a method pointer:

```
ref class Circular
{
  DeviceWatcher^ watcher;

public:
  Circular()
  {
    watcher = DeviceInformation::CreateWatcher();
    watcher.Added += ref new TypedEventHandler<
        DeviceWatcher^, DeviceInformation^>(
          this, &Circular::OnDeviceAdded);
  }

private:
  void OnDeviceAdded(DeviceWatcher^ sender, DeviceInformation^ info)
  {
     ...
  }
};
```

The object is captured by weak reference into the delegate, which means that there is no circular reference.

Note that capturing the object by weak reference means that the delegate will not keep the object alive. If you want the object to remain alive, you'll have to keep it alive yourself.

A final note is that when an event handler is created via XAML markup, the resulting delegate is of the (object, method) variety.

```
<!-- 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 += ref new EventHandler<RoutedEventArgs^>
    (this, &AwesomePage::Button_Click);
```

So you don't have to worry about circular references created by XAML markup event handlers.

[1] Naturally, you can extend the lifetime of the delegate by keeping an explicit reference to the delegate after you create it. But people rarely do that, and if you do, you know what you signed up for.

Raymond Chen

Follow