A capturing lambda can be a coroutine, but you have to save your captures while you still can

devblogs.microsoft.com/oldnewthing/20211103-00

November 3, 2021

Raymond Chen

We saw some time ago that capturing lambdas which are coroutines result in lifetime issues because the lambda itself returns at the first suspension point, at which point there's a good chance it will be destructed. After that point, any attempt by the lambda body to access those captured variables is a use-after-free bug.

```
winrt::IAsyncAction DoSomethingInBackgroundAsync()
{
    auto a = something();
    auto b = something();
    auto c = something();

    auto callback = [a, b, c]()
        -> winrt::IAsyncAction
        {
            co_await winrt::resume_background();
            DoSomething(a, b, c); // use-after-free bug!
        };
    return callback();
}
```

This problem is so insidious that there's a C++ Core Guideline about it: CP.51: Do not use capturing lambdas that are coroutines.

One workaround is to pass the captures as explicit parameters:

```
winrt::IAsyncAction DoSomethingInBackgroundAsync()
{
    auto a = something();
    auto b = something();
    auto c = something();

    auto callback = [](auto a, auto b, auto c)
        -> winrt::IAsyncAction
        {
            co_await winrt::resume_background();
            DoSomething(a, b, c); // use-after-free bug!
        };
    return callback(a, b, c);
}
```

However, this workaround isn't always available because you may not control the code that invokes the lambda.

```
void RegisterClickHandler(Button const& button, int key)
{
    button.Click([key](auto sender, auto args)
        -> winrt::fire_and_forget
        {
            co_await winrt::resume_background();
            NotifyClick(key);
        });
}
```

You aren't the one who invokes the lambda. That lambda is invoked by the Click event, and it passes two parameters (the sender and the event arugments); there's no way to convince it to pass a `key` too.

One idea would be to extract the work into a nested lambda. We control the invoke of the nested lambda and can pass the extra parameter that way.

```
void RegisterClickHandler(Button const& button, int key)
{
    button.Click([key](auto sender, auto args)
        -> winrt::fire_and_forget
        {
            return [](auto sender, auto args, int key)
            -> winrt::fire_and_forget
            {
                co_await winrt::resume_background();
                NotifyClick(key);
            }(std::move(sender), std::move(args), key);
        });
}
```

The outer lambda is not a coroutine. It's just calling another lambda and propagating the return value.

The inner lambda is a coroutine. To be safe from use-after-free, it is a captureless coroutine, and all of its state is passed as explicit parameters. Here is where we sneak in the extra `key` parameter.

Now, I'm working a bit too hard here, because the coroutine body doesn't use `sender` or `args` so I can accept them by universal reference (to avoid a copy) and just ignore them. To make sure I don't use them by mistake, I'll leave the parameters anonymous.

```
void RegisterClickHandler(Button const& button, int key)
{
    button.Click([key](auto&&, auto&&)
        -> winrt::fire_and_forget
        {
            return [](int key)
            -> winrt::fire_and_forget
            {
                co_await winrt::resume_background();
                NotifyClick(key);
            }(key);
        });
}
```

But what if I told you there was an easier way, where you can have your capturing lambda be a coroutine?

The trick is to make copies of your captures into the coroutine frame before the coroutine reaches its first suspension point. (Note that this trick requires eager-started coroutines. Lazy-started coroutines suspend immediately upon creation, so you have no opportunity to copy the captures into the frame.)

```
void RegisterClickHandler(Button const& button, int key)
{
    button.Click([key](auto&&, auto&&)
        -> winrt::fire_and_forget
        {
            auto copiedKey = key;
            co_await winrt::resume_background();
            NotifyClick(copiedKey);
        });
}
```

We explicitly copy the captured variable into the frame. When execution reaches the first suspension point at the `co_await`, the captured variables disappear. Lesser coroutine lambdas would tremble in fear, but not us! We laugh at the C++ language and say, "Go ahead, take those captured variables away and turn them into poison. It doesn't matter because I made my own copy before you turned them evil."

The tricky part, though, is making sure that we don't touch the original already-freed captures and operate only on our local copies. Somebody coming in later and making a change to the function may not realize that the captures are poisoned and try to use them. Oops. Look who's laughing now.

Next time we'll look at a way to make this slightly less error-prone.

Raymond Chen

Follow