How do I get the effect of C#'s async void in a C++ coroutine? Part 1: Why does the obvious solution crash?

devblogs.microsoft.com/oldnewthing/20190116-00

January 16, 2019

Raymond Chen

The `co_await` C++ language keyword makes it a lot easier to write coroutines. The compiler does the grunt work of transforming your function into a state machine, similar in spirit to the coroutine transformations performed by C# and JavaScript.

C# and JavaScript have an additional feature that C++ doesn't: They let you take a coroutine and treat it like a plain function which returns when the coroutine yields for the first time. With C#, this is done by declaring the function as `async void`. With JavaScript, you get this automatically because JavaScript is dynamically-typed.

But you don't get any built-in help in C++.

```
typedef void (*EventHandler_t)(int a, int b);

void MyEventHandler(int a, int b)
{
  GetReady(a);
  co_await GetSetAsync(b); // oops - this doesn't compile
  Go(a, b);
}

extern void SetEventHandler(EventHandler_t eventHandler);

void RegisterTheEventHandler()
{
   SetEventHandler(MyEventHandler);
}
```

In order to use `co_await`, your function must return a `future` or `task` or some other representation of a coroutine.

Okay, so let's do that.

```
typedef void (*EventHandler_t)(int a, int b);

Concurrency::task<void> MyEventHandler(int a, int b)
{
  GetReady(a);
  co_await GetSetAsync(b); // all right, now this compiles!
  Go(a, b);
}

extern void SetEventHandler(EventHandler_t eventHandler);

void RegisterTheEventHandler()
{
   SetEventHandler(MyEventHandler); // but this doesn't
}
```

So now what?

If you're lucky, your event handler takes a `std::function`, which is more generous about function return types, and the problem goes away.

But suppose you're not that lucky.

Gor Nishanov, one of the brains behind `co_await`, had two suggestions. The first is to create a wrapper.

```
Concurrency::task<void> MyEventHandlerAsync(int a, int b)
{
  GetReady(a);
  co_await GetSetAsync(b);
  Go(a, b);
}

void MyEventHandler(int a, int b)
{
  MyEventHandlerAsync(); // throw away the task
}

extern void SetEventHandler(EventHandler_t eventHandler);

void RegisterTheEventHandler()
{
   SetEventHandler(MyEventHandler);
}
```

The other is to roll up your sleeves and write a specialization of `coroutine_traits` to support returning `void`. I'm not going to reproduce it here because it assumes you understand the low-level machinery that makes `co_await` work,[1] and it ties you to a specific asynchronous framework (since it's going to be used for all functions returning `void`).

I thought I discovered a third way:

```
// Code in italics is wrong.
void MyEventHandler(int a, int b)
{
  [=]() -> Concurrency::task<void>
  {
    GetReady(a);
    co_await GetSetAsync(b);
    Go(a, b);
  }();
}

extern void SetEventHandler(EventHandler_t eventHandler);

void RegisterTheEventHandler()
{
   SetEventHandler(MyEventHandler);
}
```

We create a lambda that returns a coroutine, and that lambda can now use `co_await`. And then we invoke the lambda immediately and throw the result away.

This does cost you a level of indentation, but it beats writing another function entirely.

Basically, I just took Gor's first suggestion and inlined the function as a lambda.

Unfortunately, it also doesn't work.

To see why, let's write out the lambda in its expanded form.

```
void MyEventHandler(int a, int b)
{
  struct Lambda {
   Lambda(int a, int b) : a_(a), b_(b) { }
   Concurrency::task<void> operator() {
    GetReady(a_);
    co_await GetSetAsync(b_);
    Go(a_, b_);
   }
  private:
   int a_, b_;
  } lambda(a, b);

  lambda();
}
```

To make things even more explicit, let's expand out the task, too.

```
void MyEventHandler(int a, int b)
{
  struct LambdaFrame {
   LambdaFrame(Lambda* lambda) : lambda_(lambda) { }

   Concurrency::task<void> Run() {
    if (state_ == 0) {
     GetReady(lambda_->a_);
     auto innerTask = GetSetAsync(lambda_->b_);
     state_ = 1;
     return magic_create_task_from_frame(this);
    } else if (state_ == 1) {
     Go(lambda_->a_, lambda_->b_);
     state_ = 2;
     return magic_completed_task();
    }
   }
  private:
   Lambda* lambda_;
   int state_ = 0;
  };

  struct Lambda {
   Lambda(int a, int b) : a_(a), b_(b) { }
   Concurrency::task<void> operator() {
    return magic_create_task_from_frame
            <LambdaFrame>(__this, a_, b_);
   }
  private:
   int a_, b_;
  } lambda(a, b);

  lambda();
}
```

I'm glossing over a bunch of `magic_` stuff which is part of the behind-the-scenes coroutine infrastructure. The important thing is that the `task` keeps the `LambdaFrame` alive for as long as the task needs it.

But there is no magic that keeps the `lambda` alive!

What we did was invoke the lambda and then destruct it when the function returns. But the task is still running, and that task is going to try to use the `a_` and `b_` members of the lambda. Which no longer exists.

In other words, we have a case of using an object after it has destructed.

But wait, all is not lost. We can fix this by using a non-capturing lambda.

```
void MyEventHandler(int a, int b)
{
  [](int a, int b) -> Concurrency::task<void>
  {
   GetReady(a);
   co_await GetSetAsync(b);
   Go(a, b);
  }(a, b);
}
```

We cannot capture state in the lambda because the lambda is going to be destructed as soon as the task reaches a suspension point. Instead, we capture the state in the explicit lambda parameters (which are part of the frame, not the lambda).

If `MyEventHandler` is a member function, you'll also have to capture `this` explicitly. Furthermore, you need to make sure the instance doesn't get destructed while the task is running. This might be implicit in the way you use the task, or it could be explicit by maintaining a strong reference to the enclosing object.

```
void MyClass::MyEventHandler(int a, int b)
{
  [](std::shared_ptr<MyClass> self, int a, int b)
   -> Concurrency::task<void>
  {
   self->GetReady(a);
   co_await self->GetSetAsync(b);
   self->Go(a, b);
  }(this->shared_from_this(), a, b);
}
```

Unfortunately, having to type `self->` each time you access a member makes this rather cumbersome. To solve this, you can make the lambda body a separate method. But now you've come full circle back to Gor's original suggestion.

So is this hopeless? Not quite. We'll pick up the story next time.

[1] If you want to learn the low-level machinery that makes `co_await` work, you can read this series of articles by Lewis Baker:

Raymond Chen

Follow