Local variables are different from parameters in C++ coroutines

devblogs.microsoft.com/oldnewthing/20211013-00

October 13, 2021

Raymond Chen

In C++, you generally think of parameters and local variables as equivalent. A parameter behaves like a conveniently-initialized local variable.[1]

But not for coroutines.

Let's look at one of the early steps of the coroutine transformation again:

```
return_type MyCoroutine(args...)
{
    create coroutine state
    copy parameters to coroutine frame
    promise_type p;
    return_type task = p.get_return_object();

    try {
        co_await p.initial_suspend();
        coroutine function body
    } catch (...) {
        p.unhandled_exception();
    }
    co_await p.final_suspend();
    destruct promise p
    destruct parameters in coroutine frame
    destroy coroutine state
}
```

Notice that local variables are destructed when we leave the scope of the coroutine function body. In other words, the local variables destruct when we exit the `try` block.

On the other hand, the parameters are not destructed until the coroutine frame is destroyed.

Consider the following code:

```
winrt::IAsyncActionWithProgress<int> LoadAsync(std::shared_ptr<S> s)
{
    /* do stuff */
    co_return;
}

winrt::fire_and_forget Example()
{
    auto temp = std::make_shared<S>();
    auto action = LoadAsync(temp);
    action.Progress(
        [](auto&& sender, auto progress) { /* report progress */ });
    co_await action;
    ProcessData(temp);
    temp = nullptr; // finished with S

    co_await RefreshAsync();
}
```

This code creates an `S` object and loads data into it via a coroutine. That coroutine uses `IAsyncActionWithProgress` as its return type, and the `Example` function takes advantage of that by listening for progress reports. It then `co_await`s the action to wait for the `LoadAsync` coroutine to complete, while getting progress reports along the way.

After the action completes, it processes the data and then nulls out the `temp` local variable to free the `S` object, since it's not needed any more.

Finally, the function performs a refresh so it updates with the new processed data.

Do you see the error in the above analysis?

Since `LoadAsync` receives the `std::shared_ptr<S>` as a parameter, that parameter is stored in the coroutine frame and is not destructed until the frame is destructed, which for `IAsyncAction` doesn't happen until the `IAsyncAction` is destructed.

In the above example, the `IAsyncAction` is stored into a local variable `action`, and that local variable doesn't destruct until the end of the `Example` coroutine. The `S` object is being kept alive by the `s` parameter that was passed to the `LoadAsync` function, and that is kept alive by the coroutine frame, and the coroutine frame is kept alive by the `action` variable.

I discussed this issue from the point of view of the `LoadAsync` function some time ago. But now we're looking at it from the caller's point of view.

As a caller of a coroutine, you should try to destruct the `IAsyncAction` as soon as you're finished with it. Usually, this is done by never assigning to anything; just leave it as a compiler temporary, which destructs at the end of the statement.[2] If you do assign it to a variable, you should null out the variable once you've finished using it:

```
winrt::fire_and_forget Example()
{
    auto temp = std::make_shared<S>();
    auto action = LoadAsync(temp);
    action.Progress(
        [](auto&& sender, auto progress) { /* report progress */ });
    co_await action;
    action = nullptr;
    ProcessData(temp);
    temp = nullptr; // finished with S

    co_await RefreshAsync();
}
```

Or otherwise arrange for the reference to be released, say by scoping it:

```
winrt::fire_and_forget Example()
{
    auto temp = std::make_shared<S>();
    {
        auto action = LoadAsync(temp);
        action.Progress(
            [](auto&& sender, auto progress) { /* report progress */ });
        co_await action;
    } // destruct the action
    ProcessData(temp);
    temp = nullptr; // finished with S

    co_await RefreshAsync();
}
```

The nested scope presents a problem if the `co_await` returns a value, such as an `IAsync-Operation`.

Another solution is to create a helper function to avoid having to store the action in a local variable:

```
template<typename Async, typename Handler>
Async AttachProgress(Async sync, Handler&& handler)
{
    async.Progress(std::forward<Handler>(handler));
    return async;
}
```

Now you don't have to name the object when attaching the progress handler, and the prevents the lifetime from being extended. It also gives you a chance to capture the coroutine result without having to worry about how to get the variable out of a nested scope.

```
winrt::fire_and_forget Example()
{
    auto temp = std::make_shared<S>();
    auto result = co_await AttachProgress(LoadAsync(temp),
        [](auto&& sender, auto progress) { /* report progress */ });
    ProcessData(temp);
    temp = nullptr; // finished with S

    co_await RefreshAsync();
}
```

Another way to solve this problem is on the coroutine side: Move the value out of the parameter to a local variable. That way, it destructs with the locals, rather than hanging around in the parameter space. (Well, technically it's still in the parameter space, but you made it relinquish control of its resources, so what's left is empty.)

```
winrt::IAsyncActionWithProgress<int> LoadAsync(std::shared_ptr<S> s_param)
{
    auto s = std::move(s_param);

    /* do stuff */
    co_return;
}
```

Control of the `S` object has been moved out of the parameter by `std::move`'ing the `shared_ptr` into the local.

If you wrote the coroutine, you can apply this principle, but if the coroutine was provided by something outside your control, then you can't be sure how expensive it is to keep an already-completed coroutine around. Probably best to get rid of it as soon as possible.

[1] One difference is that parameters destruct in the context of the caller: If a parameter's destructor throws an exception, the exception is thrown from the caller and cannot be caught by the called function. Mind you, throwing an exception from a destructor is a bad idea, so this distinction is unlikely to be significant in practice.

[2] Formally, it destructs at the end of the *full expression*, which is smaller than a statement.

Raymond Chen

Follow