Emulating the C# using keyword in C++

devblogs.microsoft.com/oldnewthing/20170703-00

July 3, 2017

Raymond Chen

C# has a very convenient `using` statement which ensures that an object is `Dispose()`d when control exits the block. C++ has a generalization of this concept with RAII types, but things get tricky when you have tasks, lambda capture, and the need to explicitly `Close()` the hat pointer.

Here we go.

Here's one attempt, for C++/CX hat pointers:

```
template<typename T>
class unique_close
{
public:
  unique_close(T^ t) : m_t(t) { }
  ~unique_close() { delete m_t; }

  // Disallow copying
  unique_close(const unique_close& other) = delete;
  unique_close& operator=(const unique_close& other) = delete;

  // Moving transfers the obligation to Close
  unique_close(unique_close&& other)
  {
    *this = std::move(other);
  }
  unique_close& operator=(unique_close&& other)
  {
    using std::swap; // enable ADL on the swap
    swap(m_t, other.m_t);
    return *this;
  }

private:
  T^ m_t;
};

template<typename T>
auto make_unique_close(T^ t)
{
  return unique_close<T>(t);
}
```

With explicit task chaining, you need to remember to use `std::move` to move the `unique_close` object into the final task in the chain if you didn't construct it directly in the capture. If you forget to do this, then the destruction of the `unique_close` objects in the main function will close the objects prematurely because they are still in control.

```
void Scenario1_Render::ViewPage()
{
    ...

    if (outfile)
    {
        auto page = pdfDocument->GetPage(pageIndex);

        return create_task(outfile->OpenTransactedWriteAsync())
            .then([this, page, usingPage{make_unique_close(page)}]
                  (StorageStreamTransaction^ transaction) mutable
        {
            auto options = ref new PdfPageRenderOptions();
            options->DestinationHeight = (unsigned)(page->Size.Height * 2);
            options->DestinationWidth = (unsigned)(page->Size.Width * 2);
            create_task(page->RenderToStreamAsync(transaction->Stream, options))
                .then([this, usingPage{std::move(usingPage)},
                       usingTransaction{make_unique_close(transaction)}]() mutable
            {
                // destruction of usingPage and usingTransaction
                // will close the page and transaction.
            });
        });
    }
    ...
}
```

(Let us ignore the fact that this doesn't work because `task:: then` requires a copyable lambda, and ours is merely movable.)

One way to address the added cognitive burden of having to remember to keep moving the obligation to close is to share that obligation, so that only when the last shared reference is destructed does the object get closed.

```
template<typename T>
class ensure_close
{
public:
  ensure_close(T^ t) : m_t(t) { }
  ~ensure_close() { delete m_t; }

  // Disallow copying and moving
  ensure_close(const ensure_close& other) = delete;
  ensure_close& operator=(const ensure_close& other) = delete;
  ensure_close(const ensure_close&& other) = delete;
  ensure_close& operator=(const ensure_close&& other) = delete;

private:
  T^ m_t;
};

template<typename T>
using shared_close = std::shared_ptr<ensure_close<T>>;

template<typename T>
auto make_shared_close(T^ t)
{
  return std::make_shared<ensure_close<T>>(t);
}
```

Now you can copy the `shared_close` around, and only when the last copy is destructed does the wrapped hat pointer get closed.[1]

```
void Scenario1_Render::ViewPage()
{
    ...

    if (outfile)
    {
        auto page = pdfDocument->GetPage(pageIndex);
        auto usingPage = make_shared_close(page);

        return create_task(outfile->OpenTransactedWriteAsync())
            .then([this, page, usingPage]
                  (StorageStreamTransaction^ transaction)
        {
            auto usingTransaction = make_shared_close(transaction);
            auto options = ref new PdfPageRenderOptions();
            options->DestinationHeight = (unsigned)(page->Size.Height * 2);
            options->DestinationWidth = (unsigned)(page->Size.Width * 2);
            return create_task(page->RenderToStreamAsync(transaction->Stream,
options))
                .then([this, usingPage, usingTransaction]()
            {
                // destruction of the last shared usingPage
                // and usingTransaction will close the page and transaction.
            });
        });
    }
    ...
}
```

Still, you have to remember to keep passing the `usingPage` and `usingTransaction` around. If you forget, then the object gets closed prematurely. (And if the `shared_close` is created by one lambda and consumed by a sibling lambda, well you now have a `shared_ptr` to a `shared_close`, which is getting ridiculous.)

But wait, you can stop the madness.

Let's go back to `unique_close`: This class becomes much more convenient if you use the `co_await` keyword, because the compiler will do the heavy lifting of cleaning up when the last task has completed.[2]

```
task<void> Scenario1_Render::ViewPageAsync()
{
    ...

    if (outfile)
    {
        auto page = pdfDocument->GetPage(pageIndex);
        auto usingPage = make_unique_close(page);

        auto transaction =
            co_await outfile->OpenTransactedWriteAsync();
        auto usingTransaction = make_unique_close(transaction);
        auto options = ref new PdfPageRenderOptions();
        options->DestinationHeight = (unsigned)(page->Size.Height * 2);
        options->DestinationWidth = (unsigned)(page->Size.Width * 2);
        co_await page->RenderToStreamAsync(transaction->Stream, options);
        // destruction of usingPage and usingTransaction
        // will close the page and transaction.
    }
    ...
}
```

We have offloaded all the thinking to the compiler. The compiler will do the work of making sure that the `unique_close` objects are destructed when control leaves the block. The `unique_close` objects will remain alive during the `co_await` statements, which is what we want.

We could make our `unique_close` a little fancier by making it a little more `unique_ptr` y.

```
template<typename T>
class unique_close
{
public:
  unique_close(T^ t) : m_t(t) { }
  ~unique_close() { delete m_t; }

  T^ get() { return m_t; }
  T^ operator*() { return m_t; }
  T^ operator->() { return m_t; }
  ...
};
```

This leaves us with

```
task<void> Scenario1_Render::ViewPageAsync()
{
    ...

    if (outfile)
    {
        auto page = make_unique_close(
             pdfDocument->GetPage(pageIndex));
        auto transaction = make_unique_close(
            co_await outfile->OpenTransactedWriteAsync());
        auto options = ref new PdfPageRenderOptions();
        options->DestinationHeight = (unsigned)(page->Size.Height * 2);
        options->DestinationWidth = (unsigned)(page->Size.Width * 2);
        co_await page->RenderToStreamAsync(transaction->Stream, options);
        // destruction of page and transaction
        // will close the page and transaction.
    }
    ...
}
```

That doesn't seem so bad. Pretty close to C# but still in the spirit of C++.[3]

[1] You (and by "you" I mean "me") would be sorely tempted to write this with a custom deleter instead.

```
namespace Details
{
  template<typename T>
  struct close_deleter
  {
    void operator()(T^ t) { delete t; }
  };
}

template<typename T>
auto make_shared_close(T^ t)
{
    return std::shared_ptr<T>(t, Details::close_deleter<T>());
}
```

Except that this doesn't work because `std::shared_ptr` manages raw pointers, not hat pointers.

[2] If you are a total crazy person, you might consider adding a boolean conversion operator to the `unique_close`:

```
...
   operator bool() const { return true; }
...
```

This appears to serve no purpose, but it lets you write this:

```
// Oh my goodness what kind of craziness is about to happen?
#define scope_using__(t, c) \
    if (auto _scope_using_##c##_ = make_unique_close(t))
#define scope_using_(t, c) scope_using__(t, c)
#define scope_using(t) scope_using(t, __COUNTER__)

task<void> Scenario1_Render::ViewPageAsync()
{
    ...

    if (outfile)
    {
        auto page = pdfDocument->GetPage(pageIndex);
        scope_using (page)
        {
            auto transaction =
                co_await outfile->OpenTransactedWriteAsync();
            scope_using (transaction)
            {
                auto options = ref new PdfPageRenderOptions();
                options->DestinationHeight = (unsigned)(page->Size.Height * 2);
                        options->DestinationWidth = (unsigned)(page->Size.Width * 2);
                co_await page->RenderToStreamAsync(transaction->Stream, options);
                // exiting the scope_using will close
                // the corresponding objcts.
            }
        }
    }
    ...
}
```

This is a moral outrage. Let us never speak of this again.

[3] If you wanted to be cute, you could rename `make_unique_close` to `Using`.

```
auto page = Using(pdfDocument->GetPage(pageIndex));
auto transaction = Using(co_await outfile->OpenTransactedWriteAsync());
```

Raymond Chen

Follow