A pattern for obtaining a single value while holding a lock

devblogs.microsoft.com/oldnewthing/20250127-00

January 27, 2025

It is often the case that you have a mutex or other lockable object which protects access to a complex variable, and you want to read the variable's value (possibly modifying it in the process) while holding the lock, but then operate on the value outside the lock.

The traditional way is to do something like this:

```
// Assume we have these member variables
std::mutex m_mutex;
Widget m_widget;

// Get a copy of the widget
Widget widget;
{
    auto guard = std::lock_guard(m_mutex);
    widget = m_widget;
}
⟦ use the variable "widget" ⟧
```

This does suffer from the problem of running the `Widget` constructor for an object that we're going to overwrite anyway. The compiler will also have to deal with the possibility that the `lock_guard` constructor throws an exception, forcing the destruction of a freshly-constructed `Widget`.

I like to use this alternate pattern, using an immediately-invoked lambda that returns a value.

```
// Get a copy of the widget
Widget widget = [&] {
    auto guard = std::lock_guard(m_mutex);
    return m_widget;
}();
⟦ use the variable "widget" ⟧
```

Thanks to the magic of copy elision (in this specific form known as Return Value Optimization, or RVO), we never have to construct a dummy Widget. Instead, the `widget` variable directly receives a copy of `m_widget` under the lock, but the value survives the lock.

This also can be used to move a value out of a lock-protected object, so that the value can destruct outside the lock.

```
// Move the widget into our variable
Widget widget = [&] {
    auto guard = std::lock_guard(m_mutex);
    return std::move(m_widget);
}();
// Allow the widget to destruct outside the lock
```

Or to exchange the value in the object while under the lock and operate on the old value outside the lock.

```
Widget widget = [&] {
    auto guard = std::lock_guard(m_mutex);
    return std::exchange(m_widget, {});
}();
// Allow the widget to destruct outside the lock
```

If you do this more than a few times, you may want to write a helper.

```
Widget CopySavedWidget()
{
    auto guard = std::lock_guard(m_mutex);
    return m_widget;
}

template<typename T>
Widget ExchangeSavedWidget(T&& value)
{
    auto guard = std::lock_guard(m_mutex);
    return std::exchange(m_widget, std::forward<T>(value)):
}
```