How do I put a non-copyable, non-movable, non-constructible object into a std::optional?

devblogs.microsoft.com/oldnewthing/20241115-00

November 15, 2024

Last time, we wondered how you could put an object into a `std::optional<T>` if `T` is not copyable, not movable, and not constructible. You typically run into this case when `T` is an object that comes from a factory method rather than a constructor.

For expository purposes, say we have a `Widget` that can be created inside a region or outside a region. One possible design would be to have a constructor that takes a `bool` that say whether to go inside or outside.

```
struct Widget
{
    Widget(Region const& region, bool inside);

    Widget(Widget const&) = delete;
    Widget(Widget &amp;&) = delete;
    Widget& operator=(Widget const&) = delete;
    Widget& operator=(Widget &amp;&) = delete;
};
```

I'm not a fan of this design because you have to remember what that final `bool` means.

```
Widget widget(region, true); // what does "true" mean?
```

Does that `true` mean "initially enabled"? Does it mean "inside"? Does it mean "outside"?

To remove ambiguity, we can switch to factory methods.

```
struct Widget
{
    Widget() = delete;
    Widget(Widget const&) = delete;
    Widget(Widget &amp;&) = delete;
    Widget& operator=(Widget const&) = delete;
    Widget& operator=(Widget &amp;&) = delete;

    static Widget CreateInside(Region const& region);
    static Widget CreateOutside(Region const& region);
private:
    [[ ... ]]
};
```

Okay, so how can we put a `Widget` inside a `std::optional<Widget>`? All of our tools for putting an object into an `optional` are failing us. We can't use `emplace`: That will try to construct the `Widget` from the thing we passed to emplace, but `Widget` is not constructible!

The trick is that the `std::optional` constructor and assignment operator create the `T` as if by non-list-initialization. This means that implicit conversion operators are in play!

```
struct WidgetInsideRegionCreator
{
    WidgetCreator(Region const& region) : m_region(region) {}
    operator Widget() { return Widget::CreateInside(m_region); }
    Region const& m_region;
};

void sample(Region const& region)
{
    // construct with a Widget value
    std::optional<Widget> o(WidgetInsideRegionCreator(region));

    // or place a Widget into the optional
    o.emplace(WidgetInsideRegionCreator(region));
}
```

The idea here is that we create a helper object, the `WidgetInsideRegionCreator`, which supports an implicit conversion to `Widget` via the factory method. The `Widget` can then be initialized from the helper object by conversion. The return value from the conversion operator is placed directly in the `optional`'s `Widget` thanks to mandatory copy elision.

Okay, now that we know what to do, we can generalize it, so you don't have to create dozens of tiny little creator classes.

```
template<typename F>
struct EmplaceHelper
{
    EmplaceHelper(F&& f) : m_f(f) {}
    operator auto() { return m_f(); }
    F& m_f;
};

void sample(Region const& region)
{
    // construct with a Widget value
    std::optional<Widget> o(
        EmplaceHelper([&] {
            return Widget::CreateInside(region);
        }));

    // or place a Widget into the optional
    o.emplace(EmplaceHelper([&] {
            return Widget::CreateInside(region);
        }));
}
```

This trick works even if the factory method belongs to another object.

```
struct WidgetFactory
{
    Widget CreateInside(Region const& region) const;
};

void sample(WidgetFactory const& factory,
            Region const& region)
{
    // construct with a Widget value
    std::optional<Widget> o(
        EmplaceHelper([&] {
            return factory.CreateInside(region);
        }));

    // or place a Widget into the optional
    o.emplace(EmplaceHelper([&] {
            return factory.CreateInside(region);
        }));
}
```

If you don’t like lambdas, you can try invoke-oriented programming.

```
template<typename F, typename... Args>
struct EmplaceHelper
{
    EmplaceHelper(F&& f, Args&&... args)
        : m_f(f), m_args((Args&&)args...) {}
    operator auto()
        { return std::apply(m_f, m_args); }
    F& m_f;
    std::tuple<Args&&...> m_args;
};

template<typename F, typename... Args>
EmplaceHelper(F&&, Args&&...) -> EmplaceHelper<F, Args...>;

void sample(WidgetFactory const& factory,
            Region const& region)
{
    // construct with a Widget value
    std::optional o(
        EmplaceHelper(Widget::CreateInside, region));

    // or place a Widget into the optional
    o.emplace(EmplaceHelper(&WidgetFactory::CreateInside,
                            factory, region));

    // lambdas still work
    o.emplace(EmplaceHelper([&] {
            return factory.CreateInside(region);
        }));
}
```