Creating a generic insertion iterator, part 2

devblogs.microsoft.com/oldnewthing/20250131-00

January 31, 2025

Last time, we tried to create a generic insertion iterator but ran into trouble because our iterator failed to satisfy the iterator requirements of default constructibility and assignability.

We ran into this problem because we stored the lambda as a member of the iterator.

So let's not do that!

Instead of saving the lambda, we'll just save a pointer to the lambda.

```
template<typename Lambda>
struct generic_output_iterator
{
    using iterator_category = std::output_iterator_tag;
    using value_type = void;
    using pointer = void;
    using reference = void;
    using difference_type = void;

    generic_output_iterator(Lambda&& lambda) noexcept :
        insert(std::addressof(lambda)) {}

    generic_output_iterator& operator*() noexcept
        { return *this; }
    generic_output_iterator& operator++() noexcept
        { return *this; }
    generic_output_iterator& operator++(int) noexcept
        { return *this; }

    template<typename Value>
    generic_output_iterator& operator=(
        Value&& value)
    {
        (*insert)(std::forward<Value>(value));
        return *this;
    }

protected:
    Lambda* insert;

};

template<typename Lambda>
generic_output_iterator<Lambda>
generic_output_inserter(Lambda&& lambda) noexcept {
    return generic_output_iterator<Lambda>(
        std::forward<Lambda>(lambda));
}

template<typename Lambda>
generic_output_iterator(Lambda&&) ->
    generic_output_iterator<Lambda>;
```

This requires that the lambda remain valid for the lifetime of the iterator, but that may not a significant burden. Other iterators also retain references that are expected to remain valid for the lifetime of the iterator. For example, `std::back_inserter(v)` requires that `v` remain valid for as long as you use the inserter. And if you use the iterator immediately, then the requirement will be satisfied:

```
auto sample(std::vector<int>& v)
{
    std::map<int> m;
    std::copy(v.begin(), v.end(),
        generic_output_iterator(
            [&m, hint = m.begin()](int v) mutable {
            hint = m.insert(hint, { v, 0 });
        }));
}
```

This lambda is used to produce the `generic_output_iterator`, and the resulting iterator is consumed by `std::copy` before the lambda destructs at the end of the full statement.

It does become a problem if you want to save the iterator:

```
auto sample(std::vector<int>& v1, std::vector<int>& v2)
{
    std::map<int> m;
    // Don't do this
    auto output =
        generic_output_iterator(
            [&m, hint = m.begin()](int v) mutable {
            hint = m.insert(hint, { v, 0 });
        }));
    std::copy(v1.begin(), v1.begin(), output);
    std::copy(v2.begin(), v2.begin(), output);
}
```

In the above example, the resulting iterator is saved in `output`, and then the lambda destructs, leaving `output` pointing to an already-destroyed lambda.

If you need to do this, you should store the lambda in a variable whose lifetime is at least as long as the iterator.

```
auto sample(std::vector<int>& v1, std::vector<int>& v2)
{
    std::map<int> m;
    auto lambda = [&m, hint = m.begin()](int v) mutable {
        hint = m.insert(hint, { v, 0 });
    };

    auto output = generic_output_iterator(lambda);
    std::copy(v1.begin(), v1.begin(), output);
    std::copy(v2.begin(), v2.begin(), output);
}
```

Bonus chatter: If we really wanted to, we could teach the `generic_output_iterator` to make a copy of the lambda, though we would have to work around the inability to default-construct a lambda, and also deal with the possibility that the lambda is move-only.

We can simulate copy-assigning a lambda by destructing the old lambda and then copy-constructing the incoming lambda into the space occupied by the old lambda. If the lambda is noexcept copy-constructible, then we can just construct the new lambda in the space occupied by the old lambda. But if the copy constructor is potentially-throwing, we cannot contain the lambda directly but instead have to use a `unique_ptr` to the lambda that we swap in after successfully copying the incoming one.

If the lambda itself is not even copyable (for example, if it captures a `unique_ptr`), we'll have to emplace it into a `shared_ptr`.

Doing all of this is a lot of annoying typing and SFINAE, so I'll leave it as an exercise that nobody will do.