Creating a generic insertion iterator, part 1

devblogs.microsoft.com/oldnewthing/20250130-00

January 30, 2025

Last time, we created an inserter iterator that does unhinted insertion. We noticed that most of the iterator is just boilerplate, so let's generalize it into a version that is all-boilerplate.

```
// Do not use: See discussion
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) :
        insert(std::forward<Lambda>(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:
    std::decay_t<Lambda> insert;

};

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

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

For convenience, I provided both a deduction guide and a maker function, so you can use whichever version appeals to you.[1]

The generic output iterator uses the lambda to process each insertion. You can make the lambda do anything you like. For example:

```
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 });
        }));
}
```

In this example, we take a vector of what we expect to be a mostly-sorted sequence of integers and use them as keys in a newly-created map, where the associated integer is initialized to zero. We take advantage of the mostly-sorted-ness by using the location of the previously-inserted item as the hint for the next item.

Too bad this is not a valid iterator!

Among the requirements for iterators is that they be default-constructible and assignable, and ours is neither because capturing lambdas are neither default-constructible nor assignable. (Capturing lambdas can be copy-constructible and move-constructible, but they are never default-constructible or assignable.)

We'll try to fix this problem next time.

[1] The C++ standard library has a lot of maker functions because they predate class template argument deduction (CTAD) and deduction guides.