How do I create an inserter iterator that does unhinted insertion into an associative container like std::map?

devblogs.microsoft.com/oldnewthing/20250129-00

January 29, 2025

The C++ standard library contains various types of inserters:

- `back_inserter(c)` which uses `c.push_back(v)`.
- `front_inserter(c)` which uses `c.push_front(v)`.
- `inserter(c, it)` which uses `c.insert(it, v)`.

C++ standard library associative containers do not have `push_back` or `push_front` methods; your only option is to use the `inserter`. But we also learned that the hinted insertion can speed up the operation if the hint is correct, or slow it down if the hint is wrong. (Or it might not have any effect at all.)

What if you know that the items are arriving in an unpredictable order? You don't want to provide a hint, because that's a pessimization. The `inserter` requires you to pass a hint. What do you do if you don't want to provide a hint?

It looks like you'll have to write your own inserter. Fortunately, it's not hard.

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

    default_insert_iterator(Container& c) noexcept :
        container(std::addressof(c)) {}

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

    default_insert_iterator& operator=(
        typename Container::value_type const& value)
    {
        container->insert(value);
        return *this;
    }

    default_insert_iterator& operator=(
        typename Container::value_type && value)
    {
        container->insert(std::move(value));
        return *this;
    }

protected:
    Container* container;

};

template<typename Container>
default_insert_iterator<Container>
default_inserter(Container& cont) noexcept {
    return default_insert_iterator<Container>(cont);
}
```

The iterator type itself doesn't have much in it. It declares some member types to satisfy iterator requirements, and it has dummy `*` and `++` operators. The actual interesting thing is the assignment operator, which inserts the value into the container using an unhinted `insert`.

There is a subtlety: Iterators must be default-constructible, so we have to store the container as a pointer rather than a reference, since there is no default constructor for a reference. (Although iterators must be default-constructible, a default-constructed iterator is not dereferenceable, so it's okay that the pointer is uninitialized.)

But wait, there are other ways to insert into a container. Maybe we know that the items are mostly-ordered, so you want to hint the insertion with the result of the previous insertion.

We'll generalize our inserter next time.