On creating (and using) a transforming iterator

devblogs.microsoft.com/oldnewthing/20230523-00

May 23, 2023

Raymond Chen

The C++20 ranges library come with the ability to transform a view; that is, to provide a unary function that is applied to each element of a view, producing a new view.

```
std::array<int, 2> a = { 99, 42 };

// This range reports values that are one larger than the values
// in an array. The array values are unchanged.
auto r = a |
    std::ranges::views::transform([](int v) { return v + 1; });

// This creates a vector from the range
std::vector v(r.begin(), r.end());

// The resulting vector is { 100, 43 }
```

But what if your code base isn't ready to move to C++20 yet? For example, you might be a library that wants to support C++17 or even C++14.

You can take the original iterator and wrap it inside another iterator that overrides the `*` operator so that it applies a transformation to the value before returning it. Note that our transforming iterator cannot support the `->` operator since there is no value object we can return a pointer to; our value is generated on demand. Fortunately, the standard permits us to omit `->` support, provided we define `pointer` as `void`.

```
template<typename Inner>
struct Wrap
{
    Wrap(Inner const& inner) :
        m_inner(inner) {}
    Inner m_inner;
};

template<typename It, typename Transformer>
class transform_iterator : Wrap<Transformer>
{
    It m_it;
public:
    transform_iterator(It const& it,
        Transformer const& transformer) :
        m_it(it),
        Wrap<Transformer>(transformer) {}

    // copy constructors and assignment operators defaulted

    using difference_type = typename
        std::iterator_traits<It>::difference_type;
    using value_type = typename std::invoke_result<
        Transformer, It>::type;
    using pointer = void;
    using reference = void;
    using iterator_category = std::input_iterator_tag;

    bool operator==(transform_iterator const& other)
    { return m_it == other.m_it; }
    bool operator!=(transform_iterator const& other)
    { return m_it != other.m_it; }

    auto operator*() const { return (*this)(*m_it); }

    auto operator++() { ++m_it; return *this; }
    auto operator++(int)
    { auto prev = *this; ++m_it; return prev; }
};

// For C++14 (no CTAD)
template<typename It, typename Transformer>
auto make_transform_iterator(
    It const& it, Transformer const& transformer)
{
    return transform_iterator<It, Transformer>
        (it, transformer);
}
```

We can use this transforming iterator like this:

```
std::array<int, 2> a = { 99, 42 };

auto transformer = [](int v) { return v + 1; };

// This creates a vector from the array, applying
// a transformation to each value
std::vector v(
    transform_iterator(a.begin(), transformer),
    transform_iterator(a.end(), transformer));

// If C++14 (no CTAD)
std::vector<int> v(
    transform_iterator(a.begin(), transformer),
    transform_iterator(a.end(), transformer));

// The resulting vector is { 100, 43 }
```

The transformation can even change the type:

```
std::array<int, 2> a = { 99, 42 };

auto transformer = [](int v)
    { return std::make_pair(v + 1, v); };

// This creates a map from the array
std::map m(
    transform_iterator(a.begin(), transformer),
    transform_iterator(a.end(), transformer));

// If C++14 (no CTAD)
std::map<int, int> m(
    transform_iterator(a.begin(), transformer),
    transform_iterator(a.end(), transformer));

// The resulting map is
//  m[100] = 99
//  m[43] = 42
```

There are some non-obvious pieces of the above `transform_iterator`.

We want to accept not just lambdas as transformers, but any Callable. That means that we use `std::invoke` to invoke the transformer on the wrapped iterator. This allows transformers to be function pointers, member function pointers, pointers to member variables, lambdas, or any other class with a public `operator()`. For example:

```
struct S
{
    int value;
    int ValuePlusOne() { return value + 1; };
};

int ValueMinusOne(S const& s)
{
    return s.value - 1;
}

void example()
{
    std::array<S, 2> a { 99, 42 };

    // v1 = { 99, 42 }; - pointer to data member
    std::vector<int> v1(
    transform_iterator(a.begin(), &S::value),
    transform_iterator(a.end(), &S::value));

    // v2 = { 100, 43 }; - pointer to member function
    std::vector<int> v2(
    transform_iterator(a.begin(), &S::ValuePlusOne),
    transform_iterator(a.end(), &S::ValuePlusOne));

    // v3 = { 98, 41 }; - pointer to free function
    std::vector<int> v3(
    transform_iterator(a.begin(), &ValueMinusOne),
    transform_iterator(a.end(), &ValueMinusOne));
}
```

The `transform_iterator` derives from a wrapped `Transformer`. This is a space optimization that takes advantage of empty base optimization (EBO): If the `Transformer` is an empty class, then the `Wrapped<Transformer>` will also be an empty class, and empty base classes are permitted to occupy zero bytes.[1] (Normally, objects cannot be of size zero.) This means that a `transform_iterator` that has an empty class as a transformer (such as a captureless lambda) is the same size as the original iterator.

We wrap the `Transformer` inside a class because base classes must be classes, but the `Transformer` might not be a class, as we noted above.

A transforming iterator is handy for populating a `std::map` because all three of the major implementations of the C++ standard library optimize the two-iterator `insert()` overload for the case where the items are inserted in increasing key order at the end of map.[2]

In the case where you have two versions of a function, one of which takes a range and another of which takes items one at a time, you can avoid the need for a transforming iterator by turning the problem around: Instead of producing a range of transformed iterators to pass to function, you produce an output iterator that calls the single-parameter version of the function.

```
std::vector<int> v;
std::transform(a.begin(), a.end(), std::back_inserter(v),
               transformer);

std::map<int, int> m;
std::transform(a.begin(), a.end(), std::inserter(m, m.end()),
               transformer);
```

This is simpler but has its downsides:

- You lose CTAD, since the compiler cannot infer the template type parameters from the constructor.
- Repeated single-element function calls may be less efficient than a bulk operation.

For example, inserting a transformed range into a vector is linear in the number of elements inserted plus the number of elements after the insertion point. In other words, inserting n new elements in front of k existing elements is $O(n + k)$ if you do it in a single call to `insert`:

```
// Bulk insert after the first element
// This takes O(a.size() + v.size() - 1) =
// O(a.size() + v.size())
v.insert(v.begin() + 1,
    transform_iterator(a.begin(), transformer),
    transform_iterator(a.end(), transformer));
```

If you insert one element at a time, then you pay the k each time, which results in a running time of $n\,O(1 + k) = O(n + nk)$, an extra cost of $O(nk)$.

```
// One-at-a-time insert after the first element
// This takes O(a.size() + a.size() * (v.size() - 1)) =
// O(a.size() * v.size())
v.insert(v.begin() + 1,
    transform_iterator(a.begin(), transformer),
    transform_iterator(a.end(), transformer));
```

Bonus chatter: The Boost library comes with an implementation of `transform_iterator`, so you can use that one instead of the custom one here. But at least you got to see a number of techniques that are commonly seen in library code.

[1] There are other things that could prevent the empty base optimization, but they do not apply here.

[2] In other words, it's as if the ranged insertion method were written as

```
template&typename Iterator>
void insert(Iterator first, Iterator last)
{
    for (; first != last; ++first) {
        insert(end(), *first);
    }
}
```

Last time, we looked at other ways of doing efficient bulk insertions.