Reordering C++ template type parameters for usability purposes, and type deduction from the future

devblogs.microsoft.com/oldnewthing/20230609-00

June 9, 2023

Raymond Chen

Suppose you want to write a function that takes any iterable and converts it to a `std::vector`. This is sort of like the C# analogue to the `.ToList()` LINQ method. Here's a starter kit:

```
template<typename Container>
auto to_vector(Container&& c)
{
    using ElementType = std::decay_t<decltype(*c.begin())>;
    std::vector<ElementType> v;
    std::copy(c.begin(), c.end(), std::back_inserter(v));
    return v;
}
```

This deduces the underlying value type of the vector from the original container, and it uses the default allocator. The `std::decay_t` does multiple things for us here: It removes the references from the dereferenced iterator, and it also removes the `const` and `volatile` attributes. That second step allows a container of `const T` to convert to a vector of `T`.

But maybe you want to let the caller override the underlying value type. For example, given a `std::list<int>`, you want to produce a `std::vector<long>` by saying

```
std::list<int> l = /* some expression */;
auto v = to_vector<long>(l);
```

Okay, so you add an `ElementType` type parameter to the template function, and have it default to the container's underlying type.

```
template<typename Container,
    typename ElementType =
        std::decay_t<decltype(*std::declval<Container>().begin())>>
auto to_vector(Container&& c)
{
    std::vector<ElementType> v;
    std::copy(c.begin(), c.end(), std::back_inserter(v));
    return v;
}
```

This works great, but using it is an annoyance because in order to customize the `ElementType`, you have to restate the `Container` first.

```
std::list<int> l = /* some expression */;
auto v = to_vector<std::list<int>&, long>(l);
```

You'd much rather say

```
std::list<int> l = /* some expression */;
auto v = to_vector<long>(l);
```

This reads much nicer because it looks like you're saying "I'm converting to `vector<long>`." So you swap the order of the type parameters:

```
template<
    typename ElementType =
        std::decay_t<decltype(*std::declval<Container>().begin())>,
    typename Container>
auto to_vector(Container&& c)
{
    std::vector<ElementType> v;
    std::copy(c.begin(), c.end(), std::back_inserter(v));
    return v;
}
```

Unfortunately, this doesn't compile because you are trying to default `ElementType` based on a `Container` that hasn't been declared yet.

So how can you get a template type parameter to default to a value that is dependent upon a future type parameter?

You use a trick.

```
template<
    typename ElementType = void,
    typename Container>
auto to_vector(Container&& c)
{
    using ActualElementType = std::conditional_t<
        std::is_same_v<ElementType, void>,
        std::decay_t<decltype(*c.begin())>,
        ElementType>;
    std::vector<ActualElementType> v;
    std::copy(c.begin(), c.end(), std::back_inserter(v));
    return v;
}
```

The formal type parameter `ElementType` defaults to `void`, which is a sentinel value that means “Substitute the underlying value type of the collection here.” Inside the function body, after we have successfully deduced the `Container`, we calculate the `ActualElementType` by using the explicitly-provided `ElementType` or (if it was defaulted to `void`) using the underlying type of the `Container`.

Adding support for custom allocators is more complicated because the allocator is a parameter. We have to convert the `void` to an actual allocator by the time we get to the parameter list.

```
template<
    typename ElementType = void,
    typename Allocator = void,
    typename Container>
auto to_vector(Container&& c,
    std::conditional_t<
        std::is_same_v<Allocator, void>,
        std::allocator<
            std::conditional_t<
            std::is_same_v<ElementType, void>,
            std::decay_t<decltype(*std::declval<Container>().begin())>,
            ElementType>>,
        Allocator> al = {})
{
    using ActualElementType = std::conditional_t<
        std::is_same_v<ElementType, void>,
        std::decay_t<decltype(*std::declval<Container>().begin())>,
        ElementType>;
    using ActualAllocator = std::conditional_t<
        std::is_same_v<Allocator, void>,
        std::allocator<ActualElementType>,
        Allocator>;
    std::vector<ActualElementType, ActualAllocator> v(al);
    std::copy(c.begin(), c.end(), std::back_inserter(v));
    return v;
}
```

There's a lot of repetition here: We want to use `ActualAllocator` defined in the function body, but we can't use it in the function prototype, so we have to inline its definition, producing a big ugly mess.

One way to solve this is to "save" it in another defaulted template type parameter.

```
template<
    typename ElementType = void,
    typename Allocator = void,
    typename Container,
    typename ActualElementType = std::conditional_t<
        std::is_same_v<ElementType, void>,
        std::decay_t<decltype(*std::declval<Container>().begin())>,
        ElementType>;
    typename ActualAllocator = std::conditional_t<
        std::is_same_v<Allocator, void>,
        std::allocator<ActualElementType>,
        Allocator>>
auto to_vector(Container&& c,
    ActualAllocator al = ActualAllocator())
{
    std::vector<ActualElementType, ActualAllocator> v(al);
    std::copy(c.begin(), c.end(), std::back_inserter(v));
    return v;
}
```

Another solution is to create helper types to do the calculations.

```
template<typename ElementType, typename Container>
using ActualElementType = std::conditional_t<
        std::is_same_v<ElementType, void>,
        std::decay_t<decltype(*std::declval<Container>().begin())>,
        ElementType>;

template<typename ActualElementType,
    typename ActualAllocator = std::conditional_t<
        std::is_same_v<Allocator, void>,
        std::allocator<ActualElementType<ElementType, Container>>,
        Allocator>;

template<
    typename ElementType = void,
    typename Allocator = void,
    typename Container>
auto to_vector(Container&& c,
    ActualAllocator<ElementType, Allocator, Container> al = {})
{
    std::vector<ActualElementType<ElementType, Container>,
        ActualAllocator<ElementType, Allocator, Container>> v(al);
    std::copy(c.begin(), c.end(), std::back_inserter(v));
    return v;
}
```

Bonus chatter: There's still more work to be done with `to_vector`, since it doesn't work with `vector<bool>`, thanks to `vector<bool>`'s wacko proxy object, and it doesn't work with C-style arrays since they don't have a `begin()` method.

Instead, we can use `std::iterator_traits` to tell us what the iterator produces. and we can use `std::begin` to support C-style arrays.

```
template<typename ElementType, typename Container>
using ActualElementType = std::conditional_t<
        std::is_same_v<ElementType, void>,
        std::decay_t<
            std::iterator_traits<
                    decltype(
                        std::begin(std::declval<Container>()))
                >::value_type>,
        ElementType>;

template<typename ActualElementType,
    typename ActualAllocator = std::conditional_t<
        std::is_same_v<Allocator, void>,
        std::allocator<ActualElementType<ElementType, Container>>,
        Allocator>;

template<
    typename ElementType = void,
    typename Allocator = void,
    typename Container>
auto to_vector(Container&& c,
    ActualAllocator<ElementType, Allocator, Container> al = {})
{
    std::vector<ActualElementType<ElementType, Container>,
        ActualAllocator<ElementType, Allocator, Container>> v(al);
    std::copy(c.begin(), c.end(), std::back_inserter(v));
    return v;
}
```