Little gotcha with C++/WinRT iterators: The case of the mutated temporary

devblogs.microsoft.com/oldnewthing/20201002-00

October 2, 2020

Raymond Chen

C++/WinRT iterators resemble iterators in many ways, most notable they resemble them enough to let you use them in ranged for loops or in standard algorithms that use input iterators.

But they aren't full iterators, so watch out.

Most notably, the iterators for indexed collections (known internally as `fast_iterator`) support random access, such as `it += 2` to step forward two items, or `it[-1]` to read the previous item. But they are not full-fledged random access iterators.

They aren't proper random access iterators because the return type of the dereferencing `operator*` is not a reference. It's a value.

That means that you can't do this:

```
auto it = collection.First();
*it = replacement_value; // replace the first element
```

If the underlying type of the collection is not a class type, then the compiler will complain:

```
IVector<int> collection;
auto start = collection.First();
*start = 42; // C2106: '=' left operand must be an l-value
```

But if the underlying type is of class type, then the class's assignment operator will be used. You're assigning to a temporary object, which is legal, though not particularly useful.

```
IVector<Point> collection;
auto start = collection.First();
*start = Point{ 1.0f, 2.0f }; // doesn't do what you think
start[0].X = 1.0f; // doesn't do what you think

IVector<Class> collection;
auto start = collection.First();
*start = Class(); // doesn't do what you think
```

In both cases, what you're doing is assigning a new object to (or in the case of `.X`, mutating) the temporary object returned by `operator*`, and then throwing the temporary away. The original collection remains unchanged.

For reference types like the imaginary Windows Runtime class `Class`, this is largely not a problem, because the C++/WinRT projection of Windows Runtime classes is as a reference-counted object (similar to `shared_ptr`), so you can invoke methods, and those methods will affect the underlying shared object.

But for value types like `Point`, this is a hidden gotcha.

Bonus chatter: The dereferencing `operator*` could have returned a proxy object which supported conversion to `T` (which performs a `GetAt`) or assignment from `T` (which performs a `SetAt`), but that would result in a different kind of confusion, because

```
auto v = *it;
```

wouldn't actually produce the value. It would merely produce a proxy. And that would be really weird:

```
void Something(T value);

IIterable<T> it = ...;
auto v = *it;
*it = new_value;
Something(v); // passes new_value!
```

The problem is that `auto v = *it` deduces that `v` is a proxy object. It is only when the proxy is converted to a `T` that the `GetAt` is performed. In the above code fragment, that happens at the call to `Something`, and that means that when `GetAt` is called, it will fetch the `new_value` that was assigned in the meantime.

To avoid surprises like this, the C++/WinRT iterators are strictly input iterators.

Raymond Chen

Follow