An iterable of iterables: C# collections support covariance, but C++ collections do not

devblogs.microsoft.com/oldnewthing/20201008-00

October 8, 2020

Raymond Chen

Collections in the C# language support covariance. Collections in the C++ language do not.

This means, for example, that if a function wants a collection of `T`, C# lets you pass a collection of things that derive from `T`, but C++ requires you to pass a collection of exactly `T`.

This can be confusing for methods that accept collections of collections. C++ will allow the outer collection to decay, but the objects inside the collection cannot. The Windows Runtime by convention accepts collections of `T` in the form of an `IIterable<T>` (which projects into C# as of an `IEnumerable<T>`), so that you can pass anything that can produce a sequence of `T` objects. It could be a `std::vector<T>` or a `std::array<T>` or even a SQL or LINQ query that produces a sequence of `T` objects.

But things get a little weird if you have a collection of collections, because C++ collections don't support covariance.

Before we look at covariance, let's look at simple conversion. Suppose you have to pass an `IIterable<Point>` to a method like `InkManager.SelectWithPolyLine` which takes a parameter of type `IIterable<Point>`.

```
// C#
var points = new List<Point> {
        UpperLeftCorner, UpperRightCorner, LowerRightCorner, LowerLeftCorner
    };
inkManager.SelectWithPolyline(points);

// C++/CX
auto points = ref new Vector<Point>({
      UpperLeftCorner, UpperRightCorner, LowerRightCorner, LowerLeftCorner,
});
inkManager->SelectWithPolyline(points);

// C++/WinRT
auto points = single_threaded_vector<Point>({
      UpperLeftCorner, UpperRightCorner, LowerRightCorner, LowerLeftCorner,
  });
inkManager.SelectWithPolyline(points);
```

This works because `List<T>` is convertible to `IEnumerable<T>`, and `Vector<T>` is convertible to `IIterable<T>`. The conversion works one level deep.

Now suppose you are implementing the `UIElement. FindSubElementsForTouch-Targeting`:

```
// C#
virtual override IEnumerable<IEnumerable<Point>> FindSubElementsForTouchTargeting();

// C++/CX
virtual IIterable<IIterable<Point>^>^ FindSubElementsForTouchTargeting() override;

// C++/WinRT
IIterable<IIterable<Point>> FindSubElementsForTouchTargeting();
```

In C#, this is straightforward. You can have a list of lists:

```
virtual override IEnumerable<IEnumerable<Point>> FindSubElementsForTouchTargeting()
{
  var polygon1 = new List<Point> {
        UpperLeftCorner1, UpperRightCorner1, LowerRightCorner1, LowerLeftCorner1,
    };
  var polygon2 = new List<Point> {
        UpperLeftCorner2, UpperRightCorner2, LowerRightCorner2, LowerLeftCorner2,
    };
  var results = new List<List<Point>> { polygon1, polygon2 };
  return results;
}
```

Thanks to covariance, a `List<List<Point>>` is compatible with `IEnumerable<IEnumerable<Point>>` because `List<T>` is compatible with `IEnumeraable<T>`, and `IEnumeraable<T>` is covariant in `T`.

C++ is not as lucky. The analogous code in C++/CX would be something like this:

```
virtual override IEnumerable<IEnumerable<Point>> FindSubElementsForTouchTargeting()
{
  auto polygon1 = ref new Vector<Point>({
        UpperLeftCorner1, UpperRightCorner1, LowerRightCorner1, LowerLeftCorner1,
    });
  auto polygon2 = ref new Vector<Point>({
        UpperLeftCorner2, UpperRightCorner2, LowerRightCorner2, LowerLeftCorner2,
    });
  auto results = ref new Vector<Vector<Point>>({ polygon1, polygon2 });
  return results;
}
```

And in C++/WinRT:

```
IIterable<IIterable<Point>> FindSubElementsForTouchTargeting()
{
  auto polygon1 = single_threaded_vector<Point>({
        UpperLeftCorner1, UpperRightCorner1, LowerRightCorner1, LowerLeftCorner1,
    });
  auto polygon2 = single_threaded_vector<Point>({
        UpperLeftCorner2, UpperRightCorner2, LowerRightCorner2, LowerLeftCorner2,
    });
  auto results = single_threaded_vector<IVector<Point>>({ polygon1, polygon2 });
  return results;
}
```

This code doesn't work because there is no conversion from `Vector<Vector<Point>^>^` to `IIterable<IIterable<Point>^>^` (C++/CX) or from `IVector<IVector<Point>>` to `IIterable<IIterable<Point>>` (C++/WinRT). The automatic conversion gets you part way there, but it can't convert the inner portion due to the lack of covariance.

C++/CX			**C++/WinRT**		
`Vector<`	`Vector<`	`Point>^>^`	`IVector<`	`IVector`	`<Point>>`
↓			↓		
`IIterable<`	`Vector<`	`Point>^>^`	`IIterable<`	`IVector`	`<Point>>`
	↓			↓	
`IIterable<`	`IIterable<`	`Point>^>^`	`IIterable<`	`IIterable<`	`<Point>>`

You have to declare the inner portion exactly correctly the first time. The language isn't going to help you.

```
// C++/CX
virtual override IEnumerable<IEnumerable<Point>> FindSubElementsForTouchTargeting()
{
  auto polygon1 = ref new Vector<Point>({
        UpperLeftCorner1, UpperRightCorner1, LowerRightCorner1, LowerLeftCorner1,
    });
  auto polygon2 = ref new Vector<Point>({
        UpperLeftCorner2, UpperRightCorner2, LowerRightCorner2, LowerLeftCorner2,
    });
  auto results = ref new Vector<IEnumerable<Point>>({ polygon1, polygon2 });
  return results;
}
```

And in C++/WinRT:

```
IIterable<IIterable<Point>> FindSubElementsForTouchTargeting()
{
  auto polygon1 = single_threaded_vector<Point>({
        UpperLeftCorner1, UpperRightCorner1, LowerRightCorner1, LowerLeftCorner1,
    });
  auto polygon2 = single_threaded_vector<Point>({
        UpperLeftCorner2, UpperRightCorner2, LowerRightCorner2, LowerLeftCorner2,
    });
  auto results = single_threaded_vector<IIterable<Point>>({ polygon1, polygon2 });
  return results;
}
```

You do lose the ability to index the `results` and access the original vectors, since by the time you put them into the `results`, they have been turned into `IIterable`s.

C++/WinRT gives you a little help here. If the iterable of iterables is a parameter to a function, then you can let C++/WinRT auto-generate the outer iterable, at which point the automatic conversions will kick in for the inner objects because the type is being explicitly specified by the parameter.

```
extern void SomeMethod(winrt::param::iterable<
    IIterable<Point>> const& elementOutlines);

auto polygon1 = single_threaded_vector<Point>({
      UpperLeftCorner1, UpperRightCorner1, LowerRightCorner1, LowerLeftCorner1,
  });
auto polygon2 = single_threaded_vector<Point>({
      UpperLeftCorner2, UpperRightCorner2, LowerRightCorner2, LowerLeftCorner2,
  });
SomeMethod({ polygon1, polygon2 });
```

That wasn’t too messy.

Raymond Chen

Follow