C-Sharp Mapping for Sequences

Ice for .NET supports several different mappings for sequences. By default, sequences are mapped to arrays. You can use metadata directives to map sequences to a number of alternative types:

- `System.Collections.Generic.List`
- `System.Collections.Generic.LinkedList`
- `System.Collections.Generic.Queue`
- `System.Collections.Generic.Stack`
- Types derived from `Ice.CollectionBase`, which is a drop-in replacement for `System.Collections.CollectionBase` (this mapping is provided mainly for compatibility with Ice versions prior to 3.3)
- User-defined custom types that derive from `System.Collections.Generic.IEnumerable<T>`.

The different mappings allow you to map sequences to a container type that provides the correct performance trade-off for your application.

On this page:

Array Mapping for Sequences in C#

By default, the Slice-to-C# compiler maps sequences to arrays. Interestingly, no code is generated in this case; you simply define an array of elements to model the Slice sequence. For example:

Slice

```
sequence<Fruit> FruitPlatter;
```

Given this definition, to create a sequence containing an apple and an orange, you could write:

C#

```
Fruit[] fp = { Fruit.Apple, Fruit.Orange };
```

Or, alternatively:

C#

```
Fruit fp[] = new Fruit[2];
fp[0] = Fruit.Apple;
fp[1] = Fruit.Orange;
```

The array mapping for sequences is both simple and efficient, especially for sequences that do not need to provide insertion or deletion other than at the end of the sequence.

Mapping to Predefined Generic Containers for Sequences in C#

With metadata directives, you can change the default mapping for sequences to use generic containers provided by .NET. For example:

Slice

```
["clr:generic:List"] sequence<string> StringSeq;
["clr:generic:LinkedList"] sequence<Fruit> FruitSeq;
["clr:generic:Queue"] sequence<int> IntQueue;
["clr:generic:Stack"] sequence<double> DoubleStack;
```

The "`clr:generic:`*type*" metadata directive causes the `slice2cs` compiler to the map the corresponding sequence to one of the containers in the `System.Collections.Generic` namespace. For example, the `Queue` sequence maps to `System.Collections.Generic.Queue<int>` due to its metadata directive.

The predefined containers allow you to select an appropriate space-performance trade-off, depending on how your application uses a sequence. In addition, if a sequence contains value types, such as `int`, the generic containers do not incur the cost of boxing and unboxing and so are quite efficient. (For example, `System.Collections.Generic.List<int>` performs within a few percentage points of an integer array for insertion and deletion at the end of the sequence, but has the advantage of providing a richer set of operations.)

Generic containers can be used for sequences of any element type except objects. For sequences of objects, only `List` is supported because it provides the functionality required for efficient unmarshaling. Metadata that specifies any other generic type is ignored with a warning:

Slice

```
class MyClass {
    // ...
};

["clr:generic:List"]
sequence<MyClass> MyClassList; // OK

["clr:generic:LinkedList"]
sequence<MyClass> MyClassLinkedList; // Ignored
```

In this example, sequence type `MyClassList` maps to the generic container `System.Collections.Generic.List<MyClass>`, but sequence type `MyClassLinkedList` uses the default array mapping.

Mapping to Custom Types for Sequences in C#

If the array mapping and the predefined containers are unsuitable for your application (for example, because you may need a priority queue, which does not come with .NET), you can implement your own custom containers and direct `slice2cs` to map sequences to these custom containers. For example:

Slice

```
["clr:generic:MyTypes.PriorityQueue"] sequence<int> Queue;
```

This metadata directive causes the Slice `Queue` sequence to be mapped to the type `MyTypes.PriorityQueue`. You must specify the fully-qualified name of your custom type following the `clr:generic:` prefix. This is because the generated code prepends a `global::` qualifier to the type name you provide; for the preceding example, the generated code refers to your custom type as `global::MyTypes.PriorityQueue<int>`.

Your custom type can have whatever interface you deem appropriate, but it must meet the following requirements:

- The custom type must derive from `System.Collections.Generic.IEnumerable<T>`.
- The custom type must provide a readable `Count` property that returns the number of elements in the collection.
- The custom type must provide an `Add` method that appends an element to the end of the collection.
- If (and only if) the Slice sequence contains elements that are Slice classes, the custom type must provide an indexer that sets the value of an element at a specific index. (Indexes, as usual, start at zero.)

As an example, here is a minimal class (omitting implementation) that meets these criteria:

C#

```
public class PriorityQueue<T> : IEnumerable<T>
{
    public IEnumerator<T> GetEnumerator();

    public int Count
        get;

    public void Add(T elmt);

    public T this[int index] // Needed for class elements only.
        set;

    // Other methods and data members here...
}
```

`CollectionBase` Mapping for Sequences in C#

The `CollectionBase` mapping is provided mainly for compatibility with Ice versions prior to 3.3. Internally, `CollectionBase` is implemented using `System.Collections.Generic.List<T>`, so it offers the same performance trade-offs as `List<T>`. (For value types, `Ice.CollectionBase` is considerably faster than `System.Collections.CollectionBase`, however.)

`Ice.CollectionBase` is not as type-safe as `List<T>` because, in order to remain source-code compatible with `System.Collections.CollectionBase`, it provides methods that accept elements of type `object`. This means that, if you pass an element of the wrong type, the problem will be diagnosed only at run time, instead of at compile time. For this reason, we suggest that you do not use the `CollectionBase` mapping for new code.

To enable the `CollectionBase` mapping, you must use the `"clr:collection"` metadata directive:

Slice

```
["clr:collection"] sequence<Fruit> FruitPlatter;
```

With this directive, `slice2cs` generates a type that derives from `Ice.CollectionBase`:

C#

```
public class FruitPlatter : Ice.CollectionBase<M.Fruit>, System.ICloneable
{
    public FruitPlatter();
    public FruitPlatter(int capacity);
    public FruitPlatter(Fruit[] a);
    public FruitPlatter(System.Collections.Generic.IEnumerable<Fruit> l);

    public static implicit operator _System.Collections.Generic.List<Fruit>(FruitPlatter s);

    public virtual FruitPlatter GetRange(int index, int count);

    public static FruitPlatter Repeat(Fruit value, int count);

    public object Clone();
}
```

The generated `FruitPlatter` class provides the following methods:

- `FruitPlatter();`
  `FruitPlatter(int capacity);`
  `FruitPlatter(Fruit[] a);`

`FruitPlatter(IEnumerable<Fruit> l);`
Apart from calling the default constructor, you can also specify an initial capacity for the sequence or, using the array constructor, initialize a sequence from an array. In addition, you can initialize the class to contain the same elements as any enumerable collection with the same element type.

- `FruitPlatter GetRange(int index, int count);`
  This method returns a new sequence with `count` elements that are copied from the source sequence beginning at `index`.

- `FruitPlatter Repeat(Fruit value, int count);`
  This method returns a sequence with `count` elements that are initialized to `value`.

- `object Clone()`
  The `Clone` method returns a shallow copy of the source sequence.

- `static implicit operator List<Fruit> (FruitPlatter s);`
  This operator performs an implicit conversion of a `FruitPlatter` instance to a `List<Fruit>`, so you can pass a `FruitPlatter` sequence where a `List<Fruit>`, `IEnumerable<Fruit>`, or `System.Collections.IEnumerable` is expected.

The remaining methods are provided by the generic `Ice.CollectionBase` base class. This class provides the following methods:

- `CollectionBase();`
  `CollectionBase(int capacity);`
  `CollectionBase(T[] a);`
  `CollectionBase(IEnumerable<T> l);`
  The constructors initialize the sequence as for the concrete derived class.

- `int Count { get; }`
  This property returns the number of elements of the sequence.

- `int Capacity { get; set; }`
  This property controls the capacity of the sequence. Its semantics are as for the corresponding property of `List<T>`.

- `virtual void TrimToSize();`
  This method sets the capacity of the sequence to the actual number of elements.

- `int Add(object o);`
  `int Add(T value);`
  These methods append `value` at the end of the sequence. They return the index at which the element is inserted (which always is the value of `Count` prior the call to `Add`.)

- `void Insert(int index, object o);`
  `void Insert(int index, T value);`
  These methods insert an element at the specified index.

- `virtual void InsertRange(int index, CollectionBase<T> c);`
  `virtual void InsertRange(int index, T[] c);`
  These methods insert a range of values into the sequence starting at the given index.

- `virtual void SetRange(int index, CollectionBase<T> c);`
  `virtual void SetRange(int index, T[] c);`
  These methods copy the provided sequence over a range of elements in the target sequence, starting at the provided index, with semantics as for `System.Collections.ArrayList`.

- `void RemoveAt(int index);`
  This method deletes the element at the specified index.

- `void Remove(object o);`
  `void Remove(T value);`
  These methods search for the specified element and, if present, delete that element. If the element is not in the sequence, the methods do nothing.

- `virtual void RemoveRange(int index, int count);`
  This method removes `count` elements, starting at the given index.

- `void Clear();`
  This method deletes all elements of the sequence.

- `bool Contains(object o);`
  `bool Contains(T value);`
  These methods return true if the sequence contains `value`; otherwise, they return false.

- `int IndexOf(object o);`
  `int IndexOf(T value);`
  These methods return the index of the specified element. If the element is not in the sequence, the return value is `-1`.

- `virtual int LastIndexOf(T value);`
  `virtual int LastIndexOf(T value, int startIndex);`
  `virtual int LastIndexOf(T value, int startIndex, int count);`
  These methods search for the provided element and return its last occurrence in the sequence, as for `System.Collections.ArrayList.LastIndexOf`.

- `object this[int index] { get; set; }`
  `T this[int index] { get; set; }`
  The indexers allow you to read and write elements using array subscript notation.

- `IEnumerator<T> GetEnumerator();`
  This method returns an enumerator that you can use to iterate over the collection.

- `static implicit operator List<T> (CollectionBase<T> s);`
  As for the derived class, this operator permits implicit conversion to a `List<T>`.

- `void CopyTo(T[] a);`
  `void CopyTo(T[] a, int i);`
  `void CopyTo(int i, T[] a, int ai, int c);`
  `void CopyTo(System.Array a, int i);`
  These methods copy the contents of a sequence into an array. The semantics are the same as for the corresponding methods of `List<T>`.

- `T[] ToArray();`
  The `ToArray` method returns the contents of the sequence as an array.

- `void AddRange(CollectionBase<T> s);`
  `void AddRange(T[] a);`
  The `AddRange` methods append the contents of a sequence or an array to the current sequence, respectively.

- `virtual void Sort();`
  `virtual void Sort(System.Collections.IComparer comparer);`
  `virtual void Sort(int index, int count, System.Collections.IComparer comparer);`
  These methods sort the sequence.

- `virtual void Reverse();`
  `virtual void Reverse(int index, int count);`
  These methods reverse the order of elements of the sequence.

- `virtual int BinarySearch(T value);`
  `virtual int BinarySearch(T value, System.Collections.IComparer comparer);`
  `virtual int BinarySearch(int index, int count, T value, System.Collections.IComparer comparer);`
  The methods perform a binary search on the sequence, with semantics as for `System.Collections.ArrayList`.

- `static FruitPlatter Repeat(Fruit value, int count);`
  This method returns a sequence with `count` elements that are initialized to `value`.

Note that for all methods that return sequences, these methods perform a shallow copy, that is, if you have a sequence whose elements have reference type, what is copied are the references, not the objects denoted by those references.

`Ice.CollectionBase` also provides the usual `GetHashCode` and `Equals` methods, as well as the comparison operators for equality and inequality. (Two sequences are equal if they have the same number of elements and all elements in corresponding positions are equal, as determined by the `Equals` method of the elements.)

`Ice.CollectionBase` also implements the inherited `IsFixedSize`, `IsReadOnly`, and `IsSynchronized` properties (which return false), and the inherited `SyncRoot` property (which returns `this`).

Creating a sequence containing an apple and an orange is simply a matter of writing:

C#

```
FruitPlatter fp = new FruitPlatter();
fp.Add(Fruit.Apple);
fp.Add(Fruit.Orange);
```

Multi-Dimensional Sequences in C#

Slice permits you to define sequences of sequences, for example:

Slice

```
enum Fruit { Apple, Orange, Pear };
["clr:generic:List"] sequence<Fruit> FruitPlatter;
["clr:generic:LinkedList"] sequence<FruitPlatter> Cornucopia;
```

If we use these definitions as shown, the type of FruitPlatter in the generated code is:

C#

```
System.Collections.Generic.LinkedList<System.Collections.Generic.List<Fruit>>
```

Here the outer sequence contains elements of type `List<Fruit>`, as you would expect.

Now let us modify the definition to change the mapping of `FruitPlatter` to an array:

Slice

```
enum Fruit { Apple, Orange, Pear };
sequence<Fruit> FruitPlatter;
["clr:LinkedList"] sequence<FruitPlatter> Cornucopia;
```

With this definition, the type of `Cornucopia` becomes:

C#

```
System.Collections.Generic.LinkedList<Fruit[]>
```

The generated code now no longer mentions the type `FruitPlatter` anywhere and deals with the outer sequence elements as an array of `Fruit` instead.

See Also

- Metadata
- C-Sharp Mapping for Identifiers
- C-Sharp Mapping for Modules
- C-Sharp Mapping for Built-In Types
- C-Sharp Mapping for Enumerations
- C-Sharp Mapping for Structures
- C-Sharp Mapping for Dictionaries
- C-Sharp Collection Comparison
- C-Sharp Mapping for Constants
- C-Sharp Mapping for Exceptions