C++ Mapping for Sequences

On this page:

- Default Sequence Mapping in C++
- Custom Sequence Mapping in C++
  - STL Container Mapping for Sequences
  - Array Mapping for Sequences in C++
  - Range Mapping for Sequences in C++

Default Sequence Mapping in C++

Here is the definition of our `FruitPlatter` sequence once more:

Slice

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

The Slice compiler generates the following definition for the `FruitPlatter` sequence:

C++

```
typedef std::vector<Fruit> FruitPlatter;
```

As you can see, the sequence simply maps to an STL vector, so you can use the sequence like any other STL vector. For example:

C++

```
// Make a small platter with one Apple and one Orange
//
FruitPlatter p;
p.push_back(Apple);
p.push_back(Orange);
```

As you would expect, you can use all the usual STL iterators and algorithms with this vector.

Custom Sequence Mapping in C++

In addition to the default mapping of sequences to vectors, Ice supports three additional custom mappings for sequences.

STL Container Mapping for Sequences

You can override the default mapping of Slice sequences to C++ vectors with a metadata directive, for example:

Slice

```
[["cpp:include:list"]]

module Food {

    enum Fruit { Apple, Pear, Orange };

    ["cpp:type:std::list< ::Food::Fruit>"]
    sequence<Fruit> FruitPlatter;

};
```

With this metadata directive, the sequence now maps to a C++ `std::list`:

C++

```
#include <list>

namespace Food {

    typedef std::list< Food::Fruit> FruitPlatter;

    // ...
}
```

The `cpp:type` metadata directive must be applied to a sequence definition; anything following the `cpp:type:` prefix is taken to be the name of the type. For example, we could use `["cpp:type:::std::list< ::Food::Fruit>"]`. In that case, the compiler would use a fully-qualified name to define the type:

C++

```
typedef ::std::list< ::Food::Fruit> FruitPlatter;
```

Note that the code generator inserts whatever string you specify following the `cpp:type:` prefix literally into the generated code. This means that, to avoid C++ compilation failures due to unknown symbols, you should use a qualified name for the type.

Also note that, to avoid compilation errors in the generated code, you must instruct the compiler to generate an appropriate include directive with the `cpp:include` global metadata directive. This causes the compiler to add the line

C++

```
#include <list>
```

to the generated header file.

Instead of `std::list`, you can specify a type of your own as the sequence type, for example:

Slice

```
[["cpp:include:FruitBowl.h"]]

module Food {

    enum Fruit { Apple, Pear, Orange };

    ["cpp:type:FruitBowl"]
    sequence<Fruit> FruitPlatter;

};
```

With these metadata directives, the compiler will use a C++ type `FruitBowl` as the sequence type, and add an include directive for the header file `FruitBowl.h` to the generated code.

You can use any class of your choice as a sequence type, but the class must meet certain requirements. (`vector`, `list`, and `deque` happen to meet these requirements.)

- The class must have a default constructor and a single-argument constructor that takes the size of the sequence as an argument of unsigned integral type.
- The class must have a copy constructor.
- The class must provide a member function `size` that returns the number elements in the sequence as an unsigned integral type.
- The class must provide a member function `swap` that swaps the contents of the sequence with another sequence of the same type.
- The class must define `iterator` and `const_iterator` types and must provide `begin` and `end` member functions with the usual semantics; the iterators must be comparable for equality and inequality.

Less formally, this means that if the class looks like a `vector`, `list`, or `deque` with respect to these points, you can use it as a custom sequence implementation.

In addition to modifying the type of a sequence itself, you can also modify the mapping for particular return values or parameters. For example:

Slice

```
[["cpp:include:list"]]
[["cpp:include:deque"]]

module Food {

    enum Fruit { Apple, Pear, Orange };

    sequence<Fruit> FruitPlatter;

    interface Market {
        ["cpp:type:list< ::Food::Fruit>"]
        FruitPlatter barter(["cpp:type:deque< ::Food::Fruit>"] FruitPlatter offer);
    };

};
```

With this definition, the default mapping of `FruitPlatter` to a C++ `vector` still applies but the return value of `barter` is mapped as a `list`, and the `offer` parameter is mapped as a `deque`.

Array Mapping for Sequences in C++

The array mapping for sequences applies to input parameters and to out parameters of AMI and AMD operations. For example:

Slice

```
interface File {
    void write(["cpp:array"] Ice::ByteSeq contents);
};
```

The `cpp:array` metadata directive instructs the compiler to map the `contents` parameter to a pair of pointers. With this directive, the `write` method on the proxy has the following signature:

C++

```
void write(const std::pair<const Ice::Byte*, const Ice::Byte*>& contents);
```

To pass a byte sequence to the server, you pass a pair of pointers; the first pointer points at the beginning of the sequence, and the second pointer points one element past the end of the sequence.

Similarly, for the server side, the `write` method on the skeleton has the following signature:

C++

```
virtual void write(const ::std::pair<const ::Ice::Byte*, const ::Ice::Byte*>&,
                   const ::Ice::Current& = ::Ice::Current()) = 0;
```

The passed pointers denote the beginning and end of the sequence as a range `[first, last)` (that is, they use the usual STL semantics for iterators).

The array mapping is useful to achieve zero-copy passing of sequences. The pointers point directly into the server-side transport buffer; this allows the server-side run time to avoid creating a `vector` to pass to the operation implementation, thereby avoiding both allocating memory for the sequence and copying its contents into that memory.

You can use the array mapping for any sequence type. However, it provides a performance advantage only for byte sequences (on all platforms) and for sequences of integral or floating point types (x86 platforms only).

The called operation in the server must not store a pointer into the passed sequence because the transport buffer into which the pointer points is deallocated as soon as the operation completes.

Range Mapping for Sequences in C++

The range mapping for sequences is similar to the array mapping and exists for the same purpose, namely, to enable zero-copy of sequence parameters:

Slice

```
interface File {
    void write(["cpp:range"] Ice::ByteSeq contents);
};
```

The `cpp:range` metadata directive instructs the compiler to map the `contents` parameter to a pair of `const_iterator`. With this directive, the `write` method on the proxy has the following signature:

C++

```
void write(const std::pair<Ice::ByteSeq::const_iterator, Ice::ByteSeq::const_iterator>& contents);
```

Similarly, for the server side, the `write` method on the skeleton has the following signature:

C++

```
virtual void write(
    const ::std::pair<::Ice::ByteSeq::const_iterator, ::Ice::ByteSeq::const_iterator>&,
    const ::Ice::Current& = ::Ice::Current()) = 0;
```

The passed iterators denote the beginning and end of the sequence as a range `[first, last)` (that is, they use the usual STL semantics for iterators).

The motivation for the range mapping is the same as for the array mapping: the passed iterators point directly into the server-side transport buffer and so avoid the need to create a temporary `vector` to pass to the operation.

As for the array mapping, the range mapping can be used with any sequence type, but offers a performance advantage only for byte sequences (on all platforms) and for sequences of integral type (x86 platforms only).

The operation must not store an iterator into the passed sequence because the transport buffer into which the iterator points is deallocated as soon as the operation completes.

You can optionally add a type name to the `cpp:range` metadata directive, for example:

Slice

```
interface File {
    void write(["cpp:range:std::deque<Ice::Byte>"] Ice::ByteSeq contents);
};
```

This instructs the compiler to generate a pair of `const_iterator` for the specified type:

C++

```
virtual void write(
    const ::std::pair<std::deque<Ice::Byte>::const_iterator,
                      std::deque<Ice::Byte>::const_iterator>&,
    const ::Ice::Current& = ::Ice::Current()) = 0;
```

This is useful if you want to combine the range mapping with a custom sequence type that behaves like an STL container.

See Also

- Sequences
- Asynchronous Method Dispatch (AMD) in C++
- C++ Mapping for Enumerations
- C++ Mapping for Structures
- C++ Mapping for Dictionaries
- C++ Mapping for Operations