C++ Mapping for Sequences

On this page:

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 a standard `std::vector`, so you can use the sequence like any other vector. For example:

C++

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

Custom Sequence Mapping in C++

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

Custom Type 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 can be applied to a sequence or dictionary 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.

The class or template class you provide must meet the following requirements:

- The class must have a default constructor.
- The class must have a copy constructor.

If you use a class that also meets the following requirements

- The class has a single-argument constructor that takes the size of the sequence as an argument of unsigned integral type.
- The class has a member function `size` that returns the number elements in the sequence as an unsigned integral type.
- The class provides a member function `swap` that swaps the contents of the sequence with another sequence of the same type.

- The class defines `iterator` and `const_iterator` types and provides `begin` and `end` member functions with the usual semantics; its iterators are comparable for equality and inequality.

then you do not need to provide code to marshal and unmarshal your custom sequence–Ice will do it automatically.

Less formally, this means that if the provided class looks like a `vector`, `list`, or `deque` with respect to these points, you can use it as a custom sequence implementation without any additional coding.

You can also use a custom class that does not meet these requirements if you tell Ice how to handle this class. First, you need to define a `StreamHelperCategory` and `StreamableTraits` specialization or partial specialization for each category of type that shares the same marshaling and unmarshaling code. For example:

```
namespace Ice
{

//
// A new category for all Google Protocol Buffers, to be marshaled/unmarshaled to/from a sequence<byte>.
//
const StreamHelperCategory StreamHelperCategoryProtobuf = 100;

//
// All classes derived from ::google::protobuf::MessageLite will use this StreamableTraits
//
template<typename T>
struct StreamableTraits<T, typename std::enable_if<std::is_base_of< ::google::protobuf::MessageLite, T>::
value >::type>
{
    static const StreamHelperCategory helper = StreamHelperCategoryProtobuf;
    static const int minWireSize = 1;
    static const bool fixedLength = false;
};
}
```

`StreamHelperCategory` is an integer (`int`). Values between 0 and 20 are reserved for Ice.

Your `StreamableTraits` specialization must be defined in namespace `Ice` and must provide the following static const members:

- `static const StreamHelperCategory helper`
  The helper category you defined earlier.

- `static const int minWireSize`
  The minimum size (in bytes) of a marshal object of this type. This value should be one for sequences and dictionaries, since an empty sequence or dictionary is marshaled as a single byte (size = 0).

- `static const bool fixedLength`
  `true` when this object is always encoded on the same number of bytes, and `false` otherwise. For sequences and dictionaries, it should be `false`.

As shown in the example above, the second template parameter of `StreamableTraits` allows you to define a specialization for a family of types, such as all the types derived from `::google::protobuf::MessageLite`. If you want to provide a specialization for a single type, you can ignore this second template parameter (it defaults to `void` in the `StreamableTraits` base template).

Then, you need to provide a `StreamHelper` specialization or partial specialization in namespace `Ice` with `read` and `write` template member functions, for example:

```
namespace Ice
{

//
// This helper marshals a Protobuf into a sequence<byte>, and unmarshals a sequence<byte>
// into a Protobuf
//
template<typename T>
struct StreamHelper<T, StreamHelperCategoryProtobuf>
{
    template<class S> static inline void
    write(S* stream, const T& v)
    {
        std::vector<Byte> data(v.ByteSize());
        // ... marshal v into a sequence of bytes...
        stream->write(&data[0], &data[0] + data.size());
    }

    template<class S> static inline void
    read(S* stream, T& v)
    {
        std::pair<const Byte*, const Byte*> data;
        stream->read(data);
        //... unmarshal data into v...
    }
};
}
```

Your `write` function is responsible to marshal the parameter into the provided `stream`. You can assume that S is an `Ice::OutputStream`, even though Ice may not instantiate your `write` function with this type.

Likewise, your `read` function reads from the provided `stream`, and unmarshals into the provided parameter. You can assume that S is an `Ice::InputStream` in your implementation of `read`, even though Ice may not instantiate your `read` function with this type.

If your application uses a single custom type, you can skip the definition of a new `StreamHelperCategory` value and associated `StreamableTraits` specialization, and simply use `StreamHelperCategoryUnknown` as the second template parameter of your `StreamHelper` specialization or partial specialization.

Finally, if you intend to define optional data members or optional parameters with this custom type, you need to tell Ice how to encode such optional members with this type, for example:

```
namespace Ice
{
template<>
struct GetOptionalFormat<StreamHelperCategoryProtobuf, 1, false>
{
    static const OptionalFormat value = OptionalFormatVSize;
};
}
```

`GetOptionalFormat` is specialized using the constants you provided in the `StreamableTraits` specialization or partial specialization of your custom type. The `static const OptionalFormat value` represents the optional format described in the data encoding.

The `OptionalFormat` provided by `GetOptionalFormat` for `StreamHelperCategoryUnknown` and its default `StreamableTraits` is `OptionalFormatVSize`, like in the example above.

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 only to:

- Input parameters, on the client-side and on the server-side
- Out and return parameters provided by the Ice run-time to AMI type-safe callbacks and AMI lambdas
- Out and return parameters provided to AMD callbacks

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 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).

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 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).

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 standard 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