The InputStream Interface in C++

On this page:

The `InputStream` API in C++

An `InputStream` is created using the following function:

C++

```
namespace Ice {
    InputStreamPtr createInputStream(
        const Ice::CommunicatorPtr& communicator,
        const std::vector<Ice::Byte>& data);
}
```

The `InputStream` class is shown below.

C++

```
namespace Ice {
    class InputStream : ... {
    public:
        virtual CommunicatorPtr communicator() const = 0;

        virtual void sliceObjects(bool slice) = 0;

        virtual void read(bool& v) = 0;
        virtual void read(Byte& v) = 0;
        virtual void read(Short& v) = 0;
        virtual void read(Int& v) = 0;
        virtual void read(Long& v) = 0;
        virtual void read(Float& v) = 0;
        virtual void read(Double& v) = 0;
        virtual void read(std::string& s, bool convert = true) = 0;
        virtual void read(std::wstring& s) = 0;

        template<typename T> inline void read(T& v) {
            StreamReader< StreamTrait<T>::type>::read(this, v);
        }

        virtual void read(std::vector<std::string>& v, bool convert) = 0;

        virtual void read(std::pair<const bool*, const bool*>&,
                          IceUtil::ScopedArray<bool>&) = 0;

        virtual void read(std::pair<const Byte*, const Byte*>&) = 0;

        virtual void read(std::pair<const Short*, const Short*>&,
                          IceUtil::ScopedArray<Short>&) = 0;

        virtual void read(std::pair<const Int*, const Int*>&,
                          IceUtil::ScopedArray<Int>&) = 0;
```

```
        virtual void read(std::pair<const Long*, const Long*>&,
                          IceUtil::ScopedArray<Long>&) = 0;

        virtual void read(std::pair<const Float*, const Float*>&,
                          IceUtil::ScopedArray<Float>&) = 0;

        virtual void read(std::pair<const Double*, const Double*>&,
                          IceUtil::ScopedArray<Double>&) = 0;

        virtual Int readSize() = 0;
        virtual Int readAndCheckSeqSize(int minWireSize) = 0;

        virtual ObjectPrx readProxy() = 0;

        template<typename T> inline void
        read(IceInternal::ProxyHandle<T>& v) {
            // ...
        }

        virtual void readObject(const ReadObjectCallbackPtr& cb) = 0;

        template<typename T> inline void
        read(IceInternal::Handle<T>& v) {
            // ...
        }

        virtual std::string readTypeId() = 0;

        virtual void throwException() = 0;

        virtual void startSlice() = 0;
        virtual void endSlice() = 0;
        virtual void skipSlice() = 0;

        virtual void startEncapsulation() = 0;
        virtual void endEncapsulation() = 0;
        virtual void skipEncapsulation() = 0;

        virtual void readPendingObjects() = 0;

        virtual void rewind() = 0;
    };
    typedef ... InputStreamPtr;
}
```

Extracting Built-In Types in C++

Member functions are provided to extract any of the built-in types. For example, you can extract a double value followed by a string from a stream as follows:

C++

```
vector<Ice::Byte> data = ...;
in = Ice::createInputStream(communicator, data);
double d;
in->read(d);
string s;
in->read(s);
```

Extracting Sequences of Built-In Types in C++

For types other than built-in types, the following template member function performs the extraction:

C++

```
template<typename T> inline void
read(T& v) {
    StreamReader<StreamTrait<T>::type>::read(this, v);
}
```

For example, you can extract a sequence of integers as follows:

C++

```
vector<Ice::Byte> data = ...;
in = Ice::createInputStream(communicator, data);
// ...
IntSeq s; // Slice: sequence<int> IntSeq;
in->read(s);
```

The Ice run time provides an implementation of the `StreamReader` template whose `read` method reads a sequence of any of the built-in types. Note that, when reading a sequence, this reads both the sequence size that precedes the sequence elements as well as the sequence elements that follow the size.

If you are using a custom container for your sequence of built-in type, you must provide a specialization of the `StreamTrait` template in order to extract your sequence. For example, the following definition allows you to use the `QVector` container from the Qt library:

C++

```
//
// StreamTrait specialization for QVector
//
template<typename T>
struct StreamTrait< QVector<T> >
{
    static const StreamTraitType type = StreamTraitTypeSequence;
    static const int minWireSize = 1;
};
```

Extracting Sequences of Built-In Types using Zero-Copy in C++

`InputStream` provides a number of overloads that accept a pair of pointers. For example, you can extract a sequence of bytes as follows:

C++

```
vector<Ice::Byte> data = ...;
in = Ice::createInputStream(communicator, data);
std::pair<const Ice::Byte*, const Ice::Byte*> p;
in->read(p);
```

The same extraction works for the other built-in integral and floating-point types, such `int` and `double`.

If the extraction is for a byte sequence, the returned pointers always point at memory in the stream's internal marshaling buffer.

For the other built-in types, the pointers refer to the internal marshaling buffer only if the Ice encoding is compatible with the machine and compiler representation of the type, otherwise the pointers refer to a temporary array allocated to hold the unmarshaled data. The overloads for zero-copy extraction accept an additional parameter of type `IceUtil::ScopedArray` that holds this temporary array when necessary.

Here is an example to illustrate how to extract a sequence of integers, regardless of whether the machine's encoding of integers matches the on-the-wire representation or not:

C++

```
#include <IceUtil/ScopedArray.h>
...
in = Ice::createInputStream(communicator, data);
std::pair<const Ice::Int*, const Ice::Int*> p;
IceUtil::ScopedArray<Ice::Int> a;
in->read(p, a);

for(const Ice::Int* i = p.first; i != p.second; ++i) {
    cout << *i << endl;
}
```

If the on-the-wire encoding matches that of the machine, and therefore zero-copy is possible, the returned pair of pointers points into the run time's internal marshaling buffer. Otherwise, the run time allocates an array, unmarshals the data into the array, and sets the pair of pointers to point into that array. Use of the `ScopedArray` helper template ensures that the array is deallocated once you let the `ScopedArray` go out of scope, so there is no need to call `delete[]`. (`ScopedArray` is conceptually the same as the `Ptr` smart pointer types for classes.

Extracting Structures in C++

Without the `--stream` option to `slice2cpp`, you must extract structures member by member according to the data encoding rules. Otherwise, with `--stream`, `slice2cpp` generates code that allows you to extract the structure directly. For example, here is how you can extract a Slice structure called `MyStruct` from a stream:

C++

```
in = Ice::createInputStream(communicator, data);
MyStruct myStruct;
in->read(myStruct);
```

Extracting Dictionaries in C++

Without the `--stream` option to `slice2cpp`, you can extract any dictionary whose key and value types are built-in types; for any other dictionary, you must extract it as a size followed by its entries according to the data encoding rules. If you are using a custom container for your dictionary of built-in types, you must provide a specialization of the `StreamTrait` template in order to extract your dictionary. For example, the following definition allows you to use the `QMap` container from the Qt library:

C++

```
//
// StreamTrait specialization for QMap
//
template<typename K, typename V>
struct StreamTrait< QMap<K, V> >
{
    static const StreamTraitType type = StreamTraitTypeDictionary;
    static const int minWireSize = 1;
};
```

With the `--stream` option, `slice2cpp` generates code that allows you to extract any dictionary directly, for example:

C++

```
in = Ice::createInputStream(communicator, data);
MyDict myDict; // Slice: dictionary<string, SomeType> MyDict;
in->read(myDict);
```

Extracting Sequences of User-Defined Types in C++

Without the `--stream` option to `slice2cpp`, you must extract sequences of user-defined type as a size followed by the element type according to the data encoding rules. Otherwise, with `--stream`, `slice2cpp` generates code that allows you to extract a sequence directly, for example:

C++

```
in = Ice::createInputStream(communicator, data);
MyEnumS myEnumS; // Slice: sequence<MyEnum> myEnumS;
in->read(myEnumS);
```

Other `InputStream` Methods in C++

The remaining member functions of `InputStream` have the following semantics:

- `void sliceObjects(bool slice)`
  Determines the behavior of the stream when extracting Ice objects. An Ice object is "sliced" when a factory cannot be found for a Slice type ID, resulting in the creation of an object of a less-derived type. Slicing is typically disabled when the application expects all object factories to be present, in which case the exception `NoObjectFactoryException` is raised. The default behavior is to allow slicing.

- `void read(std::string& v, bool convert = true)`
  `void read(std::vector<std::string>& v, bool convert = true)`
  The optional boolean argument determines whether the strings unmarshaled by these methods are processed by the string converter, if one is installed. The default behavior is to convert the strings.

- `Ice::Int readSize()`
  The Ice encoding has a compact representation to indicate size. This function extracts a size and returns it as an integer.

- `Ice::Int readAndCheckSeqSize(int minWireSize)`
  Like `readSize`, this function reads a size and returns it, but also verifies that there is enough data remaining in the unmarshaling buffer to successfully unmarshal the elements of the sequence. The `minWireSize` parameter indicates the smallest possible on-the-wire representation of a single sequence element. If the unmarshaling buffer contains insufficient data to unmarshal the sequence, the function throws `UnmarshalOutOfBoundsException`.

- `Ice::ObjectPrx readProxy()`
  This function returns an instance of the base proxy type, `ObjectPrx`.

- `template<typename T> inline void`
  `read(IceInternal::ProxyHandle<T>& v)`
  This template function behaves like `readProxy` but avoids the need to down-cast the return value. You can pass a proxy of any type as the parameter `v`.

- `void readObject(const Ice::ReadObjectCallbackPtr &)`
  The Ice encoding for class instances requires extraction to occur in stages. The `readObject` function accepts a callback object of type `ReadObjectCallback`, whose definition is shown below:

  C++

  ```
  namespace Ice {
      class ReadObjectCallback : ... {
      public:
          virtual void invoke(const Ice::ObjectPtr&) = 0;
      };
      typedef ... ReadObjectCallbackPtr;
  }
  ```

  When the object instance is available, the callback object's `invoke` member function is called. The application must call `readPendingObjects` to ensure that all instances are properly extracted. If you are not interested in receiving a callback when the object is extracted, it is easier to use the `read(IceInternal::Handle<T>&)` template function instead (see below).

- `template<typename T> inline void read(IceInternal::Handle<T>& v)`
  This template function behaves like `readObject` but avoids the need to supply a callback. You can pass a smart pointer of any type as the parameter `v`. Note that, if you want to intercept object extraction, you must use `readObject` instead.

- `std::string readTypeId()`
  A table of Slice type IDs is used to save space when encoding Ice objects. This function returns the type ID at the stream's current position.

- `void throwException()`
  This function extracts a user exception from the stream and throws it. If the stored exception is of an unknown type, the function attempts to extract and throw a less-derived exception. If that also fails, an `UnmarshalOutOfBoundsException` is thrown.

- `void startSlice()`
  `void endSlice()`
  `void skipSlice()`
  Start, end, and skip a slice of member data, respectively. These functions are used when manually extracting the slices of an Ice object or user exception.

- `void startEncapsulation()`
  `void endEncapsulation()`
  `void skipEncapsulation()`
  Start, end, and skip an encapsulation, respectively.

- `void readPendingObjects()`
  An application must call this function after all other data has been extracted, but only if Ice objects were encoded. This function extracts the state of Ice objects and invokes their corresponding callback objects (see `readObject`).

- `void rewind()`
  Resets the position of the stream to the beginning.

See Also

- Smart Pointers for Classes
- slice2cpp Command-Line Options
- C++ Strings and Character Encoding
- Data Encoding for Classes
- Basic Data Encoding
- The C++ ScopedArray Template