The InputStream Interface in C++

On this page:

- The InputStream API in C++
- Extracting from an InputStream in C++
- Extracting Sequences of Built-In Types using Zero-Copy in C++
- Other InputStream Methods in C++

The `InputStream` API in C++

An `InputStream` that uses the Ice encoding can be created using the following functions:

C++

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

    InputStreamPtr createInputStream(
        const CommunicatorPtr& communicator,
        const std::vector<Ice::Byte>& data,
        const EncodingVersion& version);

    InputStreamPtr createInputStream(
        const CommunicatorPtr& communicator,
        const std::pair<const Ice::Byte*, const Ice::Byte*>& data);

    InputStreamPtr createInputStream(
        const CommunicatorPtr& communicator,
        const std::pair<const Ice::Byte*, const Ice::Byte*>& data,
        const EncodingVersion& version);

    InputStreamPtr wrapInputStream(
        const CommunicatorPtr& communicator,
        const std::vector<Ice::Byte>& data);

    InputStreamPtr wrapInputStream(
        const CommunicatorPtr& communicator,
        const std::vector<Ice::Byte>& data,
        const EncodingVersion& version);

    InputStreamPtr wrapInputStream(
        const CommunicatorPtr& communicator,
        const std::pair<const Ice::Byte*, const Ice::Byte*>& data);

    InputStreamPtr wrapInputStream(
        const CommunicatorPtr& communicator,
        const std::pair<const Ice::Byte*, const Ice::Byte*>& data,
        const EncodingVersion& version);
}
```

You can optionally specify an encoding version for the stream, otherwise the stream uses the communicator's default encoding version.

Note that the `createInputStream` functions make a copy of the supplied data, whereas the `wrapInputStream` functions avoid copies by keeping a reference to the data. If you use `wrapInputStream`, it is your responsibility to ensure that the memory buffer remains valid and unmodified for the lifetime of the `InputStream` object.

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

        virtual void read(::std::vector<bool>&) = 0;

        Int readEnum(Int maxValue) { ... }

        template<typename T> inline void
        read(T& 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 void throwException() = 0;
        virtual void throwException(const UserExceptionReaderFactoryPtr&) = 0;

        virtual void startObject() = 0;
        virtual SlicedDataPtr endObject(bool preserve) = 0;

        virtual void startException() = 0;
```

```
        virtual SlicedDataPtr endException(bool preserve) = 0;

        virtual std::string startSlice() = 0;
        virtual void endSlice() = 0;
        virtual void skipSlice() = 0;

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

        virtual EncodingVersion getEncoding() = 0;

        virtual void readPendingObjects() = 0;

        virtual void rewind() = 0;

        virtual void skip(Int sz) = 0;
        virtual void skipSize() = 0;

        virtual bool readOptional(Int tag, OptionalFormat fmt) = 0;

        virtual void closure(void* p) = 0;
        virtual void* closure() const = 0;
    };
    typedef ... InputStreamPtr;
}
```

Extracting from an InputStream in C++

An `InputStream` provides a number of overloaded `read` member functions that allow you to read any parameter from the stream by simply calling `read`.

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);
```

Likewise, you can extract a sequence of a built-in type, or a complex type, or any other type from the stream as follows:

C++

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

ComplexType c;
in->read(c);
```

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::wrapInputStream(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::wrapInputStream(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 comparable to a `std::unique_ptr` for arrays.

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)`
  The 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.

- `Int readEnum(Int maxValue)`
  Unmarshals the integer value of an enumerator. The `maxValue` argument represents the highest enumerator value in the enumeration. Consider the following definitions:

  Slice

  ```
  enum Color { red, green, blue };
  enum Fruit { Apple, Pear=3, Orange };
  ```

  The maximum value for `Color` is 2, and the maximum value for `Fruit` is 4.
  In general, you should simply use `read` for your enum values. `read` with an enum parameter calls `readEnum` with the `maxValue` provided by the code generated by `slice2cpp`.

- `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`. Calling `read` with a proxy parameter has the same effect.

- `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 call `read` with a `Ptr` parameter. Note that calling `endEncapsulation` implicitly calls `readPendingObjects` if necessary.

- `void throwException()`
  `void throwException(const UserExceptionReaderFactoryPtr& factory)`
  These functions extract a user exception from the stream and throw it. If the stored exception is of an unknown type, the functions attempt to extract and throw a less-derived exception. If that also fails, an exception is thrown: for the 1.0 encoding, the exception is `UnmarshalOutOfBoundsException`, for the 1.1 encoding, the exception is `UnknownUserException`.

- `void startObject()`
  `SlicedDataPtr endObject(bool preserve)`
  The `startObject` method must be called prior to reading the slices of an object. The `endObject` method must be called after all slices have been read. Pass true to `endObject` in order to preserve the slices of any unknown more-derived types, or false to discard the slices. If `preserve` is true and the stream actually preserved any slices, the return value of `endObject` is a non-nil `SlicedData` object that encapsulates the slice data. If the caller later wishes to forward the object with any preserved slices intact, it must supply this `SlicedData` object to the output stream.

- `void startException()`
  `SlicedDataPtr endException(bool preserve)`
  The `startException` method must be called prior to reading the slices of an exception. The `endException` method must be called after all slices have been read. Pass true to `endException` in order to preserve the slices of any unknown more-derived types, or false to discard the slices. If `preserve` is true and the stream actually preserved any slices, the return value of `endException` is a non-nil `SlicedData` object that encapsulates the slice data. If the caller later wishes to forward the exception with any preserved slices intact, it must supply this `SlicedData` object to the output stream.

- `std::string 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. The `startSlice` method returns the type ID of the next slice, which may be an empty string depending on the format used to encode the object or exception.

- `EncodingVersion startEncapsulation()`
  `void endEncapsulation()`
  `EncodingVersion skipEncapsulation()`
  Start, end, and skip an encapsulation, respectively. The `startEncapsulation` and `skipEncapsulation` methods return the encoding version used to encode the contents of the encapsulation.

- `EncodingVersion getEncoding()`
  Returns the encoding version currently in use by the stream.

- `void readPendingObjects()`
  An application must call this function after all other data has been extracted. This function extracts the state of Ice objects and invokes their

corresponding callback objects (see `readObject`). For backward compatibility with encoding version 1.0, this function must only be called when non-optional data members or parameters use class types.

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

- `void skip(Int sz)`
  Skips the given number of bytes.

- `void skipSize()`
  Reads a size at the current position and skips that number of bytes.

- `bool readOptional(Int tag, OptionalFormat fmt)`
  Returns true if an optional value with the given tag and format is present, or false otherwise. If this method returns true, the data associated with that optional value must be read next. Optional values must be read in order by tag from least to greatest. The `OptionalFormat` enumeration is defined as follows:

  C++

  ```
  namespace Ice {
      enum OptionalFormat {
          OptionalFormatF1, OptionalFormatF2, OptionalFormatF4, OptionalFormatF8,
          OptionalFormatSize, OptionalFormatVSize, OptionalFormatFSize,
          OptionalFormatEndMarker
      };
  }
  ```

  Refer to the encoding discussion for more information on the meaning of these values.

- `void closure(void* p)`
  `void* closure() const`
  Allows an arbitrary value to be associated with the stream.

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