The OutputStream Interface in C++

On this page:

- The OutputStream API in C++
- Inserting Built-In Types in C++
- Inserting Sequences of Built-In Types in C++
- Inserting Sequences of Built-In Types using Zero-Copy in C++
- Inserting Structures in C++
- Inserting Dictionaries in C++
- Inserting Sequences of User-Defined Types in C++
- Other OutputStream Methods in C++

The `OutputStream` API in C++

An `OutputStream` is created using the following function:

C++

```
namespace Ice {
    OutputStreamPtr createOutputStream(const Ice::CommunicatorPtr& communicator);
}
```

The `OutputStream` class is shown below.

C++

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

        virtual void write(bool v) = 0;
        virtual void write(Byte v) = 0;
        virtual void write(Short v) = 0;
        virtual void write(Int v) = 0;
        virtual void write(Long v) = 0;
        virtual void write(Float v) = 0;
        virtual void write(Double v) = 0;
        virtual void write(const std::string& v, bool convert = true) = 0;
        virtual void write(const char* v, bool convert = true) = 0;
        virtual void write(const std::wstring& v) = 0;

        virtual void write(const bool* begin, const bool* end) = 0;
        virtual void write(const Byte* begin, const Byte* end) = 0;
        virtual void write(const Short* begin, const Short* end) = 0;
        virtual void write(const Int* begin, const Int* end) = 0;
        virtual void write(const Long* begin, const Long* end) = 0;
        virtual void write(const Float* begin, const Float* end) = 0;
        virtual void write(const Double* begin, const Double* end) = 0;

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

        template<typename T> inline void
        write(const T& v) {
            StreamWriter<StreamTrait<T>::type>::write(this, v);
        }

        virtual void writeSize(Ice::Int sz) = 0;

        virtual void writeProxy(const Ice::ObjectPrx& v) = 0;

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

        virtual void writeObject(const Ice::ObjectPtr& v) = 0;

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

        virtual void writeTypeId(const std::string& id) = 0;

        virtual void writeException(const Ice::UserException& e) = 0;

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

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

        virtual void writePendingObjects() = 0;

        virtual void finished(std::vector<Ice::Byte>& v) = 0;

        virtual void reset(bool) = 0;
    };
}
```

Inserting Built-In Types in C++

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

C++

```
out = Ice::createOutputStream(communicator);
Ice::Double d = 3.14;
out->write(d);
string s = "Hello";
out->write(s);
```

Inserting Sequences of Built-In Types in C++

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

C++

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

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

C++

```
out = Ice::createOutputStream(communicator);
IntSeq s = ...;
out->write(s);
```

The Ice run time provides an implementation of the `StreamWriter` template whose `write` method writes a sequence of any of the built-in types. Note that, when writing a sequence, this writes both the sequence size that precedes the sequence elements and 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 insert 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;
};
```

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

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

C++

```
out = Ice::createOutputStream(communicator);
vector<Ice::Byte> data = ...;
out->write(&v[0], &v[v.size()]);
```

The same insertion technique works for the other built-in integral and floating-point types, such `int` and `double`. Insertion in this way can avoid an additional data copy during marshaling if the internal representation of the data in memory is the same as the on-the-wire representation. (Note that the two pointers must point at a contiguous block of memory.)

Inserting Structures in C++

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

C++

```
out = Ice::createOutputStream(communicator);
MyStruct myStruct;
// Initialize myStruct...
out->write(myStruct);
```

Inserting Dictionaries in C++

Without the `--stream` option to `slice2cpp`, you can insert any dictionary whose key and value types are built-in types; for any other dictionary, you must insert 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 insert 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 insert any dictionary directly, for example:

C++

```
out = Ice::createOutputStream(communicator);
MyDict myDict; // Slice: dictionary<int, SomeType> MyDict;
// Initialize myDict...
out->write(myDict);
```

Inserting Sequences of User-Defined Types in C++

Without the `--stream` option to `slice2cpp`, you must insert 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 insert a sequence directly, for example:

C++

```
out = Ice::createOutputStream(communicator);
MyEnumS myEnumS; // Slice: sequence<MyEnum> myEnumS;
// Initialize myEnumS...
out->write(myEnumS);
```

Other `OutputStream` Methods in C++

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

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

- `void writeSize(Ice::Int sz)`
  The Ice encoding has a compact representation to indicate size. This function converts the given non-negative integer into the proper encoded representation.

- `void writeProxy(const Ice::ObjectPtr & v)`
  Inserts a proxy.

- `template<typename T> inline void`
  `write(const IceInternal::Handle<T>& v)`
  This template function behaves like `writeObject`. You can pass a smart pointer of any type as the parameter `v`.

- `void writeObject(const Ice::ObjectPtr & v)`
  Inserts an Ice object. The Ice encoding for class instances may cause the insertion of this object to be delayed, in which case the stream retains a reference to the given object and the stream does not insert its state it until `writePendingObjects` is invoked on the stream.

- `template<typename T> inline void`
  `write(const IceInternal::ProxyHandle<T>& v)`
  This template function behaves like `writeProxy`. You can pass a proxy of any type as the parameter `v`.

- `void writeTypeId(const std::string & id)`
  A table of Slice type IDs is used to save space when encoding Ice objects. This function adds the given type ID to the table and encodes the type ID. `writeTypeId` may only be invoked in the context of a call to `writePendingObjects` (see below).

- `void writeException(const Ice::UserException & ex)`
  Inserts a user exception. You can also use the template member function `write(const T&)` to insert a user exception.

- `void startSlice()`
  `void endSlice()`
  Starts and ends a slice of object or exception member data.

- `void startEncapsulation()`
  `void endEncapsulation()`
  Starts and ends an encapsulation, respectively.

- `void writePendingObjects()`
  Encodes the state of Ice objects whose insertion was delayed during `writeObject`. This member function must only be called once.

- `void finished(std::vector< Ice::Byte > & data)`
  Indicates that marshaling is complete. The given byte sequence is filled with the encoded data. This member function must only be called once.

- `void reset(bool clearBuffer)`
  Resets the writing position of the stream to the beginning. If `clearBuffer` is true, the stream releases the memory it has allocated to hold the encoded data.

See Also

- Basic Data Encoding

- slice2cpp Command-Line Options
- C++ Strings and Character Encoding
- Data Encoding for Classes
- Data Encoding for Exceptions