C++ Mapping for Classes

On this page:

- Basic C++ Mapping for Classes
- Inheritance from Ice::Object in C++
- Class Data Members in C++
- Class Constructors in C++
- Class Operations in C++
- Class Factories in C++

Basic C++ Mapping for Classes

A Slice class is mapped to a C++ class with the same name. The generated class contains a public data member for each Slice data member (just as for structures and exceptions), and a virtual member function for each operation. Consider the following class definition:

Slice

```
class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

The Slice compiler generates the following code for this definition:

C++

```
class TimeOfDay;

typedef IceInternal::ProxyHandle<IceProxy::TimeOfDay> TimeOfDayPrx;
typedef IceInternal::Handle<TimeOfDay> TimeOfDayPtr;

class TimeOfDay : virtual public Ice::Object {
public:
    Ice::Short hour;
    Ice::Short minute;
    Ice::Short second;

    virtual std::string format() = 0;

    TimeOfDay() {};
    TimeOfDay(Ice::Short, Ice::Short, Ice::Short);

    virtual bool ice_isA(const std::string&);
    virtual const std::string& ice_id();
    static const std::string& ice_staticId();

    typedef TimeOfDayPrx ProxyType;
    typedef TimeOfDayPtr PointerType;

    // ...
};
```

The ProxyType and PointerType definitions are for template programming.

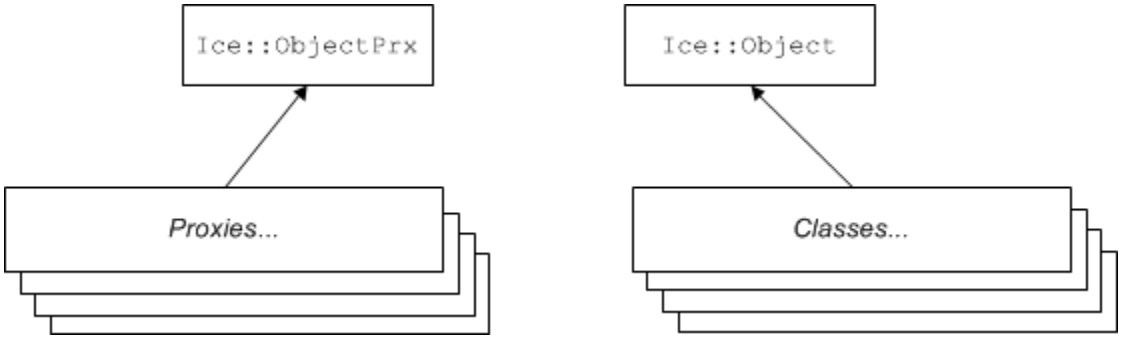
There are a number of things to note about the generated code:

1. The generated class `TimeOfDay` inherits from `Ice::Object`. This means that all classes implicitly inherit from `Ice::Object`, which is the ultimate ancestor of all classes. Note that `Ice::Object` is *not* the same as `IceProxy::Ice::Object`. In other words, you *cannot* pass a class where a proxy is expected and vice versa.
2. The generated class contains a public member for each Slice data member.
3. The generated class has a constructor that takes one argument for each data member, as well as a default constructor.
4. The generated class contains a pure virtual member function for each Slice operation.
5. The generated class contains additional member functions: `ice_isA`, `ice_id`, `ice_staticId`, and `ice_factory`.
6. The compiler generates a type definition `TimeOfDayPtr`. This type implements a smart pointer that wraps dynamically-allocated instances of the class. In general, the name of this type is <*`class-name`*>`Ptr`. Do not confuse this with <*`class-name`*>`Prx` — that type exists as well, but is the proxy handle for the class, not a smart pointer.

There is quite a bit to discuss here, so we will look at each item in turn.

Inheritance from `Ice::Object` in C++

Like interfaces, classes implicitly inherit from a common base class, `Ice::Object`. However, as shown in the figure below, classes inherited from `Ice::Object` instead of `Ice::ObjectPrx` (which is at the base of the inheritance hierarchy for proxies). As a result, you cannot pass a class where a proxy is expected (and vice versa) because the base types for classes and proxies are not compatible.



Inheritance from `Ice::ObjectPrx` and `Ice::Object`.

`Ice::Object` contains a number of member functions:

C++

```
namespace Ice {
    class Object : public virtual IceInternal::GCShared {
    public:
        virtual bool ice_isA(const std::string&, const Current& = Current()) const;
        virtual void ice_ping(const Current&  = Current()) const;
        virtual std::vector<std::string> ice_ids(const Current& = Current()) const;
        virtual const std::string& ice_id(const Current& = Current()) const;
        static const std::string& ice_staticId();
        virtual ObjectPtr ice_clone() const;

        virtual void ice_preMarshal();
        virtual void ice_postUnmarshal();

        virtual DispatchStatus ice_dispatch(
            Ice::Request&,
            const DispatchInterceptorAsyncCallbackPtr& = 0);

        virtual bool operator==(const Object&) const;
        virtual bool operator!=(const Object&) const;
        virtual bool operator<(const Object&) const;
        virtual bool operator<=(const Object&) const;
        virtual bool operator>(const Object&) const;
        virtual bool operator>=(const Object&) const;
    };
}
```

The member functions of `Ice::Object` behave as follows:

- `ice_isA`
  This function returns `true` if the object supports the given type ID, and `false` otherwise.
- `ice_ping`
  As for interfaces, `ice_ping` provides a basic reachability test for the class. Note that `ice_ping` is normally only invoked on the proxy for a class that might be remote because a class instance that is local (in the caller's address space) can always be reached.
- `ice_ids`
  This function returns a string sequence representing all of the type IDs supported by this object, including `::Ice::Object`.
- `ice_id`
  This function returns the actual run-time type ID for a class. If you call `ice_id` through a smart pointer to a base instance, the returned type id is the actual (possibly more derived) type ID of the instance.
- `ice_staticId`
  This function returns the static type ID of a class.
- `ice_clone`
  This function makes a polymorphic shallow copy of a class.
- `ice_preMarshal`
  The Ice run time invokes this function prior to marshaling the object's state, providing the opportunity for a subclass to validate its declared data members.
- `ice_postUnmarshal`
  The Ice run time invokes this function after unmarshaling an object's state. A subclass typically overrides this function when it needs to perform additional initialization using the values of its declared data members.
- `ice_dispatch`
  This function dispatches an incoming request to a servant. It is used in the implementation of dispatch interceptors.
- `operator==`
  `operator!=`
  `operator<`
  `operator<=`
  `operator>`
  `operator>=`
  The comparison operators permit you to use classes as elements of STL sorted containers. Note that sort order, unlike for structures, is based on the memory address of the class, not on the contents of its data members of the class.

Class Data Members in C++

By default, data members of classes are mapped exactly as for structures and exceptions: for each data member in the Slice definition, the generated class contains a corresponding public data member. Optional data members are mapped to instances of the `IceUtil::Optional` template.

If you wish to restrict access to a data member, you can modify its visibility using the `protected` metadata directive. The presence of this directive causes the Slice compiler to generate the data member with protected visibility. As a result, the member can be accessed only by the class itself or by one of its subclasses. For example, the `TimeOfDay` class shown below has the `protected` metadata directive applied to each of its data members:

Slice

```
class TimeOfDay {
    ["protected"] short hour;   // 0 - 23
    ["protected"] short minute; // 0 - 59
    ["protected"] short second; // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

The Slice compiler produces the following generated code for this definition:

C++

```
class TimeOfDay : virtual public Ice::Object {
public:

    virtual std::string format() = 0;

    // ...

protected:

    Ice::Short hour;
    Ice::Short minute;
    Ice::Short second;
};
```

For a class in which all of the data members are protected, the metadata directive can be applied to the class itself rather than to each member individually. For example, we can rewrite the `TimeOfDay` class as follows:

Slice

```
["protected"] class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

Class Constructors in C++

Classes have a default constructor that default-constructs each data member. Members having a complex type, such as strings, sequences, and dictionaries, are initialized by their own default constructor. However, the default constructor performs no initialization for members having one of the simple built-in types boolean, integer, floating point, or enumeration. For such a member, it is not safe to assume that the member has a reasonable default value. This is especially true for enumerated types as the member's default value may be outside the legal range for the enumeration, in which case an exception will occur during marshaling unless the member is explicitly set to a legal value.

To ensure that data members of primitive types are initialized to reasonable values, you can declare default values in your Slice definition. The default constructor initializes each of these data members to its declared value. Optional data members are unset unless they declare default values.

Classes also have a second constructor that has one parameter for each data member. This allows you to construct and initialize a class instance in a single statement. For each optional data member, its corresponding constructor parameter uses the same mapping as for operation parameters, allowing you to pass its initial value or `IceUtil::None` to indicate an unset value.

For derived classes, the constructor has one parameter for each of the base class's data members, plus one parameter for each of the derived class's data members, in base-to-derived order. For example:

Slice

```
class Base {
    int i;
};

class Derived extends Base {
    string s;
};
```

This generates:

C++

```
class Base : virtual public ::Ice::Object
{
public:
    ::Ice::Int i;

    Base() {};
    explicit Base(::Ice::Int);

    // ...
};

class Derived : public Base
{
public:
    ::std::string s;

    Derived() {};
    Derived(::Ice::Int, const ::std::string&);

    // ...
};
```

Note that single-parameter constructors are defined as `explicit`, to prevent implicit argument conversions.

By default, derived classes derive non-virtually from their base class. If you need virtual inheritance, you can enable it using the `["cpp:virtual"]` metadata directive.

Class Operations in C++

Operations of classes are mapped to pure virtual member functions in the generated class. This means that, if a class contains operations (such as the `format` operation of our `TimeOfDay` class), you must provide an implementation of the operation in a class that is derived from the generated class. For example:

C++

```
class TimeOfDayI : virtual public TimeOfDay {
public:
    virtual std::string format() {
        std::ostringstream s;
        s << setw(2) << setfill('0') << hour << ":";
        s << setw(2) << setfill('0') << minute << ":";
        s << setw(2) << setfill('0') << second;
        return s.c_str();
    }

protected:
    virtual ~TimeOfDayI() {}  // Optional
};
```

We discuss the motivation for the protected destructor in Preventing Stack-Allocation of Class Instances.

Class Factories in C++

Having created a class such as `TimeOfDayI`, we have an implementation and we can instantiate the `TimeOfDayI` class, but we cannot receive it as the return value or as an out-parameter from an operation invocation. To see why, consider the following simple interface:

Slice

```
interface Time {
    TimeOfDay get();
};
```

When a client invokes the `get` operation, the Ice run time must instantiate and return an instance of the `TimeOfDay` class. However, `TimeOfDay` is an abstract class that cannot be instantiated. Unless we tell it, the Ice run time cannot magically know that we have created a `TimeOfDayI` class that implements the abstract `format` operation of the `TimeOfDay` abstract class. In other words, we must provide the Ice run time with a factory that knows that the `TimeOfDay` abstract class has a `TimeOfDayI` concrete implementation. The `Ice::Communicator` interface provides us with the necessary operations:

Slice

```
module Ice {
    local interface ObjectFactory {
        Object create(string type);
        void destroy();
    };

    local interface Communicator {
        void addObjectFactory(ObjectFactory factory, string id);
        ObjectFactory findObjectFactory(string id);
        // ...
    };
};
```

To supply the Ice run time with a factory for our `TimeOfDayI` class, we must implement the `ObjectFactory` interface:

Slice

```
module Ice {
    local interface ObjectFactory {
        Object create(string type);
        void destroy();
    };
};
```

The object factory's `create` operation is called by the Ice run time when it needs to instantiate a `TimeOfDay` class. The factory's `destroy` operation is called by the Ice run time when its communicator is destroyed. A possible implementation of our object factory is:

C++

```
class ObjectFactory : public Ice::ObjectFactory {
public:
    virtual Ice::ObjectPtr create(const std::string& type) {
        assert(type == M::TimeOfDay::ice_staticId());
        return new TimeOfDayI;
    }
    virtual void destroy() {}
};
```

The `create` method is passed the type ID of the class to instantiate. For our `TimeOfDay` class, the type ID is `"::M::TimeOfDay"`. Our implementation of `create` checks the type ID: if it matches, the method instantiates and returns a `TimeOfDayI` object. For other type IDs, the method asserts because it does not know how to instantiate other types of objects.

Note that we used the `ice_staticId` method to obtain the type ID rather than embedding a literal string. Using a literal type ID string in your code is discouraged because it can lead to errors that are only detected at run time. For example, if a Slice class or one of its enclosing modules is renamed and the literal string is not changed accordingly, a receiver will fail to unmarshal the object and the Ice run time will raise `NoObjectFactoryException`. By using `ice_staticId` instead, we avoid any risk of a misspelled or obsolete type ID, and we can discover at compile time if a Slice class or module has been renamed.

Given a factory implementation, such as our `ObjectFactory`, we must inform the Ice run time of the existence of the factory:

C++

```
Ice::CommunicatorPtr ic = ...;
ic->addObjectFactory(new ObjectFactory, M::TimeOfDay::ice_staticId());
```

Now, whenever the Ice run time needs to instantiate a class with the type ID `"::M::TimeOfDay"`, it calls the `create` method of the registered `ObjectFactory` instance.

The `destroy` operation of the object factory is invoked by the Ice run time when the communicator is destroyed. This gives you a chance to clean up any resources that may be used by your factory. Do not call `destroy` on the factory while it is registered with the communicator — if you do, the Ice run time has no idea that this has happened and, depending on what your `destroy` implementation is doing, may cause undefined behavior when the Ice run time tries to next use the factory.

The run time guarantees that `destroy` will be the last call made on the factory, that is, `create` will not be called concurrently with `destroy`, and `create` will not be called once `destroy` has been called. However, calls to `create` can be made concurrently.

Note that you cannot register a factory for the same type ID twice: if you call `addObjectFactory` with a type ID for which a factory is registered, the Ice run time throws an `AlreadyRegisteredException`.

Finally, keep in mind that if a class has only data members, but no operations, you need not create and register an object factory to transmit instances of such a class. Only if a class has operations do you have to define and register an object factory.

See Also

- Classes
- Smart Pointers for Classes
- C++ Mapping for Operations
- C++ Mapping for Optional Values
- Asynchronous Method Invocation (AMI) in C++
- Dispatch Interceptors