C++ Mapping for Interfaces

The mapping of Slice interfaces revolves around the idea that, to invoke a remote operation, you call a member function on a local class instance that is a proxy for the remote object. This makes the mapping easy and intuitive to use because making a remote procedure call is no different from making a local procedure call (apart from error semantics).

On this page:

Proxy Classes and Proxy Handles

On the client side, a Slice interface maps to a class with member functions that correspond to the operations on that interface. Consider the following simple interface:

Slice

```
module M {
    interface Simple {
        void op();
    };
};
```

The Slice compiler generates the following definitions for use by the client:

C++

```
namespace IceProxy {
    namespace M {
        class Simple;
    }
}

namespace M {
    class Simple;
    typedef IceInternal::ProxyHandle< ::IceProxy::M::Simple> SimplePrx;
    typedef IceInternal::Handle< ::M::Simple> SimplePtr;
}

namespace IceProxy {
    namespace M {
        class Simple : public virtual IceProxy::Ice::Object {
        public:
            typedef ::M::SimplePrx ProxyType;
            typedef ::M::SimplePtr PointerType;

            void op();
            void op(const Ice::Context&);
            // ...
        };
    };
}
```

As you can see, the compiler generates a *proxy class* `Simple` in the `IceProxy::M` namespace, as well as a *proxy* handle `M::SimplePrx`. In general, for a module `M`, the generated names are `::IceProxy::M::<`*interface?name*`>` and `::M::<`*interface?name*`>Prx`.

In the client's address space, an instance of `IceProxy::M::Simple` is the local ambassador for a remote instance of the `Simple` interface in a server and is known as a *proxy class instance*. All the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance.

Inheritance from `Ice::Object`

`Simple` inherits from `IceProxy::Ice::Object`, reflecting the fact that all Ice interfaces implicitly inherit from `Ice::Object`. For each operation in the interface, the proxy class has two overloaded member functions of the same name. For the preceding example, we find that the operation `op` has been mapped to two member functions `op`.

One of the overloaded member functions has a trailing parameter of type `Ice::Context`. This parameter is for use by the Ice run time to store information about how to deliver a request; normally, you do not need to supply a value here and can pretend that the trailing parameter does not exist. (The parameter is also used by IceStorm.)

Proxy Handles

Client-side application code never manipulates proxy class instances directly. In fact, you are not allowed to instantiate a proxy class directly. The following code will not compile because `Ice::Object` is an abstract base class with a protected constructor and destructor:

C++

```
IceProxy::M::Simple s;  // Compile?time error!
```

Proxy instances are always instantiated on behalf of the client by the Ice run time, so client code never has any need to instantiate a proxy directly. When the client receives a proxy from the run time, it is given a *proxy handle* to the proxy, of type `<`*interface-name*`>Prx` (`SimplePrx` for the preceding example). The client accesses the proxy via its proxy handle; the handle takes care of forwarding operation invocations to its underlying proxy, as well as reference-counting the proxy. This means that no memory-management issues can arise: deallocation of a proxy is automatic and happens once the last handle to the proxy disappears (goes out of scope).

Because the application code always uses proxy handles and never touches the proxy class directly, we usually use the term proxy to denote both proxy handle and proxy class. This reflects the fact that, in actual use, the proxy handle looks and feels like the underlying proxy class instance. If the distinction is important, we use the terms *proxy class*, *proxy class instance*, and *proxy handle*.

`ProxyType` and `PointerType`

The generated proxy class contains type definitions for `ProxyType` and `PointerType`. These are provided so you can refer to the proxy type and smart pointer type in template definitions without having to resort to preprocessor trickery, for example:

C++

```
template<typename T>
class ProxyWrapper {
public:
    T::ProxyType proxy() const;
    // ...
};
```

Methods on Proxy Handles

As we saw for the preceding example, the handle is actually a template of type `IceInternal::ProxyHandle` that takes the proxy class as the template parameter. This template has the usual default constructor, copy constructor, and assignment operator.

Default constructor

You can default-construct a proxy handle. The default constructor creates a proxy that points nowhere (that is, points at no object at all). If you invoke an operation on such a null proxy, you get an `IceUtil::NullHandleException`:

C++

```
try {
    SimplePrx s;        // Default?constructed proxy
    s->op();            // Call via nil proxy
    assert(0);          // Can't get here
} catch (const IceUtil::NullHandleException&) {
    cout << "As expected, got a NullHandleException" << endl;
}
```

Copy constructor

The copy constructor ensures that you can construct a proxy handle from another proxy handle. Internally, this increments a reference count on the proxy; the destructor decrements the reference count again and, once the count drops to zero, deallocates the underlying proxy class instance. That way, memory leaks are avoided:

C++

```
{                               // Enter new scope
    SimplePrx s1 = ...;         // Get a proxy from somewhere
    SimplePrx s2(s1);           // Copy?construct s2
    assert(s1 == s2);           // Assertion passes
}                               // Leave scope; s1, s2, and the
                                // underlying proxy instance
                                // are deallocated
```

Note the assertion in this example: proxy handles support comparison.

Assignment operator

You can freely assign proxy handles to each other. The handle implementation ensures that the appropriate memory-management activities take place. Self-assignment is safe and you do not have to guard against it:

C++

```
SimplePrx s1 = ...;     // Get a proxy from somewhere
SimplePrx s2;           // s2 is nil
s2 = s1;                // both point at the same object
s1 = 0;                 // s1 is nil
s2 = 0;                 // s2 is nil
```

Widening assignments work implicitly. For example, if we have two interfaces, `Base` and `Derived`, we can widen a `DerivedPrx` to a `BasePrx` implicitly:

C++

```
BasePrx base;
DerivedPrx derived;
base = derived;         // Fine, no problem
derived = base;         // Compile?time error
```

Implicit narrowing conversions result in a compile error, so the usual C++ semantics are preserved: you can always assign a derived type to a base type, but not vice versa.

Checked cast

Proxy handles provide a `checkedCast` method:

C++

```
namespace IceInternal {
    template<typename T>
    class ProxyHandle : public IceUtil::HandleBase<T> {
    public:
        template<class Y>
        static ProxyHandle checkedCast(const ProxyHandle<Y>& r);

        template<class Y>
        static ProxyHandle checkedCast(const ProxyHandle<Y>& r, const ::Ice::Context& c);

        // ...
    };
}
```

A checked cast has the same function for proxies as a C++ `dynamic_cast` has for pointers: it allows you to assign a base proxy to a derived proxy. If the base proxy's actual run-time type is compatible with the derived proxy's static type, the assignment succeeds and, after the assignment, the derived proxy denotes the same object as the base proxy. Otherwise, if the base proxy's run-time type is incompatible with the derived proxy's static type, the derived proxy is set to null. Here is an example to illustrate this:

C++

```
BasePrx base = ...;     // Initialize base proxy
DerivedPrx derived;
derived = DerivedPrx::checkedCast(base);
if (derived) {
        // Base has run?time type Derived,
        // use derived...
} else {
        // Base has some other, unrelated type
}
```

The expression `DerivedPrx::checkedCast(base)` tests whether `base` points at an object of type `Derived` (or an object with a type that is derived from `Derived`). If so, the cast succeeds and `derived` is set to point at the same object as `base`. Otherwise, the cast fails and `derived` is set to the null proxy.

Note that `checkedCast` is a static member function so, to do a down-cast, you always use the syntax <*interface-name*>`Prx::checkedCast`.

Also note that you can use proxies in boolean contexts. For example, `if (proxy)` returns true if the proxy is not null.

A `checkedCast` typically results in a remote message to the server.The message effectively asks the server "is the object denoted by this reference of typeDerived?"

In some cases, the Ice run time can optimize the cast and avoid sending a message. However, the optimization applies only in narrowly-defined circumstances, so you cannot rely on a `checkedCast` not sending a message.

The reply from the server is communicated to the application code in form of a successful (non-null) or unsuccessful (null) result. Sending a remote message is necessary because, as a rule, there is no way for the client to find out what the actual run-time type of a proxy is without confirmation from the server. (For example, the server may replace the implementation of the object for an existing proxy with a more derived one.) This means that you have to be prepared for a `checkedCast` to fail. For example, if the server is not running, you will receive a `ConnectFailedException;` if the server is running, but the object denoted by the proxy no longer exists, you will receive an `ObjectNotExistException`.

Unchecked cast

In some cases, it is known that an object supports a more derived interface than the static type of its proxy. For such cases, you can use an unchecked down-cast:

C++

```
namespace IceInternal {
    template<typename T>
    class ProxyHandle : public IceUtil::HandleBase<T> {
    public:
        template<class Y>
        static ProxyHandle uncheckedCast(const ProxyHandle<Y>& r);
        // ...
  };
}
```

An `uncheckedCast` provides a down-cast *without* consulting the server as to the actual run-time type of the object, for example:

C++

```
BasePrx base = ...;     // Initialize to point at a Derived
DerivedPrx derived;
derived = DerivedPrx::uncheckedCast(base);
// Use derived...
```

You should use an `uncheckedCast` only if you are certain that the proxy indeed supports the more derived type: an `uncheckedCast`, as the name implies, is not checked in any way; it does not contact the object in the server and, if it fails, it does not return null. (An unchecked cast is implemented internally like a C++ `static_cast`, no checks of any kind are made). If you use the proxy resulting from an incorrect `uncheckedCast` to invoke an operation, the behavior is undefined. Most likely, you will receive an `OperationNotExistException`, but, depending on the circumstances, the Ice run time may also report an exception indicating that unmarshaling has failed, or even silently return garbage results.

Despite its dangers, `uncheckedCast` is still useful because it avoids the cost of sending a message to the server. And, particularly during initialization, it is common to receive a proxy of static type `Ice::Object`, but with a known run-time type. In such cases, an `uncheckedCast` saves the overhead of sending a remote message.

Stream insertion and stringification

For convenience, proxy handles also support insertion of a proxy into a stream, for example:

C++

```
Ice::ObjectPrx p = ...;
cout << p << endl;
```

This code is equivalent to writing:

C++

```
Ice::ObjectPrx p = ...;
cout << p->ice_toString() << endl;
```

Either code prints the stringified proxy. You could also achieve the same thing by writing:

C++

```
Ice::ObjectPrx p = ...;
cout << communicator->proxyToString(p) << endl;
```

The advantage of using the `ice_toString` member function instead of `proxyToString` is that you do not need to have the communicator available at the point of call.

Using Proxy Methods in C++

The base proxy class `ObjectPrx` supports a variety of methods for customizing a proxy. Since proxies are immutable, each of these "factory methods" returns a copy of the original proxy that contains the desired modification. For example, you can obtain a proxy configured with a ten second timeout as shown below:

C++

```
Ice::ObjectPrx proxy = communicator->stringToProxy(...);
proxy = proxy->ice_timeout(10000);
```

A factory method returns a new proxy object if the requested modification differs from the current proxy, otherwise it returns the current proxy. With few exceptions, factory methods return a proxy of the same type as the current proxy, therefore it is generally not necessary to repeat a down-cast after using a factory method. The example below demonstrates these semantics:

C++

```
Ice::ObjectPrx base = communicator->stringToProxy(...);
HelloPrx hello = HelloPrx::checkedCast(base);
hello = hello->ice_timeout(10000); // Type is preserved
hello->sayHello();
```

The only exceptions are the factory methods `ice_facet` and `ice_identity`. Calls to either of these methods may produce a proxy for an object of an unrelated type, therefore they return a base proxy that you must subsequently down-cast to an appropriate type.

Object Identity and Proxy Comparison in C++

Proxy handles support comparison using the following operators:

- `operator==`
  `operator!=`
  These operators permit you to compare proxies for equality and inequality. To test whether a proxy is null, use a comparison with the literal `0`, for example:

  C++

  ```
  if (proxy == 0)
      // It's a nil proxy
  else
      // It's a non?nil proxy
  ```

- `operator<`
  `operator<=`
  `operator>`
  `operator>=`
  Proxies support comparison. This allows you to place proxies into STL containers such as maps or sorted lists.

- Boolean comparison
  Proxies have a conversion operator to `bool`. The operator returns true if a proxy is not null, and false otherwise. This allows you to write:

  C++

  ```
  BasePrx base = ...;
  if (base)
          // It's a non?nil proxy
  else
          // It's a nil proxy
  ```

Note that proxy comparison uses *all* of the information in a proxy for the comparison. This means that not only the object identity must match for a comparison to succeed, but other details inside the proxy, such as the protocol and endpoint information, must be the same. In other words, comparison with == and != tests for *proxy* identity, *not* object identity. A common mistake is to write code along the following lines:

C++

```
Ice::ObjectPrx p1 = ...;        // Get a proxy...
Ice::ObjectPrx p2 = ...;        // Get another proxy...

if (p1 != p2) {
    // p1 and p2 denote different objects       // WRONG!
} else {
    // p1 and p2 denote the same object         // Correct
}
```

Even though `p1` and `p2` differ, they may denote the same Ice object. This can happen because, for example, both `p1` and `p2` embed the same object identity, but each use a different protocol to contact the target object. Similarly, the protocols may be the same, but denote different endpoints (because a single Ice object can be contacted via several different transport endpoints). In other words, if two proxies compare equal with ==, we know that the two proxies denote the same object (because they are identical in all respects); however, if two proxies compare unequal with ==, we know absolutely nothing: the proxies may or may not denote the same object.

To compare the object identities of two proxies, you can use helper functions in the `Ice` namespace:

C++

```
namespace Ice {

    bool proxyIdentityLess(const ObjectPrx&, const ObjectPrx&);
    bool proxyIdentityEqual(const ObjectPrx&, const ObjectPrx&);
    bool proxyIdentityAndFacetLess(const ObjectPrx&, const ObjectPrx&);
    bool proxyIdentityAndFacetEqual(const ObjectPrx&, const ObjectPrx&);

}
```

The `proxyIdentityEqual` function returns true if the object identities embedded in two proxies are the same and ignores other information in the proxies, such as facet and transport information. To include the facet name in the comparison, use `proxyIdentityAndFacetEqual` instead.

The `proxyIdentityLess` function establishes a total ordering on proxies. It is provided mainly so you can use object identity comparison with STL sorted containers. (The function uses `name` as the major ordering criterion, and `category` as the minor ordering criterion.) The `proxyIdentityAndFacetLess` function behaves similarly to `proxyIdentityLess`, except that it also compares the facet names of the proxies when their identities are equal.

`proxyIdentityEqual` and `proxyIdentityAndFacetLess` allow you to correctly compare proxies for object identity. The example below demonstrates how to use `proxyIdentityEqual`:

C++

```
Ice::ObjectPrx p1 = ...;        // Get a proxy...
Ice::ObjectPrx p2 = ...;        // Get another proxy...

if (!Ice::proxyIdentityEqual(p1, p2) {
    // p1 and p2 denote different objects       // Correct
} else {
    // p1 and p2 denote the same object         // Correct
}
```

See Also

- Interfaces, Operations, and Exceptions
- Proxies
- C++ Mapping for Operations
- Example of a File System Client in C++
- Using Proxies
- Facets and Versioning
- IceStorm