C++11 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 M
{
    class SimplePrx : public virtual Ice::ObjectPrx
    {
    public:
        void op(const Ice::Context& = Ice::noExplicitContext);
        ...
        static const std::string& ice_staticId();
    };
}
```

Your client code interacts directly with the *proxy class*, `M::SimplePrx` in the example above. More generally, the generated proxy class for an interface in module `M` is the C++ proxy class `M::<`*`interface-name`*`>Prx`.

In the client's address space, an instance of the proxy class 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`

All generated proxy classes inherit directly or indirectly from the `Ice::ObjectPrx` proxy class, reflecting the fact that all Slice interfaces implicitly inherit from `Ice::Object`.

Interface Inheritance

Inheritance relationships among Slice interfaces are maintained in the generated C++ classes. For example:

Slice

```
module M
{
    interface A { ... }
    interface B { ... }
    interface C extends A, B { ... }
}
```

The generated code for `CPrx` reflects the inheritance hierarchy:

C++

```
namespace M
{
    class CPrx : public virtual APrx, public virtual BPrx
    {
        ...
    };
}
```

Given a proxy for `C`, a client can invoke any operation defined for interface `C`, as well as any operation inherited from `C`'s base interfaces.

Receiving Proxies

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:

C++

```
M::SimplePrx 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 `std::shared_ptr<proxy class>`.

The client accesses the proxy via this `shared_ptr`; the `shared_ptr` 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 `shared_ptr` to the proxy disappears (goes out of scope).

Down-casting Proxies with `checkedCast` and `uncheckedCast`

The Ice namespaces provides two template functions, `checkedCast` and `uncheckedCast`, modeled after `std::dynamic_pointer_cast`:

C++

```
namespace Ice
{
    // Modeled after std::dynamic_pointer_cast. P is the derived proxy class, for example DerivedPrx.
    template<typename P, typename T, ...>
    std::shared_ptr<P> checkedCast(const std::shared_ptr<T>& b, const Ice::Context& context = Ice::
noExplicitContext)
    {
        ...
    }

    template<typename P, typename T, ...> std::shared_ptr<P> uncheckedCast(const std::shared_ptr<T>& b)
    {
        ...
    }
}
```

These functions allow you to down-cast a proxy to a more derived proxy type.

Checked cast

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 type of the base proxy's target object is compatible with the derived proxy's static type, the assignment succeeds and, after the assignment, the derived proxy denotes the same remote object as the base proxy. Otherwise, if the type of the base proxy's target object is incompatible with the derived proxy's static type, the derived proxy is set to null. Here is an example to illustrate this:

C++

```
std::shared_ptr<BasePrx> base = ...;     // Initialize base proxy
auto derived = Ice::checkedCast<DerivedPrx>(base); // returns a shared_ptr<DerivedPrx>
if(derived)
{
    // Base's target object has run-time type Derived,
    // use derived...
}
else
{
    // Base has some other, unrelated type
}
```

The expression `checkedCast<DerivedPrx>(base)` tests whether `base` points at a remote 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 null.

A `checkedCast` always results in a remote message, `ice_isA`, to the server. The message effectively asks the server "is the object denoted by this proxy of type Derived?". When `ice_isA` returns true, `checkedCast` manufactures and returns a new proxy instance.

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

C++

```
std::shared_ptr<BasePrx> base = ...;     // Initialize to point at a Derived
auto derived = Ice::uncheckedCast<DerivedPrx>(base);
// Use derived...
```

You should use an `uncheckedCast` only if you are certain that target object 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 never fails. 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.

Calling `uncheckedCast` on a proxy that is already of the desired proxy type returns immediately that proxy. Otherwise, `uncheckedCast` creates a new instance of the desired proxy class.

Despite its dangers, `uncheckedCast` is 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::ObjectPrx`, but with a known run-time type. In such cases, an `uncheckedCast` saves the overhead of sending a remote message.

Typed Proxy Factory 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 invocation timeout as shown below:

C++11

```
std::shared_ptr<Ice::ObjectPrx> proxy = communicator->stringToProxy(...);
proxy = proxy->ice_invocationTimeout(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, the corresponding C++ factory member functions return a proxy of the same type as the current proxy, therefore it is generally not necessary to down-cast after calling such a factory. The example below demonstrates these semantics:

C++

```
auto base = communicator->stringToProxy(...);
auto hello = checkedCast<HelloPrx>(base);
hello = hello->ice_invocationTimeout(10000); // Type is preserved
hello->sayHello();
```

The only exceptions are the factory member functions `ice_facet` and `ice_identity`. Calls to either of these functions 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++

You can compare proxies for equality. By default, proxy comparison compares all aspects of a proxy, including the object identity, facet name, addressing information, and all the proxy settings; two proxies compare equal only if they are identical in all respects. The mapping provides helper functions to simplify the comparison of proxies stored in `shared_ptr` values.

Note however that the more common use case is determining whether two proxies denote the same Ice object, in which case you should only be comparing their object identities. To compare the object identities of two proxies, you can use helper functions and classes in the `Ice` namespace:

C++

```
namespace Ice
{
    bool proxyIdentityLess(const std::shared_ptr<ObjectPrx>&, const std::shared_ptr<ObjectPrx>&);
    bool proxyIdentityEqual(const std::shared_ptr<ObjectPrx>&, const std::shared_ptr<ObjectPrx>&);
    bool proxyIdentityAndFacetLess(const std::shared_ptr<ObjectPrx>&, const std::shared_ptr<ObjectPrx>&);
    bool proxyIdentityAndFacetEqual(const std::shared_ptr<ObjectPrx>&, const std::shared_ptr<ObjectPrx>&);

    struct ProxyIdentityLess : std::binary_function<bool, std::shared_ptr<ObjectPrx>&, std::
shared_ptr<ObjectPrx>&>
    {
        bool operator()(const std::shared_ptr<ObjectPrx>& lhs, const std::shared_ptr<ObjectPrx>& rhs) const
        {
            return proxyIdentityLess(lhs, rhs);
        }
    };

    struct ProxyIdentityEqual ...
    struct ProxyIdentityAndFacetLess ...
    struct ProxyIdentityAndFacetEqual ...
}
```

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 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.

See Also

- Interfaces, Operations, and Exceptions
- Proxies for Ice Objects
- C++11 Mapping for Operations
- Example of a File System Client in C++11
- Versioning