C++ Mapping for Operations

On this page:

Basic C++ Mapping for Operations

As we saw in the C++ mapping for interfaces, for each operation on an interface, the proxy class contains a corresponding member function with the same name. To invoke an operation, you call it via the proxy handle. For example, here is part of the definitions for our file system:

Slice

```
module Filesystem {
    interface Node {
        idempotent string name();
    };
    // ...
};
```

The proxy class for the `Node` interface, tidied up to remove irrelevant detail, is as follows:

C++

```
namespace IceProxy {
    namespace Filesystem {
        class Node : virtual public IceProxy::Ice::Object {
        public:
            std::string name();
            // ...
        };
        typedef IceInternal::ProxyHandle<Node> NodePrx;
        // ...
    }
    // ...
}
```

The `name` operation returns a value of type `string`. Given a proxy to an object of type `Node`, the client can invoke the operation as follows:

C++

```
NodePrx node = ...;             // Initialize proxy
string name = node->name();     // Get name via RPC
```

The proxy handle overloads `operator->` to forward method calls to the underlying proxy class instance which, in turn, sends the operation invocation to the server, waits until the operation is complete, and then unmarshals the return value and returns it to the caller.

Because the return value is of type `string`, it is safe to ignore the return value. For example, the following code contains no memory leak:

C++

```
NodePrx node = ...;             // Initialize proxy
node->name();                   // Useless, but no leak
```

This is true for all mapped Slice types: you can safely ignore the return value of an operation, no matter what its type — return values are always returned by value. If you ignore the return value, no memory leak occurs because the destructor of the returned value takes care of deallocating memory as needed.

Normal and `idempotent` Operations in C++

You can add an `idempotent` qualifier to a Slice operation. As far as the signature for the corresponding proxy methods is concerned, `idempotent` has no effect. For example, consider the following interface:

Slice

```
interface Example {
              string op1();
    idempotent string op2();
    idempotent void op3(string s);
};
```

The proxy class for this interface looks like this:

C++

```
namespace IceProxy {
    class Example : virtual public IceProxy::Ice::Object {
    public:
        std::string op1();
        std::string op2();              // idempotent
        void op3(const std::string&);   // idempotent
        // ...
    };
}
```

Because `idempotent` affects an aspect of call dispatch, not interface, it makes sense for the mapping to be unaffected by the `idempotent` keyword.

Passing Parameters in C++

In-Parameters in C++

The parameter passing rules for the C++ mapping are very simple: parameters are passed either by value (for small values) or by `const` reference (for values that are larger than a machine word). Semantically, the two ways of passing parameters are identical: it is guaranteed that the value of a parameter will not be changed by the invocation (with some caveats — see Out-Parameters below and Location Transparency).

Here is an interface with operations that pass parameters of various types from client to server:

Slice

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ClientToServer {
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
};
```

The Slice compiler generates the following code for this definition:

C++

```
struct NumberAndString {
    Ice::Int x;
    std::string str;
    // ...
};

typedef std::vector<std::string> StringSeq;

typedef std::map<Ice::Long, StringSeq> StringTable;

namespace IceProxy {
    class ClientToServer : virtual public IceProxy::Ice::Object {
    public:
        void op1(Ice::Int, Ice::Float, bool, const std::string&);
        void op2(const NumberAndString&, const StringSeq&, const StringTable&);
        void op3(const ClientToServerPrx&);
        // ...
    };
}
```

Given a proxy to a `ClientToServer` interface, the client code can pass parameters as in the following example:

C++

```
ClientToServerPrx p = ...;              // Get proxy...

p->op1(42, 3.14, true, "Hello world!"); // Pass simple literals

int i = 42;
float f = 3.14;
bool b = true;
string s = "Hello world!";
p->op1(i, f, b, s);                     // Pass simple variables

NumberAndString ns = { 42, "The Answer" };
StringSeq ss;
ss.push_back("Hello world!");
StringTable st;
st[0] = ss;
p->op2(ns, ss, st);                     // Pass complex variables

p->op3(p);                              // Pass proxy
```

You can pass either literals or variables to the various operations. Because everything is passed by value or `const` reference, there are no memory-management issues to consider.

Out-Parameters in C++

The C++ mapping passes out-parameters by reference. Here is the Slice definition once more, modified to pass all parameters in the `out` direction:

Slice

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ServerToClient {
    void op1(out int i, out float f, out bool b, out string s);
    void op2(out NumberAndString ns, out StringSeq ss, out StringTable st);
    void op3(out ServerToClient* proxy);
};
```

The Slice compiler generates the following code for this definition:

C++

```
namespace IceProxy {
    class ServerToClient : virtual public IceProxy::Ice::Object {
    public:
       void op1(Ice::Int&, Ice::Float&, bool&, std::string&);
       void op2(NumberAndString&, StringSeq&, StringTable&);
       void op3(ServerToClientPrx&);
       // ...
    };
}
```

Given a proxy to a `ServerToClient` interface, the client code can pass parameters as in the following example:

C++

```
ServerToClientPrx p = ...;      // Get proxy...

int i;
float f;
bool b;
string s;

p->op1(i, f, b, s);
// i, f, b, and s contain updated values now

NumberAndString ns;
StringSeq ss;
StringTable st;

p->op2(ns, ss, st);
// ns, ss, and st contain updated values now

p->op3(p);
// p has changed now!
```

Again, there are no surprises in this code: the caller simply passes variables to an operation; once the operation completes, the values of those variables will be set by the server.

It is worth having another look at the final call:

C++

```
p->op3(p);      // Weird, but well?defined
```

Here, p is the proxy that is used to dispatch the call. That same variable p is also passed as an out-parameter to the call, meaning that the server will set its value. In general, passing the same parameter as both an input and output parameter is safe: the Ice run time will correctly handle all locking and memory-management activities.

Another, somewhat pathological example is the following:

Slice

```
sequence<int> Row;
sequence<Row> Matrix;

interface MatrixArithmetic {
    void multiply(Matrix m1, Matrix m2, out Matrix result);
};
```

Given a proxy to a `MatrixArithmetic` interface, the client code could do the following:

C++

```
MatrixArithmeticPrx ma = ...;      // Get proxy...
Matrix m1 = ...;                   // Initialize one matrix
Matrix m2 = ...;                   // Initialize second matrix
ma->squareAndCubeRoot(m1, m2, m1); // !!!
```

This code is technically legal, in the sense that no memory corruption or locking issues will arise, but it has surprising behavior: because the same variable `m1` is passed as an input parameter as well as an output parameter, the final value of `m1` is indeterminate — in particular, if client and server are collocated in the same address space, the implementation of the operation will overwrite parts of the input matrix `m1` in the process of computing the result because the result is written to the same physical memory location as one of the inputs. In general, you should take care when passing the same variable as both an input and output parameter and only do so if the called operation guarantees to be well-behaved in this case.

Chained Invocations in C++

Consider the following simple interface containing two operations, one to set a value and one to get it:

Slice

```
interface Name {
    string getName();
    void setName(string name);
};
```

Suppose we have two proxies to interfaces of type `Name`, `p1` and `p2`, and chain invocations as follows:

C++

```
p2->setName(p1->getName());
```

This works exactly as intended: the value returned by `p1` is transferred to `p2`. There are no memory-management or exception safety issues with this code.

Exception Handling in C++

Any operation invocation may throw a run-time exception and, if the operation has an exception specification, may also throw user exceptions. Suppose we have the following simple interface:

Slice

```
exception Tantrum {
    string reason;
};

interface Child {
    void askToCleanUp() throws Tantrum;
};
```

Slice exceptions are thrown as C++ exceptions, so you can simply enclose one or more operation invocations in a `try-catch` block:

C++

```
ChildPrx child = ...;           // Get proxy...
try {
    child->askToCleanUp();      // Give it a try...
} catch (const Tantrum& t) {
    cout << "The child says: " << t.reason << endl;
}
```

Typically, you will catch only a few exceptions of specific interest around an operation invocation; other exceptions, such as unexpected run-time errors, will typically be dealt with by exception handlers higher in the hierarchy. For example:

C++

```
void run()
{
    ChildPrx child = ...;       // Get proxy...
    try {
        child->askToCleanUp();  // Give it a try...
    } catch (const Tantrum& t) {
        cout << "The child says: " << t.reason << endl;

        child->scold();         // Recover from error...
    }
    child->praise();            // Give positive feedback...
}

int
main(int argc, char* argv[])
{
    int status = 1;
    try {
        // ...
        run();
        // ...
        status = 0;
    } catch (const Ice::Exception& e) {
        cerr << "Unexpected run?time error: " << e << endl;
    }
    // ...
    return status;
}
```

This code handles a specific exception of local interest at the point of call and deals with other exceptions generically. (This is also the strategy we used for our first simple application.)

For efficiency reasons, you should always catch exceptions by `const` reference. This permits the compiler to avoid calling the exception's copy constructor (and, of course, prevents the exception from being sliced to a base type).

Exceptions and Out-Parameters in C++

The Ice run time makes no guarantees about the state of out-parameters when an operation throws an exception: the parameter may have still have its original value or may have been changed by the operation's implementation in the target object. In other words, for out-parameters, Ice provides the weak exception guarantee [1] but does not provide the strong exception guarantee.

This is done for reasons of efficiency: providing the strong exception guarantee would require more overhead than can be justified.

Exceptions and Return Values in C++

For return values, C++ provides the guarantee that a variable receiving the return value of an operation will not be overwritten if an exception is thrown. (Of course, this guarantee holds only if you do not use the same variable as both an out-parameter and to receive the return value of an invocation).

See Also

- Operations
- Slice for a Simple File System
- C++ Mapping for Interfaces

References

1. Sutter, H. 1999. Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions. Reading, MA: Addison-Wesley.