Parameter Passing in C++11

Previous Next

For each parameter of a Slice operation, the C++ mapping generates a corresponding parameter for the virtual member function in the skeleton. In addition, every operation has a trailing parameter of type `const Ice::Current&`. For example, the `name` operation of the `Node` interface has no parameters, but the `name` member function of the `Node` skeleton class has a single parameter of type `const Ice::Current&`. We will ignore this parameter for now.

Parameter passing on the server side has the following rules:

- in-parameters are passed by value only
- out-parameters are passed by reference
- return values are passed by value
- optional parameters are enclosed in `Ice::optional` values

Compared to the client side rules, the only difference is for in-parameters: on the client side, they are mapped to value or const reference (depending on the parameter type), while on the server side they are always passed by value.

On the client side, you allocate these in-parameters and Ice only needs to read them, so const reference is fine for parameters like strings and vectors. On the server side, Ice allocates these parameters and then relinquishes them to your servant: getting these parameters by value allows your servant to adopt (move) them.

To illustrate the rules, consider the following interface that passes string parameters in all possible directions:

Slice

```
module M
{
    interface Example
    {
        string op(string sin, out string sout);
    }
}
```

The generated skeleton class for this interface looks as follows:

C++

```
namespace M
{
    class Example : public virtual Ice::Object
    {
    public:
        virtual std::string op(string, std::string&, const Ice::Current&) = 0;
        // ...
    };
}
```

As you can see, there are no surprises here. For example, we could implement `op` as follows:

C++

```
std::string
ExampleI::op(std::string sin, std::string& sout, const Ice::Current&)
{
    cout << sin << endl;        // In parameters are initialized
    sout = "Hello World!";      // Assign out parameter
    return "Done";              // Return a string
}
```

This code is in no way different from what you would normally write if you were to pass strings to and from a function; the fact that remote procedure calls are involved does not impact on your code in any way. The same is true for parameters of other types, such as proxies, classes, or dictionaries: the parameter passing conventions follow normal C++ rules and do not require special-purpose API calls or memory management.

Back to Top ^

Thread-Safe Marshaling in C++

The marshaling semantics of the Ice run time present a subtle thread safety issue that arises when an operation returns data by reference. For C++ applications, this can affect servant methods that return instances of Slice classes or types referencing Slice classes.

The potential for corruption occurs whenever a servant returns data by reference, yet continues to hold a reference to that data. For example, consider the following Slice:

Slice

```
sequence<sequence<int>> IntIntSeq;
sequence<string> StringSeq;
class Grid
{
    StringSeq xLabels;
    StringSeq yLabels;
    IntIntSeq values;
}

interface GridIntf
{
   Grid getGrid();
   void clearValues();
}
```

And the following servant implementation:

C++

```
class GridIntfI : public GridIntf
{
public:
    std::shared_ptr<Grid> getGrid(const Ice::Current&);
    void clear(const Ice::Current&);

private:
    std::mutex _mutex;
    std::shared_ptr<Grid> _grid;
};

std::shared_ptr<Grid>
GridIntfI::getGrid(const Ice::Current&)
{
    std::lock_guard<std::mutex> lock(_mutex);
    return _grid;
}

void
GridIntfI::clearValues(const Ice::Current&)
{
    std::lock_guard<std::mutex> lock(_mutex);
    _grid->values.clear();
}
```

Suppose that a client invoked the `getGrid` operation. While the Ice run time marshals the returned class in preparation to send a reply message, it is possible for another thread to dispatch the `clearValues` operation on the same servant. This race condition can result in several unexpected outcomes, including a failure during marshaling or inconsistent data in the reply to `getGrid`. Synchronizing the `getGrid` and `setValue` operations does not fix the race condition because the Ice run time performs its marshaling outside of this synchronization.

Solution 1: Copying

One solution is to implement accessor operations, such as `getGrid`, so that they return copies of any data that might change. There are several drawbacks to this approach:

- Excessive copying can have an adverse affect on performance.
- The operations must return deep copies in order to avoid similar problems with nested values.
- The code to create deep copies is tedious and error-prone to write.

Solution 2: Copy on Write

Another solution is to make copies of the affected data only when it is modified. In the revised code shown below, `clearValues` replaces `_grid` with a copy that contains empty values, leaving the previous contents of `_grid` unchanged:

C++

```
void GridIntfI::clearValues(const Ice::Current&)
{
    std::lock_guard<std::mutex> lock(_mutex);
    shared_ptr<Grid> grid = make_shared<Grid>();
    grid->xLabels = _grid->xLabels;
    grid->yLabels = _grid->yLabels;
    _grid = grid;
}
```

This allows the Ice run time to safely marshal the return value of `getGrid` because the `values` array is never modified again. For applications where data is read more often than it is written, this solution is more efficient than the previous one because accessor operations do not need to make copies. Furthermore, intelligent use of shallow copying can minimize the overhead in mutating operations.

Solution 3: Marshal Immediately

Finally, a third approach is to modify the servant mapping using metadata in order to force the marshaling to occur immediately within your synchronization. Annotating a Slice operation with the `marshaled-result` metadata directive changes the signature of the corresponding servant method, if that operation returns mutable types. The metadata directive has the following effects:

- For an operation `op` that returns one or multiple values and at least one of those values has a mutable type, the Slice compiler generates an `OpMarshaledResult` class and the return type of the servant method becomes `OpMarshaledResult`.
- The constructor for `OpMarshaledResult` takes an extra argument of type `Current`. The servant must supply the `Current` in order for the results to be marshaled correctly.

The metadata directive has no effect on the proxy mapping, nor does it affect the servant mapping of Slice operations that return `void` or return only immutable values.

You can also annotate an interface with the `marshaled-result` metadata and it will be applied to all of the interface's operations.

After applying the metadata, we can now implement the `Grid` servant as follows:

C++

```
GetGridMarshaledResult
GridIntfI::getGrid(const Ice::Current& currrent)
{
    return GetGridMarshaledResult(_grid, curr); // _grid is marshaled immediately
}
```

Here are more examples to demonstrate the mapping:

Slice

```
class C { ... }
struct S { ... }
sequence<string> Seq;

interface Example
{
    C getC();

    ["marshaled-result"]
    C getC2();

    void getS(out S val);

    ["marshaled-result"]
    void getS2(out S val);

    string getValues(string name, out Seq val);

    ["marshaled-result"]
    string getValues2(string name, out Seq val);
}
```

Review the generated code below to see the changes that the presence of the metadata causes in the servant method signatures:

C++

```
class Example : public virtual Ice::Object
{
public:
    class GetCMarshaledResult : public Ice::MarshaledResult
    {
    public:
        GetCMarshaledResult(const std::shared_ptr<C>&, const Ice::Current&);
    };

    class GetS2MarshaledResult : public Ice::MarshaledResult
    {
    public:
        GetS2MarshaledResult(const S&, const Ice::Current&);
    };

    class GetValues2MarshaledResult : public Ice::MarshaledResult
    {
    public:
        GetValues2MarshaledResult(const std::string&, const Seq&, const Ice::Current&);
    };

    virtual std::shared_ptr<C> getC(const Ice::Current&) = 0;
    virtual GetC2MarshaledResult getC2(const Ice::Current&) = 0;
    virtual S getS(const Ice::Current&) = 0;
    virtual GetS2MarshaledResult getS2(const Ice::Current&) = 0;
    virtual std::string getValues(std::string, Seq&, const Ice::Current&) = 0;
    virtual GetValues2MarshaledResult getValues2(std::string, const Ice::Current&) = 0;
};
```

Back to Top ^

See Also

- Server-Side C++11 Mapping for Interfaces
- C++11 Mapping for Operations
- C++11 Mapping for Optional Values
- Raising Exceptions in C++11
- The Current Object

Previous

Next