Parameter Passing in Objective-C

This page shows how to implement parameters for Slice operations in Objective-C.

On this page:

Implementing Parameters for Slice Operations in Objective-C

For each parameter of a Slice operation, the Objective-C mapping generates a corresponding parameter for the method in the skeleton. In addition, every method has an additional, trailing parameter of type `ICECurrent`. For example, the `name` operation of the `Node` interface has no parameters, but the `name` method of the `Node` skeleton protocol has a single parameter of type `ICECurrent`. We will ignore this parameter for now.

Parameter passing on the server side follows the rules for the client side (with one exception):

- In-parameters and the return value are passed by value or by pointer, depending on the parameter type.
- Out-parameters are passed by pointer-to-pointer.

The exception to the client-side rules concerns types that come in mutable and immutable variants (strings, sequences, and dictionaries). For these, the server-side mapping passes the mutable variant where the client-side passes the immutable variant, and vice versa.

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

Slice

```
interface Intf
{
    string op(string sin, out string sout);
}
```

The generated skeleton protocol for this interface looks as follows:

Objective-C

```
@protocol EXIntf <ICEObject>
-(NSString *) op:(NSMutableString *)sin
                 sout:(NSString **)sout
                 current:(ICECurrent *)current;
@end
```

As you can see, the in-parameter `sin` is of type `NSMutableString`, and the out parameter and return value are passed as `NSString` (the opposite of the client-side mapping). This means that in-parameters are passed to the servant as their mutable variant, and it is safe for you to modify such in-parameters. This is useful, for example, if a client passes a sequence to the operation, and the operation returns the sequence with a few minor changes. In that case, there is no need for the operation implementation to copy the sequence. Instead, you can simply modify the passed sequence as necessary and return the modified sequence to the client.

Here is an example implementation of the operation:

Objective-C

```
-(NSString *) op:(NSMutableString *)sin
                 sout:(NSString **)sout
                 current:(ICECurrent *)current
{
    printf("%s\n", [sin UTF8String]); // In-params are initialized
    *sout = [sin appendString:@"appended"]; // Assign out-param
    return @"Done";                          // Return a string
}
```

Memory Management for Operations in Objective-C

If you are not using ARC, to avoid leaking memory, you must be aware of how the Ice run time manages memory for operation implementations:

- In-parameters passed to the servant are already autoreleased.
- Out-parameters and return values must be returned by the servant as autoreleased values.

This follows the usual Objective-C convention: the allocator of a value is responsible for releasing it. This is what the Ice run time does for in-parameters, and what you are expected to do for out-parameters and return values. These rules also mean that it is safe to return an in-parameter as an out-parameter or return value. For example:

Objective-C

```
-(NSString *) op:(NSMutableString *)sin
                 sout:(NSString **)sout
                 current:(ICECurrent *)current
{
    *sout = sin; // Works fine.
    return sin;  // Works fine.
}
```

The Ice run time creates and releases a separate autorelease pool for each invocation. This means that the memory for parameters is reclaimed as soon as the run time has marshaled the operation results back to the client.

Thread-Safe Marshaling in Objective-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 Objective-C applications, this can affect servant methods that return instances of Slice classes, structures, sequences, or dictionaries.

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

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

And the following servant implementation:

Objective-C

```
-(Grid*) getGrid:(ICECurrent *)current
{
    Grid* r;
    @synchronized(self)
    {
        r = grid;
    }
    return r;
}

-(void) clearValues:(ICECurrent *)current
{
    @synchronized(self)
    {
                if([grid.values isKindOfClass:[NSMutableArray class]])
        {
            [(NSMutableArray*)grid.values removeAllObjects];
        }
        else
        {
            grid.values = [MutableIntIntSeq array];
        }
    }
}
```

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 `clearValues` operatio ns 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:

Objective-C

```
-(void) clearValues:(ICECurrent *)current
{
    @synchronized(self)
    {
        grid.values = [MutableIntIntSeq array];
    }
}
```

This allows the Ice run time to safely marshal the return value of `getGrid` because its members are 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.

See Also

- Server-Side Objective-C Mapping for Interfaces
- Raising Exceptions in Objective-C
- The Current Object