Data Encoding for Class Graphs

On this page:

Encoding a Class Graph

Classes support pointer semantics, that is, you can construct graphs of classes. It follows that classes can arbitrarily point at each other. The class identity is used to distinguish instances and pointers as follows:

- A class identity of 0 denotes a null pointer.
- A class identity > 0 precedes the marshaled contents of an instance
- A class identity < 0 denotes a pointer to an instance.

Identity values less than zero are pointers. For example, if the receiver receives the identity -57, this means that the corresponding class member that is currently being unmarshaled will eventually point at the instance with identity 57.

For structures, classes, exceptions, sequences, and dictionary members that do not contain class members, the Ice encoding uses a simple depth-first traversal algorithm to marshal the members. For example, structure members are marshaled in the order of their Slice definition; if a structure member itself is of complex type, such as a sequence, the sequence is marshaled in toto where it appears inside its enclosing structure. For complex types that contain class members, this depth-first marshaling is suspended: instead of marshaling the actual class instance at this point, a negative identity is marshaled that indicates which class instance that member must eventually denote. For example, consider the following definitions:

Slice

```
class C {
    // ...
};

struct S {
    int i;
    C firstC;
    C secondC;
    C thirdC;
    int j;
};
```

Suppose we initialize a structure of type S as follows:

C++

```
S myS;
myS.i = 99;
myS.firstC = new C;             // New instance
myS.secondC = 0;                // null
myS.thirdC = myS.firstC;        // Same instance as previously
myS.j = 100;
```

When this structure is marshaled, the contents of the three class members are not marshaled in-line. Instead, the sender marshals the negative identities of the corresponding instances. Assuming that the sender has assigned the identity 78 to the instance assigned to `myS.firstC`, `myS` is marshaled as shown in the table.

Marshaled Value	Size in Bytes	Type	Byte offset
`99` *(myS.i)*	`4`	`int`	`0`
`-78` *(myS.firstC)*	`4`	`int`	`4`
`0` *(myS.secondC)*	`4`	`int`	`8`
`-78` *(mys.thirdC)*	`4`	`int`	`12`

100 *(myS.j)*	4	int	16

Note that `myS.firstC` and `myS.thirdC` both use the identity `-78`. This allows the receiver to recognize that `firstC` and `thirdC` point at the same class instance (rather than at two different instances that happen to have the same contents).

Marshaling the negative identities instead of the contents of an instance allows the receiver to accurately reconstruct the class graph that was sent by the sender. However, this begs the question of *when* the actual instances are to be marshaled as described at the beginning of this section. In Ice prot ocol messages, parameters and return values are marshaled as if they were members of a structure. For example, if an operation invocation has five input parameters, the client marshals the five parameters end-to-end as if they were members of a single structure. If any of the five parameters are class instances, or are of complex type (recursively) containing class instances, the sender marshals the parameters in multiple passes: the first pass marshals the parameters end-to-end, using the usual depth-first algorithm:

- If the sender encounters a class member during marshaling, it checks whether it has marshaled the same instance previously for the current request or reply:
  - If the instance has not been marshaled before, the sender assigns a new identity to the instance and marshals the negative identity.
  - Otherwise, if the instance was marshaled previously, the sender sends the same negative identity that is previously sent for that instance.

In effect, during marshaling, the sender builds an identity table that is indexed by the address of each instance; the lookup value for the instance is its identity.

Once the first pass ends, the sender has marshaled all the parameters, but has not yet marshaled any of the class instances that may be pointed at by various parameters or members. The identity table at this point contains all those instances for which negative identities (pointers) were marshaled, so whatever is in the identity table at this point are the classes that the receiver still needs. The sender now marshals those instances in the identity table, but with positive identities and followed by their contents, as described in our earlier example. The outstanding instances are marshaled as a sequence, that is, the sender marshals the number of instances as a size, followed by the actual instances.

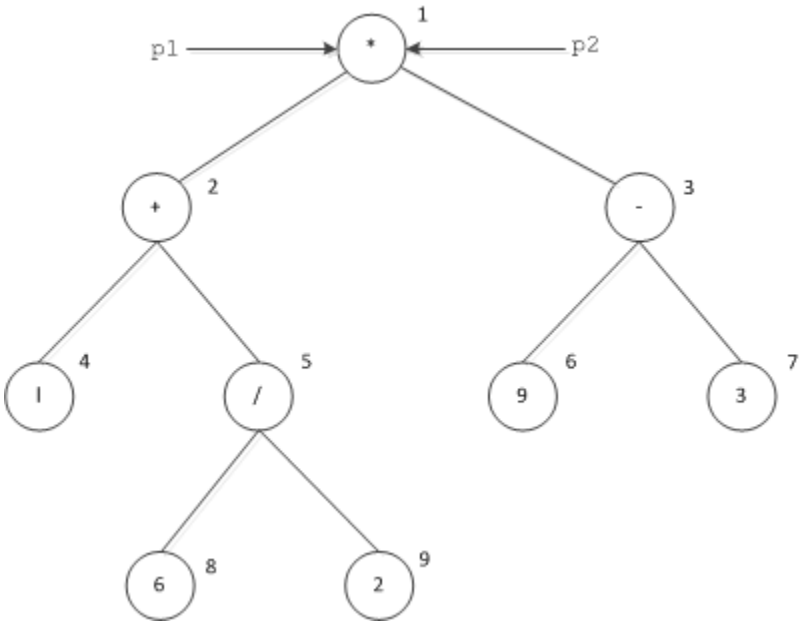
In turn, the instances just sent may themselves contain class members; when those class members are marshaled, the sender assigns an identity to new instances or uses a negative identity for previously marshaled instances as usual. This means that, by the end of the second pass, the identity table may have grown, necessitating a third pass. That third pass again marshals the outstanding class instances as a size followed by the actual instances. The third pass contains all those instances that were not marshaled in the second pass. Of course, the third pass may trigger yet more passes until, finally, the sender has sent all outstanding instances, that is, marshaling is complete. At this point, the sender terminates the sequence of passes by marshaling an empty sequence (the value `0` encoded as a size).

To illustrate this with an example, consider the definitions shown in Classes with Operations once more:

Slice

```
enum UnaryOp { UnaryPlus, UnaryMinus, Not };
enum BinaryOp { Plus, Minus, Multiply, Divide, And, Or };

class Node {
    idempotent long eval();
};

class UnaryOperator extends Node {
    UnaryOp operator;
    Node operand;
};

class BinaryOperator extends Node {
    BinaryOp op;
    Node operand1;
    Node operand2;
};

class Operand {
    long val;
};
```

These definitions allow us to construct expression trees. Suppose the client initializes a tree to the shape shown in the illustration below, representing the expression `(1 + 6 / 2) * (9 - 3)`. The values outside the nodes are the identities assigned by the client.



Expression tree for the expression `(1 + 6 / 2) * (9 - 3)`. Both `p1` and `p2` denote the root node.

The client passes the root of the tree to the following operation in the parameters `p1` and `p2`, as shown in the illustration above. (Even though it does not make sense to pass the same parameter value twice, we do it here for illustration purposes):

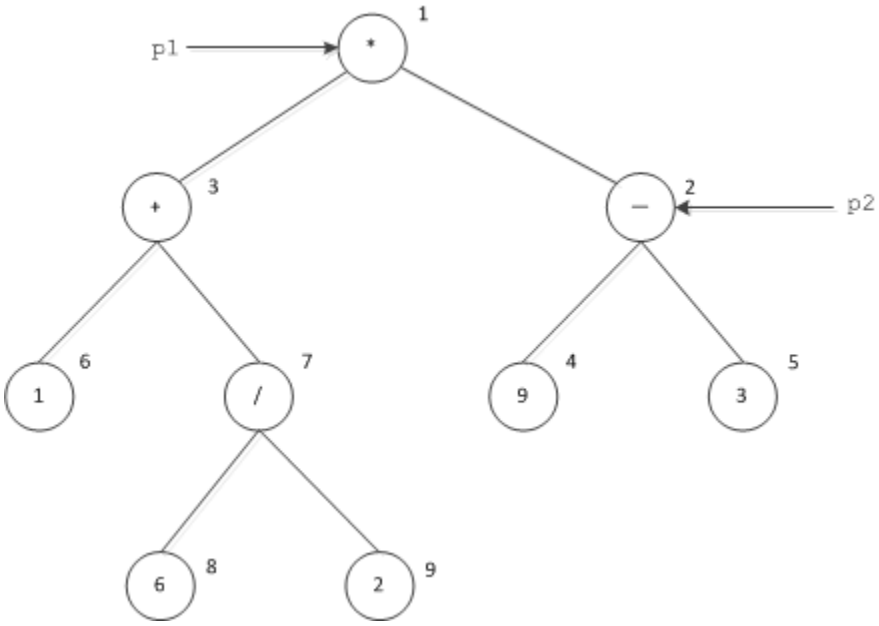
Slice

```
interface Tree {
    void sendTree(Node p1, Node p2);
};
```

The client now marshals the two parameters `p1` and `p2` to the server, resulting in the value `-1` being sent twice in succession. (The client arbitrarily assigns an identity to each node. The value of the identity does not matter, as long as each node has a unique identity. For simplicity, the Ice implementation numbers instances with a counter that starts counting at `1` and increments by one for each unique instance.) This completes the marshaling of the parameters and results in a single instance with identity `1` in the identity table. The client now marshals a sequence containing a single element, node 1, as described in the example. In turn, node 1 results in nodes 2 and 3 being added to the identity table, so the next sequence of nodes contains two elements, nodes 2 and 3. The next sequence of nodes contains nodes 4, 5, 6, and 7, followed by another sequence containing nodes 8 and 9. At this point, no more class instances are outstanding, and the client marshals an empty sequence to indicate to the receiver that the final sequence has been marshaled.

Within each sequence, the order in which class instances are marshaled is irrelevant. For example, the third sequence could equally contain nodes 7, 6, 4, and 5, in that order. What is important here is that each sequence contains nodes that are an equal number of "hops" away from the initial node: the first sequence contains the initial node(s), the second sequence contains all nodes that can be reached by traversing a single link from the initial node(s), the third sequence contains all nodes that can be reached by traversing two links from the initial node(s), and so on.

Now consider the same example once more, but with different parameter values for `sendTree`: `p1` denotes the root of the tree, and `p2` denotes the `-` operator of the right-hand sub-tree, as shown:



The expression tree of with p1 and p2 denoting different nodes.

The graph that is marshaled is exactly the same, but instances are marshaled in a different order and with different identities:

- During the first pass, the client sends the identities `-1` and `-2` for the parameter values.
- The second pass marshals a sequence containing nodes 1 and 2.
- The third pass marshals a sequence containing nodes 3, 4, and 5.
- The fourth pass marshals a sequence containing nodes 6 and 7.
- The fifth pass marshals a sequence containing nodes 8 and 9.
- The final pass marshals an empty sequence.

In this way, any graph of nodes can be transmitted (including graphs that contain cycles). The receiver reconstructs the graph by filling in a patch table during unmarshaling:

- Whenever the receiver unmarshals a negative identity, it adds that identity to a patch table; the lookup value is the memory address of the parameter or member that eventually will point at the corresponding instance.
- Whenever the receiver unmarshals an actual instance, it adds the instance to an unmarshaled table; the lookup value is the memory address of the instantiated class. The receiver then uses the address of the instance to patch any parameters or members with the actual memory address.

Note that the receiver may receive negative identities that denote class instances that have been unmarshaled already (that is, point "backward" in the unmarshaling stream), as well as instances that are yet to be unmarshaled (that is, point "forward" in the unmarshaling stream). Both scenarios are possible, depending on the order in which instances are marshaled, as well as their in-degree.

To provide another example, consider the following definition:

Slice

```
class C {
    // ...
};

sequence<C> CSeq;
```

Suppose the client marshals a sequence of 100 `C` instances to the server, with each instance being distinct. (That is, the sequence contains 100 pointers to 100 different instances, not 100 pointers to the same single instance.) In that case, the sequence is marshaled as a size of 100, followed by 100 negative identities, `-1` to `-100`. Following that, the client marshals a single sequence containing the 100 instances, each instance with its positive identity in the range `1` to `100`, and completes by marshaling an empty sequence.

On the other hand, if the client sends a sequence of 100 elements that all point to the same single class instance, the client marshals the sequence as a size of 100, followed by 100 negative identities, all with the value `-1`. The client then marshals a sequence containing a single element, namely instance `1`, and completes by marshaling an empty sequence.

Impact of Slicing on Class Graph Decoding

It is important to note that when a graph of class instances is sent, it always forms a connected graph. However, when the receiver rebuilds the graph, it may end up with a disconnected graph, due to slicing. Consider:

Slice

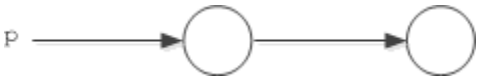
```
class Base {
    // ...
};

class Derived extends Base {
    // ...
    Base b;
};

interface Example {
    void op(Base p);
};
```

Suppose the client has complete type knowledge, that is, understands both types `Base` and `Derived`, but the server only understands type `Base`, so the derived part of a `Derived` instance is sliced. The client can instantiate classes to be sent as parameter `p` as follows:

C++

```
DerivedPtr p = new Derived;
p->b = new Derived;
ExamplePrx e = ...;
e->op(p);
```

As far as the client is concerned, the graph looks like the one shown below:



Sender-side view of a graph containing derived instances.

However, the server does not understand the derived part of the instances and slices them. Yet, the server unmarshals all the class instances, leading to the situation where the class graph has become disconnected, as shown here:

Receiver-side view of the graph.

Of course, more complex situations are possible, such that the receiver ends up with multiple disconnected graphs, each containing many instances.

See Also

- Classes with Operations
- Basic Data Encoding
- Data Encoding for Classes
- Simple Example of Class Encoding
- Protocol Messages