Data Encoding for Class Graphs

Classes support pointer semantics, that is, you can construct graphs of classes. It follows that classes can arbitrarily point at each other, and therefore the encoding must provide a scheme for serializing and deserializing a class graph. Note that the marshaling format for class references and instances differs significantly between versions 1.0 and 1.1 of the encoding.

On this page:

- Encoding a Class Graph version 1.0
- Encoding a Class Graph version 1.1
  - Class Graph in the Compact Format
  - Class Graph in the Sliced Format
  - Importance of the Indirection Table
- Impact of Slicing on Class Graph Decoding

Encoding a Class Graph version 1.0

In version 1.0 of the encoding, an instance ID is used to distinguish instances and pointers as follows:

- An instance ID of 0 denotes a null pointer.
- An instance ID > 0 precedes the marshaled contents of an instance.
- An instance ID < 0 denotes a pointer to an instance.

Instance ID values less than zero are pointers. For example, if the receiver receives the instance ID `-57`, this means that the corresponding class member that is currently being unmarshaled will eventually point at the instance with ID `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 instance ID 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 instance IDs of the corresponding instances. Assuming that the sender has assigned the instance ID `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 instance ID `-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 instance IDs 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 protocol 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 instance ID to the instance and marshals the negative ID.
  - Otherwise, if the instance was marshaled previously, the sender sends the same negative ID that it previously sent for that instance.

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

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 instance table at this point contains all those instances for which negative IDs (pointers) were marshaled, so whatever is in the table at this point are the classes that the receiver still needs. The sender now marshals those instances in the table, but with positive IDs 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.

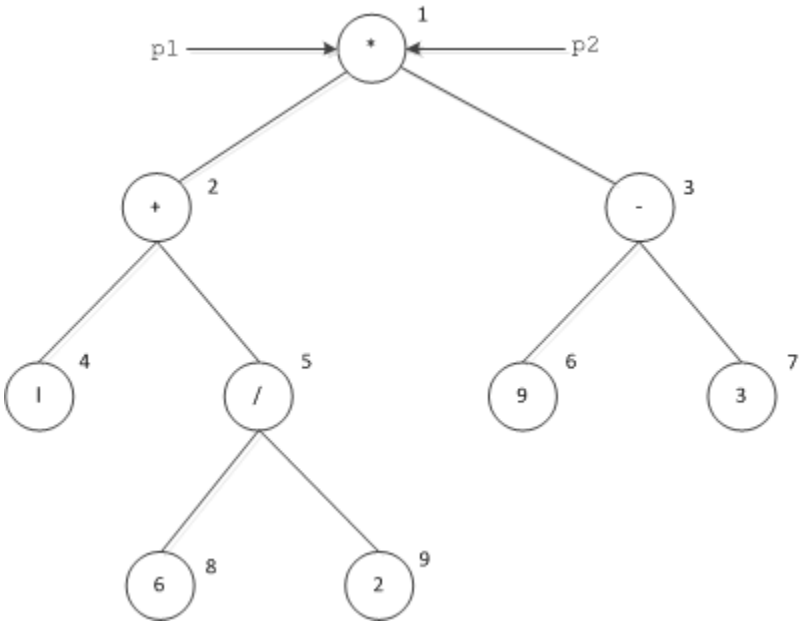
In turn, the instances just sent may themselves contain class members; when those class members are marshaled, the sender assigns IDs to new instances or uses a negative ID for previously marshaled instances as usual. This means that, by the end of the second pass, the table may have grown, necessitating 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):

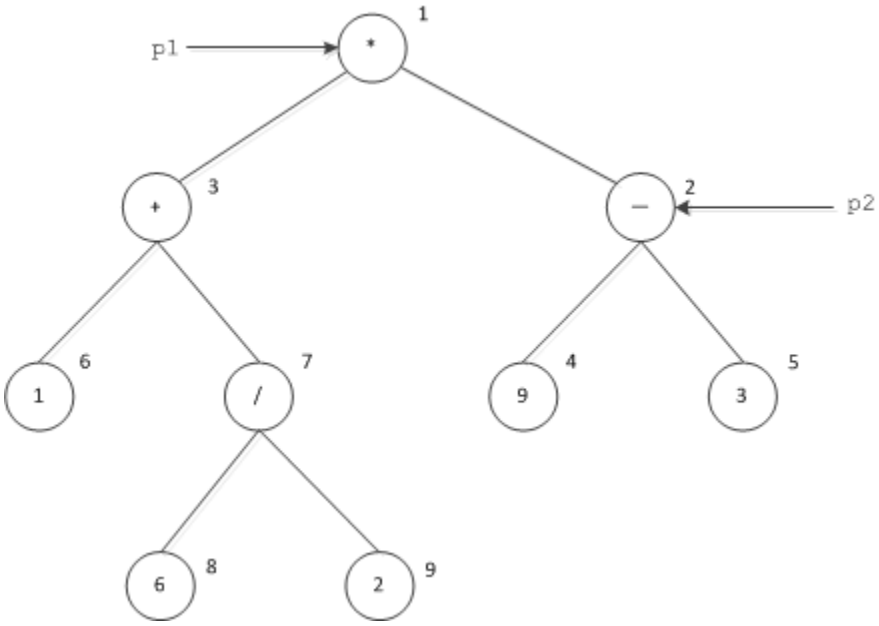
Slice

```
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 instance ID to each node. The value of the ID does not matter, as long as each node has a unique ID. 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 ID 1 in the instance 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 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 below:



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 IDs:

- During the first pass, the client sends the IDs `-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 ID, it adds that ID 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 IDs 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 IDs, `-1` to `-100`. Following that, the client marshals a single sequence containing the 100 instances, each instance with its positive ID 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 IDs, 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.

Encoding a Class Graph version 1.1

The most significant difference in the class encoding between version 1.0 and 1.1 is the location of class instances in the output stream. In version 1.0, instances are always marshaled at the end of the encapsulation, whereas in version 1.1 it is possible for instances to be marshaled *inline* at the point of first reference. Our previous discussion of class references describes the factors that determine whether a given class reference is marshaled as a reference or as an inline instance. The encoding rules can be summarized as follows:

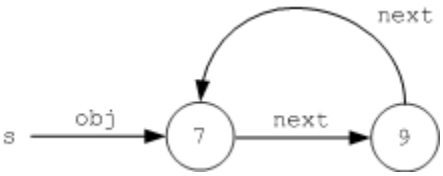
- When using the compact format, always marshal an instance inline at the point of its first reference, and marshal all subsequent occurrences of the same instance as a reference.
- When using the sliced format, the encoding depends on the context in which a reference occurs: if the reference occurs while encoding a slice of an object or exception, encode it as an index into an indirection table that appears at the end of the slice, otherwise encode it as an inline instance or reference.

We can use the Slice definitions below to explore these situations:

Slice

```
class Node {
    int value;
    Node next;
};

struct S {
    Node obj;
};
```

Suppose we create an instance of structure `S` and assign it to the variable `s`, then construct the following class graph:



Class graph with circular reference.

We discuss the encoding for these values in separate sections below.

Class Graph in the Compact Format

As we marshal the structure's `obj` member, the compact format produces the following encoding:

Marshaled value	Size in bytes	Type	Byte offset
1 (*`obj` - inline instance marker - assigned ID 2*)	1	`size`	0
33 (*slice flags: string type ID, last slice*)	1	`byte`	1
`"::Node"` (*type ID - assigned index 1*)	7	`string`	2
7 (*`value`*)	4	`int`	9
1 (*`next` - inline instance marker - assigned ID 3*)	1	`size`	13
34 (*slice flags: type ID index, last slice*)	1	`byte`	14
1 (*type ID index*)	1	`size`	15
9 (*`value`*)	4	`int`	16
2 (*`next` - reference to ID 2*)	1	`size`	20

This example shows that inline instances are marshaled immediately upon discovery. As we encode the `obj` member, we encounter a new `Node` instance, assign it the ID 2, and begin to encode it inline using the reserved instance marker value 1. Next, we encode the instance's `value` member and then examine its `next` member. Since we have not encountered this instance yet, we assign it the ID 3 and begin to encode this new instance inline. Its `next` member is a circular reference to the object with ID 2, so we simply encode a reference to this instance to complete the structure.

Class Graph in the Sliced Format

The need to support both slicing and slice preservation makes the sliced format more complex than the compact format. The encoding for our example values shows how class instances are handled differently depending on their context:

Marshaled value	Size in bytes	Type	Byte offset
`1` (*`obj` - inline instance marker - assigned ID 2*)	1	`size`	0
`57` (*slice flags: string type ID, size is present, indirection table is present, last slice*)	1	`byte`	1
`"::Node"` (*type ID - assigned index `1`*)	7	`string`	2
`9` (*byte count for slice*)	4	`int`	9
`7` (*`value`*)	4	`int`	13
`1` (*`next` - indirection table index*)	1	`size`	17
`1` (*size of indirection table*)	1	`size`	18
`1` (*indirection table entry - inline instance marker - assigned ID 3*)	1	`size`	19
`58` (*slice flags: type ID index, size is present, indirection table is present, last slice*)	1	`byte`	20
`1` (*type ID index*)	1	`size`	21
`9` (*byte count for slice*)	4	`int`	22
`9` (*`value`*)	4	`int`	26
`1` (*`next` - indirection table index - ID 2*)	1	`size`	30
`1` (*size of indirection table*)	1	`size`	31
`2` (*indirection table entry - reference to ID 2*)	1	`size`	32

For this example, let us assume that the structure value is an operation parameter, and therefore the structure member `obj` occurs *outside* the context of a class or exception slice. As a result, we encode the instance with ID `2` as an inline instance. However, when we process the instance's `next` member, we are now *inside* the context of a class slice, so we must use an indirection table. The sender adds an entry to the indirection table for each unique class reference in the slice; instead of encoding a reference or inline instance, the sender encodes the index of the corresponding entry in the table. As shown above, the value encoded for the `next` member is `1`, representing the first entry in the table (`0` is still reserved for nil references).

The indirection table follows immediately after the last data member in the slice, but the table is *not* included in the byte count for the slice. A leading size denotes the number of entries in the table. The one and only entry in this table, the instance with ID `3`, has not yet been encoded, therefore it is marshaled immediately as an inline instance. This instance also uses an indirection table, although in this case the table entry is simply a reference to instance ID `2`, which has already been encoded.

Importance of the Indirection Table

The indirection table is necessary for implementing the slice preservation feature. Normally, when a receiver does not recognize the type ID in a slice, it has the option of ignoring that slice by skipping ahead in the stream by the number of bytes in the slice. However, when slice preservation is enabled, the receiver must keep a copy of the slice data in case the instance is later remarshaled. The need for the indirection table becomes apparent when you consider that an opaque blob of slice data may contain class references, and those class references can change during remarshaling. For example, without an indirection table, the sender might encode the instance ID `3` as the value of member `next`, but what happens if the receiver assigns that instance a different ID, such as `12`, when it remarshals the preserved slice? The receiver preserved the slice because it did not understand the type ID, which means it does not know the contents of the slice data and therefore it cannot "patch" any class references the slice might contain. The indirection table serves as an external "patch table" to solve this problem, essentially making the opaque slice data *relocatable* with respect to class references.

The byte count for a slice does not include the indirection table because the receiver must process the table regardless of whether it recognizes that slice's type ID. Consequently, to "skip" a slice, the receiver can skip (or preserve) the number of bytes specified by the slice's byte count, but still must decode the indirection table if the slice flags indicate that a table is present. If the receiver preserves the slice, it must also associate an indirection table with that slice; during remarshaling, the sender copies the opaque slice data into the stream, and then reconstructs the indirection table using (potentially) new instance IDs for the instances referenced in the table.

It is possible that the *only* reference to an instance is in an indirection table. To properly implement slice preservation, a receiver must therefore retain *every* instance that is referenced by a preserved indirection table. This is true even if the receiver does not recognize any of the type IDs in an instance; in effect, the receiver must construct a temporary "unknown object" placeholder for the instance, whose only purpose is to encapsulate the data comprising its slices in case the instance is later remarshaled.

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.

The slice preservation feature in version 1.1 of the encoding allows a receiver to remarshal the original graph intact, despite the fact that the receiver's in-memory object graph may appear to be disconnected.

See Also

- Classes with Operations
- Basic Data Encoding
- Data Encoding for Classes
- Simple Example of Class Encoding
- Protocol Messages
- Understanding Objects and Exceptions