Data Encoding for Classes

The marshaling for classes is complex, due to the need to deal with the pointer semantics for graphs of classes, as well as the need for the receiver to slice or preserve classes of unknown derived type. In addition, the marshaling for classes uses a type ID compression scheme to avoid repeatedly marshaling the same type IDs for large graphs of class instances.

Encoding for Class References

Consider the following Slice definitions:

Slice

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

struct S {
    Node obj;
};
```

We call the `obj` member a *reference* to a class instance. References can appear as data members, as in the case of `obj` above, or nested inside of other types, such as a sequence element or dictionary value. There are significant differences in the encoding for references between versions 1.0 and 1.1.

Class Reference Encoding version 1.0

Ice encodes a nil reference as a 32-bit integer with value zero. For non-nil references, the encoder maintains a table per encapsulation that associates a unique non-zero positive integer with each class instance. We refer to this integer as an *instance ID*. The first time the encoder encounters a reference to a particular instance, it assigns the instance an unused ID, inserts it into the table, and encodes the reference as a 32-bit integer whose value is the *negative* of the ID. If the encoder has already encountered that instance, it simply encodes the instance's previously-assigned ID as a negative 32-bit integer.

All class instances are encoded at the end of the encapsulation.

Class Reference Encoding version 1.1

As in version 1.0 of the encoding, each instance is assigned a unique ID per encapsulation. However, to conserve space, version 1.1 encodes a reference as a size value, with a nil reference encoded as a size value of zero and a non-nil reference encoded as a positive size value. The encoding reserves ID value zero to denote a nil reference, and ID value 1 to denote an *inline* instance in which an instance's state is marshaled at the point of its first reference. Consequently, instance IDs must start at 2. The encoding assigns instance IDs sequentially in order of appearance, with the first instance assigned an ID of 2, the next instance has ID 3, and so on.

The encoding for a reference depends on the format being used, and may also depend on the context in which the reference occurs:

- **Compact format**
  In this format, the encoding always uses the inline scheme. Suppose we are encoding a value of the structure type S shown earlier. If its `obj` member refers to a class instance that the encoder has not yet encountered in the current encapsulation, the encoding assigns it the next available sequential ID, marshals a size with value 1 signifying that an instance is about to be written, and then immediately marshals the instance itself. Any subsequent reference to the same instance is encoded as a size whose value is the instance's corresponding ID.

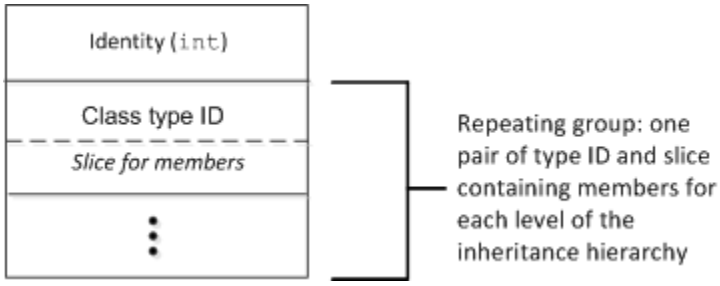
- **Sliced format**
  When a reference occurs *outside* the context of a class instance, Ice uses the same inline scheme as in the compact format. For example, suppose a Slice operation declared a parameter of structure type S shown above. When marshaling this parameter, its `obj` member is encoded as a size with value 1, followed by the encoding of the `Node` instance itself. Now consider the marshaling of the `Node` instance, where we encounter a class reference in member `next` that occurs *inside* the context of a class instance. To assist the Ice run time in unmarshaling and remarshaling instances, such a reference is encoded as a size whose value is an index into an indirection table.

Encoding for Class Instances

It is important to understand the distinction between marshaling a reference and marshaling an instance. In C++ terms, this is equivalent to the difference between a pointer and the heap data containing an object's state at which the pointer is pointing. Marshaling an instance means we are encoding the data members of a class instance.

Class Instance Encoding version 1.0

Classes are marshaled as a number of pairs containing a type ID and a slice (one pair for each level of the inheritance hierarchy) and marshaled in derived-to-base order. Only data members are marshaled — no information is sent that would relate to operations. Each marshaled class instance is preceded by a (non-zero) positive integer that provides an identity for the instance. The sender assigns this identity during marshaling such that each marshaled instance has a different identity. The receiver uses that identity to correctly reconstruct graphs of classes. The overall marshaling format for classes is shown below:



Marshaling format for classes.

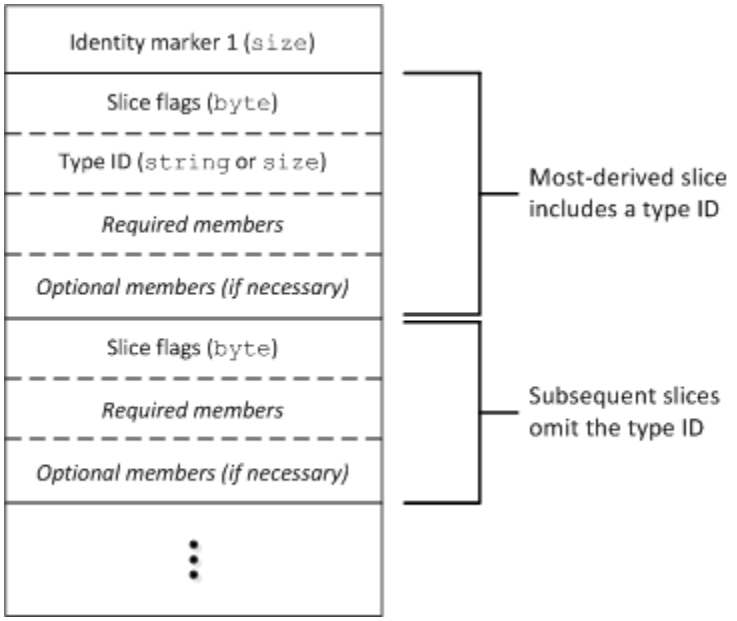
Prior to ending an encapsulation, the sender must encode all instances referenced within that encapsulation in an object table. The encoding may require multiple passes: as the sender processes an instance in the current pass, it may encounter additional instances to be processed in a subsequent pass. For each pass, the sender encodes a size denoting the number of objects in the pass. To signify that all instances have been encoded, the sender must marshal a size value of zero.

This object table must be written if any operation parameter or data member has a class type, even if all references are nil.

Class Instance Encoding version 1.1

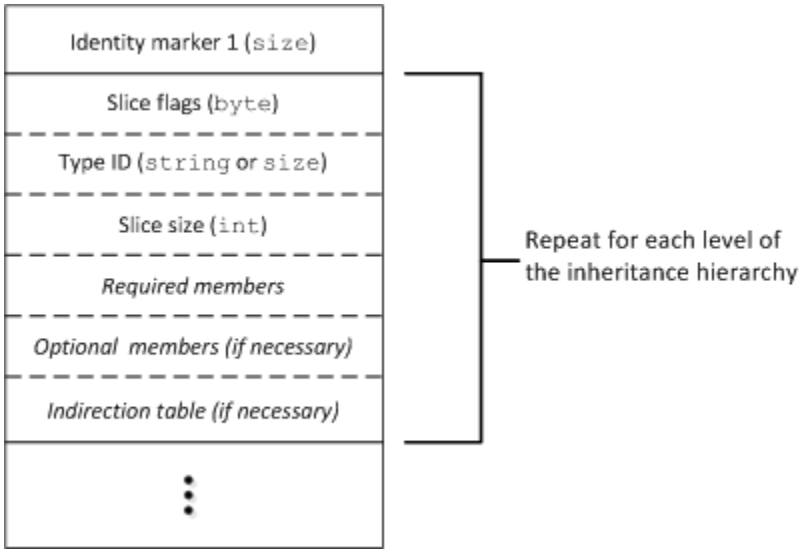
The leading byte of a class instance is a size value of 1. Following this byte is a collection of slices arranged in derived-to-base order. Only data members are marshaled — no information is sent that would relate to operations. The initial (most-derived) slice always includes a type ID that may be encoded as a string or as a numeric value, as specified by the flags that begin each slice. Depending on the format being used, subsequent slices may or may not include a type ID.

Each slice consists of a leading byte representing the slice flags, an optional type ID, an optional slice size, the required members for that slice in order of declaration, and the optional members of that slice. The sender must set the appropriate slice flags to indicate whether the slice includes a type ID, size, and optional data members, and whether this is the last slice of the instance. The compact format only includes a type ID in the initial (most-derived) slice and omits the slice size, as shown in the following diagram:



Compact format for classes.

When using the sliced format, a type ID is included in every slice, along with a slice size:



Sliced format for classes.

A slice that contains at least one non-nil class reference must use an indirection table for all references in that slice. The table is an array of instances encoded using the inline scheme. A leading size value indicates the number of elements in the table; each element is either an instance ID (if the instance has already been encoded within the current encapsulation), or the instance itself denoted by a leading size of 1. The indirection table appears in the encoding immediately following the required and optional data members, *but is not included in the byte count denoted by the slice size*. To skip a slice for an unknown type, the receiver can advance the input stream by the number of bytes specified in the slice size, but then must process all references or instances in the indirection table, if one is present.

To support slice preservation for an instance, the receiver must temporarily retain the slices of any unknown derived types, and also be able to reconstruct the indirection table in its original order for each of these slices in case the instance is later remarshaled.

See Also

- Classes
- Type IDs
- Data Encoding for Exceptions