Sequences

On this page:

Sequence Syntax and Semantics

Sequences are variable-length collections of elements:

Slice

```
sequence<Fruit> FruitPlatter;
```

A sequence can be empty—that is, it can contain no elements, or it can hold any number of elements up to the memory limits of your platform.

Sequences can contain elements that are themselves sequences. This arrangement allows you to create lists of lists:

Slice

```
sequence<FruitPlatter> FruitBanquet;
```

Sequences are used to model a variety of collections, such as vectors, lists, queues, sets, bags, or trees. (It is up to the application to decide whether or not order is important; by discarding order, a sequence serves as a set or bag.)

Using Sequences for Optional Values

Using a sequence to model an optional value is unnecessary with the introduction of optional data members and optional parameters in Ice 3.5.

One particular use of sequences has become idiomatic, namely, the use of a sequence to indicate an optional value. For example, we might have a `Part` structure that records the details of the parts that go into a car. The structure could record things such as the name of the part, a description, weight, price, and other details. Spare parts commonly have a serial number, which we can model as a `long` value. However, some parts, such as simple screws, often do not have a serial number, so what are we supposed to put into the serial number field of a screw? There are a number of options for dealing with this situation:

- **Use a sentinel value, such as zero, to indicate the "no serial number" condition.**
  This approach is workable, provided that a sentinel value is actually available. While it may seem unlikely that anyone would use a serial number of zero for a part, it is not impossible. And, for other values, such as a temperature value, all values in the range of their type can be legal, so no sentinel value is available.
- **Change the type of the serial number from `long` to `string`.**
  Strings come with their own built-in sentinel value, namely the empty string, so we can use an empty string to indicate the "no serial number" case. This is workable but not ideal: we should not have to change the natural data type of something to `string` just so we get a sentinel value.
- **Add an indicator as to whether the contents of the serial number are valid:**

Slice

```
struct Part {
    string name;
    string description;
    // ...
    bool   serialIsValid;  // true if part has serial number
    long   serialNumber;
};
```

This is guaranteed to get you into trouble eventually: sooner or later, some programmer will forget to check whether the serial number is valid before using it and create havoc.

- **Use a sequence to model the optional field.**
  This technique uses the following convention:

  Slice

  ```
  sequence<long> SerialOpt;

  struct Part {
      string    name;
      string    description;
      // ...
      SerialOpt serialNumber; // optional: zero or one element
  };
  ```

  By convention, the `Opt` suffix is used to indicate that the sequence is used to model an optional value. If the sequence is empty, the value is obviously not there; if it contains a single element, that element is the value. The obvious drawback of this scheme is that someone could put more than one element into the sequence. This could be rectified by adding a special-purpose Slice construct for optional values. However, optional values are not used frequently enough to justify the complexity of adding a dedicated language feature. (As we will see in Classes, you can also use class hierarchies to model optional fields.)

See Also

- Enumerations
- Structures
- Dictionaries
- Constants and Literals
- Classes