Container Classes

The Sequence Class

sequence vs bag

What’s the difference?

- the sequence class stores a collection of items, just like a bag...
- however, the sequence has an inherent order that is apparent via its public interface

The bag may have been stored sequentially in an array...

- this was an implementation detail and irrelevant to anyone using the class
- it presents itself as an unordered collection and provides such methods as appropriate

The sequence:

- will be kept in a specific order, and this will be exposed to the user of the sequence
- will expose methods (providing an internal iterator) that allow the user to iterate over its items

The Sequence Class—Specification

Type definitions and member constants:

```
    // data type of items in the sequence
    typedef _________ value_type;

    // value_type must be a built-in type, or support:
    //   - instantiation via a default constructor
    //   - instantiation via a copy constructor
    //   - assignment operator   (x = y)
```

The Sequence Class—Specification

Type definitions and member constants:

```
// data type of variables that track a sequence's size
typedef _________ size_type;

// the max number of items a sequence can hold
static const size_type CAPACITY = ___;
```

The Sequence Class—Specification

Constructors:

```
// creates an empty sequence
sequence();

// postcondition:
//   the sequence has been initialized as empty
```

The Sequence Class—Specification

Value semantics:

```
// sequence objects may be:
//   assigned using operator =
//   copied via the copy constructor
```

The Sequence Class—Specification

Modification member functions:

```
// the first item in the sequence is set to current
void start();

// postcondition:
//   the first item in the sequence becomes the current
//     item
//   if the sequence is empty, then there is no current
//    item
```

The Sequence Class—Specification

Modification member functions:

```
// advances the current item by one
void advance();


// precondition:
//   is_item() is true
// postcondition:
//   if the current item was already the last in the
//     sequence, then there is no longer a current item
//   Otherwise, the new item is the item immediately
//     after the previous current item
```

The Sequence Class—Specification

Modification member functions:

```
// adds @entry to the sequence before the current item
void insert(const value_type& entry);

// precondition:
//   size() < CAPACITY
// postcondition:
//   A new copy of entry has been inserted in the
//     sequence before the current item.
//   If there was no current item, then the new entry
//     has been inserted at the front of the sequence.
//   The new item is now the current item
```

The Sequence Class—Specification

Modification member functions:

```
// adds @entry to the sequence after the current item
void attach(const value_type& entry);


// precondition:
//   size() < CAPACITY
// postcondition:
//   A new copy of entry has been inserted in the
//     sequence after the current item.
//   If there was no current item, then the new entry
//     has been attached to the end of the sequence.
//   The new item is now the current item
```

The Sequence Class—Specification

Modification member functions:

```
    // removes the current item from the sequence
    void remove_current();

    // precondition:
    //   is_item() returns true
    // postcondition:
    //   The current item has been removed from the sequence
    //   The item after the removed element (if there is
    //     one) is the new current item
```

The Sequence Class—Specification

Constant member functions:

```
    // returns the total number of items in the sequence
    size_type size() const;

    // postcondition:
    //   return value is the number of items in the sequence
```

The Sequence Class—Specification

Constant member functions:

```
// returns true if the current element is valid
bool is_item() const;

// postcondition:
//   A true return value indicates that there is a valid
//     "current" item that may be retrieved by the
//     current member function
//   A false return value indicates that there is no
//     valid current item
```

The Sequence Class—Specification

Constant member functions:

```
// returns the current item
value_type current() const;

// precondition:
//    is_item() returns true
// postcondition:
//    The returned item is the current item in the
//    sequence
```

Examining the Sequence

The sequence class has several methods for examining itself in order:

```
// the first item in the sequence is set to current
void start();

// returns the current item
value_type current() const;

// advances the current item by one
void advance();

// returns true if the current element is valid
bool is_item() const;
```

Examining the Sequence

The sequence class has several methods for examining itself in order:

- these methods work together to enforce the in-order retrieval of items

Assume that `numbers` contains `37, 10, 83, and 42`:

```
// prints the first three items in order
numbers.start();    // beginning
cout << numbers.current() << endl; // outputs 37
numbers.advance(); // next item
cout << numbers.current() << endl; // outputs 10
numbers.advance(); // next item
cout << numbers.current() << endl; // outputs 83
```

Examining the Sequence

The sequence class has several methods for examining itself in order:

- these methods work together to enforce the in-order retrieval of items

Remember the precondition for `current()`:

```
// precondition: is_item() returns true
value_type current() const;
```

The `is_item()` function checks if the current item is valid

```
// only access the current item if it is valid
if (numbers.is_item()) {
    cout << numbers.current() << endl;
}
```

Examining the Sequence

The sequence class has several methods for examining itself in order:

- these methods work together to enforce the in-order retrieval of items

We can use these four functions to loop over a sequence:

```
// print each item in the sequence
for (nums.start(); nums.is_item(); nums.advance()) {
    cout << nums.current() << endl;
}
```

These functions provide what is called an internal iterator

- internal iterators are member functions that are used to access items in a collection
- this differs from external iterators (which are widely used by the standard library)

Modifying the Sequence

The sequence class also has methods to add/remove items:

```
// adds @entry to the sequence before the current item
void insert(const value_type& entry);

// adds @entry to the sequence after the current item
void attach(const value_type& entry);

// removes the current item from the sequence
void remove_current();
```

Modifying the Sequence

Assume we have the following sequence declared:

```
// an empty sequence that holds integers

sequence nums;
```

How would you use the modification methods to:

- add the values: `0, 10, 20, 30, ..., 80, 90`
- remove all items but the zero,
- insert `100` before the zero and `200` after it?