Member Function Implementations Of a Layered Component

Queue Layered on List
Implementation #2

Implementation #2 of Queue layered on List

- Has 2 internal contracts:
  1. The *correspondence*
  2. A *representation invariant*

Recall:

- The *correspondence*:

    - Defines how the abstract value of the client's declared variable can be obtained from the internal data member, i.e., from the concrete internal representation

- *internal contract*:

    - The correspondence is a contract that has meaning only within the component, therefore it is known as an *internal contract*

    - When is this internal contract used?

    1. When an exported operation is called:

        That operation can *assume* the correspondence holds

    1. When the exported operation terminates:

        It must *guarantee* that the correspondence holds

The *representation invariant*:

- Is also an internal contract:
    - It places *constraints* on the values stored by component's data members, i.e., on the concrete internal representation

The internal contracts for Queue layered on List:

- *correspondence*:

```
self = s.left * s.right
```

- *representation invariant*:

```
s.left = < >
```

  That is, all of the data stored in the data member ‘s’ is constrained to be stored in s.right, and s.left will remain empty

- Important note about an *internal contract*:

  An internal contract *does not have to hold* during the execution of the exported operation, only at the outset when it is called, and at the point where it terminates

```
// Filename: Queue2.hpp
#pragma once
#include "List\List1.hpp"

template <class T>
class Queue2
   //! is modeled by string(T)
   //!   exemplar self
{
public:
   // Standard Operations
   Queue2();
      //! alters self
      //! ensures: self = < >
   ~Queue2();
   void clear(void);
      //! alters self
      //! ensures: self = < >
   void transferFrom(Queue2& source);
      //! alters self, source
      //! ensures: self = #source  and  source = < >
   Queue2& operator =(Queue2& rhs);
      //! alters self
      //! preserves rhs
      //! ensures: self = rhs

   // Queue2 Specific Operations
   void enqueue(T& x);
      //! alters self
      //! consumes x
      //! ensures: self = #self * <#x>
   void dequeue(T& x);
      //! alters self
      //! produces x
      //! requires: self /= < >
      //! ensures: <x> is prefix of #self  and
      //!          self = #self[1, |#self|)
   void replaceFront(T& x);
      //! alters self, x
      //! requires: self /= < >
      //!  ensures: <x> is prefix of #self  and
      //!           self = <#x> * #self[1, |#self|)
   Integer length(void);
      //! preserves self
      //! ensures: length = |self|
   T& front(void);
      //! preserves self
      //! requires: self /= < >
      //!  ensures: <front> is prefix of self

private: // Internal Representation

   // Create instance of List
   typedef List1<T> ListOfT;

   // Data member of Queue2 - stores all data enqueued
      ListOfT s;
   // correspondence: self = s.left * s.right
   // representation invariant: s.left = < >
};
```

Implementation #2 – Internal Contracts

- The *correspondence*:

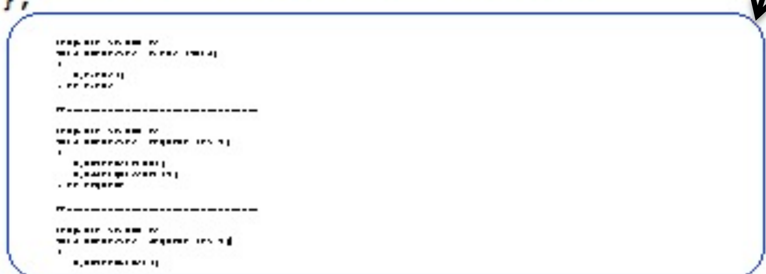
  `self = s.left * s.right`

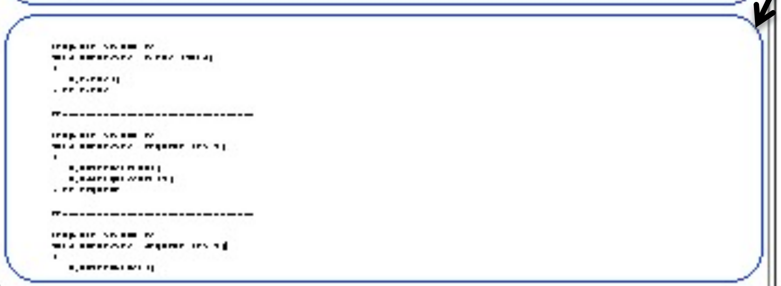
- The *representation invariant*:

  `s.left = < >`

```
// Filename: Queue2.hpp
#pragma once
#include "List\List1.hpp"

template <class T>
class Queue2
{
public:
   // Standard Operations
   Queue2();
   ~Queue2();
   void clear(void);
   void transferFrom(Queue2& source);
   Queue2& operator =(Queue2& rhs);
   // Queue2 Specific Operations
   void enqueue(T& x);
   void dequeue(T& x);
   void replaceFront(T& x);
   Integer length(void);
   T& front(void);
private: // Internal Representation
   typedef List1<T> ListOfT;
   ListOfT s;
// correspondence: self = s.left * s.right
// representation invariant: s.left = < >
};
```





Reexamine the Member Function Implementations

- Standard Operations Part
  - These operations remain the same as they are for Implementation #1 – i.e., when where there was no representation invariant
  - This is not always the case – sometimes the standard operations do need to take into account the representation invariant, but not for this example
- Component Specific Operations Part
  - These operations need to take into account the representation invariant of `s.left = <>`

```
// Filename: Queue2.hpp
#pragma once
#include "List\List1.hpp"

template <class T>
class Queue2
{
  // Queue2 Specific Operations
  void enqueue(T& x);
  void dequeue(T& x);
  void replaceFront(T& x);
  Integer length(void);
  T& front(void);
private: // Internal Representation
   typedef List1<T> ListOfT;
   ListOfT s;
// correspondence: self = s.left * s.right
// representation invariant: s.left = < >
};

template <class T>
void enqueue(T& x)
{
   s.clear();
} // enqueue

template <class T>
void dequeue(T& x)
{
   s.clear();
} // dequeue

template <class T>
void replaceFront(T& x)
{
   s.clear();
} // replaceFront

template <class T>
Integer length(void)
{
   s.clear();
} // clear

template <class T>
T& front(void)
{
   s.clear();
} // dequeue
```

Component Specific Operations

- Queue's component specific operations are:

  1. enqueue
  2. dequeue
  3. replaceFront
  1. length
  2. front

- We will modify the implementations of these operations so that they take into account both the *correspondence* and the *representation invariant*

- What do we mean by *take into account* the representation invariant?
  1. The implementation of each operation will assume the representation invariant holds when the operation is called
  2. And each implementation will guarantee that it holds when it terminates

enqueue

enqueue from Implementation #1

```
template <class T>
void enqueue(T& x)
   //! alters self
   //! consumes x
   //! ensures: self = #self * <#x>
{
   s.moveToFinish();
   s.addRightFront(x);
} // enqueue
```

enqueue from Implementation #2

```
template <class T>
void enqueue(T& x)
   //! alters self
   //! consumes x
   //! ensures: self = #self * <#x>
{
   s.moveToFinish();
   s.addRightFront(x);
   s.moveToStart();
} // enqueue
```

```
{
   // Example client of Queue2
   typedef Queue2<Integer> IntegerQueue;
   IntegerQueue q1;
   Integer y2;

   // Incoming: q1 = <18,15,27>  y2 = 5
   q1.enqueue(y2);
   // Outgoing: q1 = <18,15,27,5>  y2 = 0
}
```

- enqueue must still do the original two steps:
  1. move the fence to the finish
  2. add the item
- Recall that the the internal contract *must hold* at the termination of all exported operations
- So we must add a call to List's moveToStart in order to reestablish the representation invariant of `s.left = < >`
- Notice that more code is required for enqueue

  However, we will see for the other 4 Queue specific operations, less code will be required

dequeue

dequeue from Implementation #1

```
template <class T>
void dequeue(T& x)
   //! alters self
   //! produces x
   //! requires: self /= <#x>
   //!  ensures: <x> is prefix of #self  and
   //!           self = #self[1, |#self|)
{
   s.moveToStart();
   s.removeRightFront(x);
} // dequeue
```

dequeue from Implementation #2

```
template <class T>
void dequeue(T& x)
   //! alters self
   //! produces x
   //! requires: self /= <#x>
   //!  ensures: <x> is prefix of #self  and
   //!           self = #self[1, |#self|)
{
   s.removeRightFront(x);
} // dequeue
```

```
{
   // Example client of Queue2
   typedef Queue2<Integer> IntegerQueue;
   IntegerQueue q1;
   Integer y2;

   // Incoming: q1 = <18,15,27>  y2 = 5
   q1.dequeue(y2);
   // Outgoing: q1 = <15,27>  y2 = 18
}
```

- *Without* a representation invariant, s's fence is permitted to be anywhere, for incoming q1, it could be in 4 different locations:
  1. `s = < > <18,15,27>`
  2. `s = <18> <15,27>`
  3. `s = <18,15> <27>`
  4. `s = <18,15,27> < >`
- This is why we needed the `s.moveToStart();` found in dequeue in Implementation #1
- With the representation invariant of `s.left = < >`, s's fence is guaranteed to be at the start:
  `s = < > <18,15,27>`
- For Implementation #2, we can eliminate the call to moveToStart and simply call removeRightFront

replaceFront

replaceFront from Implementation #1

```
template <class T>
void replaceFront(T& x)
   //! alters self, x
   //! requires: self /= < >
   //!  ensures: <x> is prefix of #self  and
   //!           self = <#x> * #self[1, |#self|)
{
   s.moveToStart();
   s.replaceRightFront(x);
} // replaceFront
```

- Similar to dequeue, replaceFront works with the item at the front of the queue.
- `s.left = < >` is our representation invariant

  So the implementer of replaceFront can assume the s's fence is at the start
- Therefore, we can eliminate the call to List's moveToStart, and simply call List's replaceRightFront
- Here is the code for Queue's replaceFront in Implementation #2

replaceFront from Implementation #2

```
template <class T>
void replaceFront(T& x)
   //! alters self, x
   //! requires: self /= < >
   //!  ensures: <x> is prefix of #self  and
   //!           self = <#x> * #self[1, |#self|)
{
   s.replaceRightFront(x);
} // replaceFront
```

length

length from Implementation #1

```
template <class T>
Integer length(void)
   //! preserves self
   //! ensures: length = |self|
{
   return s.leftLength() + s.rightLength();
} // length
```

length from Implementation #2

```
template <class T>
Integer length(void)
   //! preserves self
   //! ensures: length = |self|
{
   return s.rightLength();
} // length
```

- `s.left = < >` is our representation invariant

  So the implementer of length can assume the s's fence is at the start, and the call to moveToStart is not needed

- So the implementation of Queue's length can just call List's rightLength to determine how many items are in the queue

front

front from Implementation #1

```
template <class T>
T& front(void)
   //! preserves self
   //! requires: self /= < >
   //!  ensures: <front> is prefix of self
{
   s.moveToStart();
   return s.rightFront();
} // front
```

front from Implementation #2

```
template <class T>
T& front(void)
   //! preserves self
   //! requires: self /= < >
   //!  ensures: <front> is prefix of self
{
   return s.rightFront();
} // front
```

- `s.left = < >` is our representation invariant

  So the implementer of length can assume the s's fence is at the start, and the call to moveToStart is not needed

- So the implementation of Queue's front can just call List's rightFront to access the item at the front of s.right