Member Function Implementations Of a Layered Component

Queue Layered on StaticArray Implementation #2

Illustrating the *correspondence*

```
self = IteratedConcatenation (
       z:Integer 0 <= z < currentLength, <contents(z)>
    )
```

```
// Filename: BoundedQueue1.hpp
#pragma once
#include "StaticArray\StaticArray1.hpp"

template <class T, int maxLength>
class BoundedQueue1
{
public:
   // Standard Operations
   BoundedQueue1();
   ~BoundedQueue1();
   void clear(void);
   void transferFrom(BoundedQueue1& source);
   BoundedQueue1& operator =(BoundedQueue1& rhs);

   // BoundedQueue1 Specific Operations
   void enqueue(T& x);
   void dequeue(T& x);
   void replaceFront(T& x);
   Integer length(void);
      //! preserves self
      //! ensures: length = |self|
   T& front(void);
   Integer remainingCapacity(void);

private: // internal representation
   enum {lowerBound = 0, upperBound = (maxLength - 1)};
   typedef StaticArray1 <T, lowerBound, upperBound> ArrayOfT;

   ArrayOfT contents;
   Integer currentLength;

   // correspondence self = IteratedConcatenation(
   //       z:Integer 0 <= z < currentLength, <contents(z)>)
   // convention 0 <= currentLength < maxLength
};
```

A Bounded Queue Implementation

- When the client program creates an instance of BoundedQueue it must supply a 2nd template parameter that is an integer which specifies the extent of the Queue

- Below is an example of a template instantiation of BoundedQueue where the extent of the bounded Queue is 10:

```
typedef BoundedQueue1 <Ingeter, 10> IntegerQueue;
```

```
// Filename: BoundedQueue1.hpp
#pragma once
#include "StaticArray\StaticArray1.hpp"

template <class T, int maxLength>
class BoundedQueue1
{
public:
   // Standard Operations
   BoundedQueue1();
   ~BoundedQueue1();
   void clear(void);
   void transferFrom(BoundedQueue1& source);
   BoundedQueue1& operator =(BoundedQueue1& rhs);

   // BoundedQueue1 Specific Operations
   void enqueue(T& x);
   void dequeue(T& x);
   void replaceFront(T& x);
   Integer length(void);
      //! preserves self
      //! ensures: length = |self|
   T& front(void);
   Integer remainingCapacity(void);

private: // internal representation
   enum {lowerBound = 0, upperBound = (maxLength - 1)};
   typedef StaticArray1 <T, lowerBound, upperBound> ArrayOfT;

   ArrayOfT contents;
   Integer currentLength;

   // correspondence self = IteratedConcatenation(
   //       z:Integer 0 <= z < currentLength, <contents(z)>)
   // convention 0 <= currentLength < maxLength
};
```

A Bounded Queue Implementation

- This 2nd actual parameter is passed into the template through the formal template parameter named *maxLength* (in this example)

```
typedef BoundedQueue1 <Ingeter, 10> IntegerQueue;
```

```
// Filename: BoundedQueue1.hpp
#pragma once
#include "StaticArray\StaticArray1.hpp"

template <class T, int maxLength>
class BoundedQueue1
{
public:
   // Standard Operations
   BoundedQueue1();
   ~BoundedQueue1();
   void clear(void);
   void transferFrom(BoundedQueue1& source);
   BoundedQueue1& operator =(BoundedQueue1& rhs);

   // BoundedQueue1 Specific Operations
   void enqueue(T& x);
   void dequeue(T& x);
   void replaceFront(T& x);
   Integer length(void);
      //! preserves self
      //! ensures: length = |self|
   T& front(void);
   Integer remainingCapacity(void);

private: // internal representation
   enum {lowerBound = 0, upperBound = (maxLength - 1)};
   typedef StaticArray1 <T, lowerBound, upperBound> ArrayOfT;

   ArrayOfT contents;
   Integer currentLength;

   // correspondence self = IteratedConcatenation(
   //       z:Integer 0 <= z < currentLength, <contents(z)>)
   // convention 0 <= currentLength < maxLength
};
```

A Bounded Queue Implementation

- So what makes this a *bounded* Queue is the fact that the Queue is layered on StaticArray which has a fixed (bounded) size by using the *maxLength* template parameter

- In the private part two named values are declared using the C++ enum construct
  - `lowerBound = 0`
  - `upperBound = (maxLength - 1)`

```
// Filename: BoundedQueue1.hpp
#pragma once
#include "StaticArray\StaticArray1.hpp"

template <class T, int maxLength>
class BoundedQueue1
{
public:
   // Standard Operations
   BoundedQueue1();
   ~BoundedQueue1();
   void clear(void);
   void transferFrom(BoundedQueue1& source);
   BoundedQueue1& operator =(BoundedQueue1& rhs);

   // BoundedQueue1 Specific Operations
   void enqueue(T& x);
   void dequeue(T& x);
   void replaceFront(T& x);
   Integer length(void);
      //! preserves self
      //! ensures: length = |self|
   T& front(void);
   Integer remainingCapacity(void);

private: // internal representation
   enum {lowerBound = 0, upperBound = (maxLength - 1)};
   typedef StaticArray1 <T, lowerBound, upperBound> ArrayOfT;

   ArrayOfT contents;
   Integer currentLength;

   // correspondence self = IteratedConcatenation(
   //       z:Integer 0 <= z < currentLength, <contents(z)>)
   // convention 0 <= currentLength < maxLength
};
```

A Bounded Queue Implementation

- These two named values (lowerBound and upperBound) are used to instantiate an instance of StaticArray
- In this example with *maxLength* = 10 then the array will be indexed from 0 .. 9
- So the BoundedQueue will be able to hold a maximum of 10 items

```
// Filename: BoundedQueue1.hpp
#pragma once
#include "StaticArray\StaticArray1.hpp"

template <class T, int maxLength>
class BoundedQueue1
{
public:
   // Standard Operations
   BoundedQueue1();
   ~BoundedQueue1();
   void clear(void);
   void transferFrom(BoundedQueue1& source);
   BoundedQueue1& operator =(BoundedQueue1& rhs);

   // BoundedQueue1 Specific Operations
   void enqueue(T& x);
   void dequeue(T& x);
   void replaceFront(T& x);
   Integer length(void);
      //! preserves self
      //! ensures: length = |self|
   T& front(void);
   Integer remainingCapacity(void);

private: // internal representation
   enum {lowerBound = 0, upperBound = (maxLength - 1)};
   typedef StaticArray1 <T, lowerBound, upperBound> ArrayOfT;

   ArrayOfT contents;
   Integer currentLength;

   // correspondence self = IteratedConcatenation(
   //       z:Integer 0 <= z < currentLength, <contents(z)>)
   // convention 0 <= currentLength < maxLength
};
```

A Bounded Queue Implementation

- The *data members* for this implementation are:
  - `ArrayOfT contents;`
  - `Integer currentLength;`

- `contents` – is declared from the template instance declared from StaticArray and will hold all the enqueued items

- `currentLength` – is an integer that keeps track of how many items are currently in the Queue

- Reminder – because of compiler enforced encapsulation, the internal data members *contents* and *currentLength* are not accessible by the client program

```
// Filename: BoundedQueue1.hpp
#pragma once
#include "StaticArray\StaticArray1.hpp"

template <class T, int maxLength>
class BoundedQueue1
{
public:
   // Standard Operations
   BoundedQueue1();
   ~BoundedQueue1();
   void clear(void);
   void transferFrom(BoundedQueue1& source);
   BoundedQueue1& operator =(BoundedQueue1& rhs);

   // BoundedQueue1 Specific Operations
   void enqueue(T& x);
   void dequeue(T& x);
   void replaceFront(T& x);
   Integer length(void);
      //! preserves self
      //! ensures: length = |self|
   T& front(void);
   Integer remainingCapacity(void);

private: // internal representation
   enum {lowerBound = 0, upperBound = (maxLength - 1)};
   typedef StaticArray1 <T, lowerBound, upperBound> ArrayOfT;

   ArrayOfT contents;
   Integer currentLength;

   // correspondence self = IteratedConcatenation(
   //       z:Integer 0 <= z < currentLength, <contents(z)>)
   // convention 0 <= currentLength < maxLength
};
```

Member Function Implementations

- Are placed at the bottom of the .hpp file

  There are two parts:

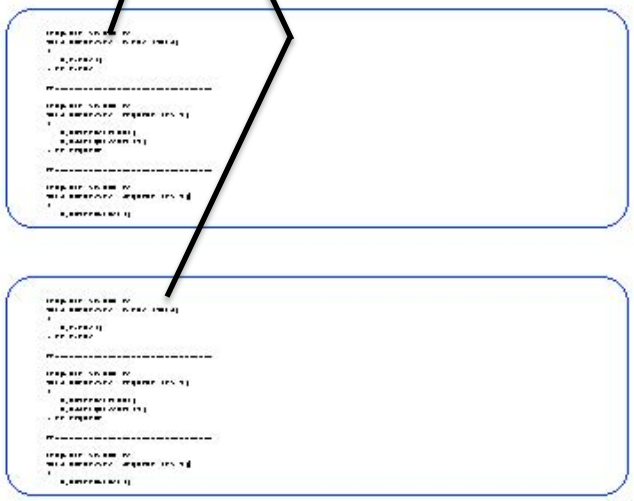
  1. Standard Operations Part

     The member functions that implement the 5 standard operations

  2. Component Specific Operations Part

     The member functions that implement the component specific operations

```
// Filename: BoundedQueue1.hpp
#pragma once
#include "StaticArray\StaticArray1.hpp"

template <class T, int maxLength>
class BoundedQueue1
{
public:
   // Standard Operations
   BoundedQueue1();
   ~BoundedQueue1();
   void clear(void);
   void transferFrom(BoundedQueue1& source);
   BoundedQueue1& operator =(BoundedQueue1& rhs);

   // BoundedQueue1 Specific Operations
   void enqueue(T& x);
   void dequeue(T& x);
   void replaceFront(T& x);
   Integer length(void);
      //! preserves self
      //! ensures: length = |self|
   T& front(void);
   Integer remainingCapacity(void);

private: // internal representation
   enum {lowerBound = 0, upperBound = (maxLength - 1)};
   typedef StaticArray1 <T, lowerBound, upperBound> ArrayOfT;

   ArrayOfT contents;
   Integer currentLength;

   // correspondence self = IteratedConcatenation(
   //       z:Integer 0 <= z < currentLength, <contents(z)>)
   // convention 0 <= currentLength < maxLength
};
```



Member Function Implementations

- Recall: all member function implementations:
  - Work with the concrete internal representation
  - For this example of BoundedQueue template, the *member functions* work with the *data members* `contents` and `currentLength`

```
// Filename: BoundedQueue1.hpp
#pragma once
#include "StaticArray\StaticArray1.hpp"
template <class T, int maxLength>
class BoundedQueue1
{
  // Queue1 Specific Operations
  void enqueue(T& x);
  void dequeue(T& x);
private: // internal representation
   enum {lowerBound = 0, upperBound = (maxLength - 1)};
   typedef StaticArray1 <T, lowerBound, upperBound> ArrayOfT;

   ArrayOfT contents;
   Integer currentLength;
};
```

```
template <class T, int maxLength>
BoundedQueue1<T, maxLength>::BoundedQueue1 ()
{
  s.clear();
  // enqueue
}

template <class T, int maxLength>
BoundedQueue1<T, maxLength>::~BoundedQueue1 ()
{
  s.clear();
  // enqueue
}

template <class T, int maxLength>
void BoundedQueue1<T, maxLength>::
        transferFrom (BoundedQueue1& source)
{
  s.clear();
  // enqueue
}

template <class T, int maxLength>
BoundedQueue1<T, maxLength>&
BoundedQueue1<T, maxLength>::operator = (BoundedQueue1& rhs)
{
  s.clear();
  // enqueue
}

template <class T, int maxLength>
void BoundedQueue1<T, maxLength>::clear (void)
{
  s.clear();
  // enqueue
}
```

```
template <class T, int maxLength>
BoundedQueue1<T, maxLength>::operator = (BoundedQueue1& rhs)
{
  s.clear();
  // enqueue
}
```

Standard Operations Part

- The 5 Standard Operations are:
  1. BoundedQueue1 – the constructor
  2. ~BoundedQueue1 – the destructor
  3. transferFrom
  1. operator = – the assignment operator
  2. clear

The Constructor - Implementation

```
template <class T, int maxLength>
BoundedQueue1<T, maxLength>::BoundedQueue1 ()
   //! alters self
   //! ensures: self = < >
{
}  // BoundedQueue1
```

- *BoundedQueue1* – Has the same name as the component
- The constructor's code initializes the data members
- This constructor has no executable code
- Why?
  - Because data members declared from layered upon components, e.g., `contents` declared from ArrayOfT, are automatically initialized by the layered upon component's constructors
  - We do not make explicit calls to these lower level constructors, the C++ compiler guarantees that these constructors will be called automatically for us
  - In this example, StaticArray's constructor will automatically be called on `contents`, and Integer's constructor will automatically be called for `currentLength`

The Constructor - Initialization

```
template <class T, int maxLength>
BoundedQueue1<T, maxLength>::BoundedQueue1 ()
    //! alters self
    //! ensures: self = < >
{
}  // BoundedQueue1
```

- StaticArray's constructor initializes `contents` to:
  `contents = [0,0,0,0,0,0,0,0,0,0]`

  Recall: in this example Queue has been instantiated by the client with type Integer, so internal to the Queue Integers will be stored in the array

- Integer's constructor initializes `currentLength` to:
  `currentLength = 0`

The Constructor - Correspondence

```
template <class T, int maxLength>
BoundedQueue1<T, maxLength>::BoundedQueue1 ()
   //! alters self
   //! ensures: self = < >
{
}  // BoundedQueue1
```

The *correspondence* for BoundedQueue states:

```
self = IteratedConcatenation (
       z:Integer 0 <= z < currentLength, <contents(z)>)
```

Explanation

- When `currentLength = 0` then:
  `self = < >`
  Because: 0 <= z < currentLength iterates IteratedConcatenation zero times and the base case produced by IteratedConcatenation is the empty string, or < >

- If q = <33, 44,77> after 3 calls to enqueue then
  `currentLength = 3`
  `contents = [33,44,77,0,0,0,0,0,0,0]`

  In this case 0 <= z < 3 IteratedConcatenation iterates 3 times and produces the string <33,44,77>

  By iterating and concatenating `<contents(z)> for z = 0 up to 2`, we obtain the following:
  `self = <contents(0)> * <contents(1)> * <contents(2)>`
  `self = <33> * <44> * <77>`
  `self = <33,44,77>`

The Destructor

```
template <class T, int maxLength>
BoundedQueue1<T, maxLength>::~BoundedQueue1 ()
{
}  // ~BoundedQueue1
```

- *~BoundedQueue1* - Has the same name as the constructor with a tilde prepended to the name
- The job of the destructor is to return back to the system dynamically allocated resources
- Data members from layered upon components: Have their destructors called when the variable goes out of scope, the C++ compiler guarantees this
- Because Queue is layered on StaticArray, no code is required for Queue's destructor, because StaticArray's destructor will automatically get called for the data member `contents`

BoundedQueue() automatically called just after q1 & q2 are declared

~BoundedQueue () automatically called as q1 & q2 go out of scope

```
{  // Example client of BoundedQueue1

   typedef BoundedQueue1<Integer> IntegerQueue;
   IntegerQueue q1, q2;

   // client code manipulating q1 and q2
   //        code is not shown

}
```

clear

```
template <class T, int maxLength>
void BoundedQueue1<T, maxLength>::clear (void)
{
   contents.clear();
   currentLength.clear();
}  // clear
```

clear' s job is to reset the value of the variable back to its initial value

For Queue layered on StaticArray:

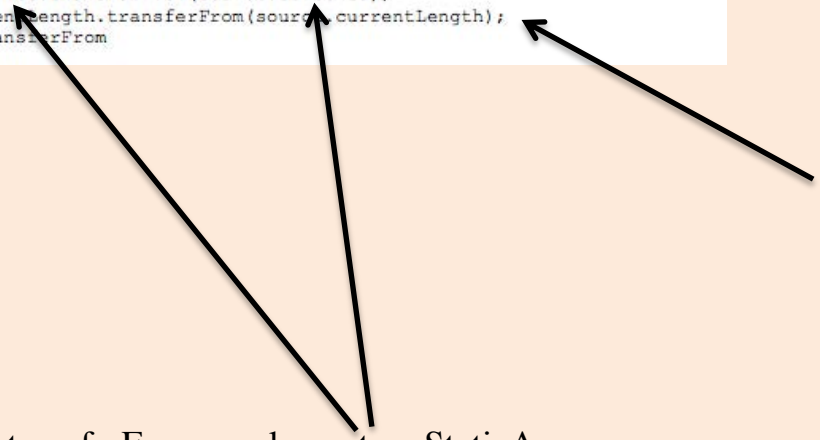
- The ensures clause for clear:
  `self = < >`

- Queue's clear only has to do a *call through* for both of its data members:
  - to StaticArray's *clear* operation for `contents`
  - to Integer's *clear* operation for `currentLength`

- A *call through* is when an operation in a layered component simply calls the operation with the same name at the lower level, i.e., from the layered upon component – in this example Queue's clear calls StaticArray's clear and Integer's clear

transferFrom

```
template <class T, int maxLength>
void BoundedQueue1<T, maxLength>::transferFrom (BoundedQueue1& source)
{
   contents.transferFrom(source.contents);
   currentLength.transferFrom(source.currentLength);
} // transferFrom
```



transferFrom works on two StaticArrays:

1. `contents` in front of the dot
2. `source.contents` passed in as a parameter

transferFrom's job is to transfer the value from the source variable to the variable in front of the dot

For Queue layered on StaticArray:

- This is easily accomplished by calling through to StaticArray's *transferFrom* and Integer's *transferFrom*
- BoundedQueue1's *transferFrom* has parameter `source`
  - It is of type BoundedQueue1
  - StaticArray's *transferFrom* cannot be called as follows: `countents.tranferFrom(source);`

    This would be a type mis-match because StaticArray's *transferFrom* works on two StaticArrays

    So we must use the dot operator to dot our way into source's data member `contents`, which of course is a StaticArray

    The correct call is:
    `contents.tranferFrom(source.contents);`

operator =

```
template <class T, int maxLength>
BoundedQueue1<T, maxLength>&
BoundedQueue1<T, maxLength>::operator = (BoundedQueue1& rhs)
{
        contents = rhs.contents;
        currentLength = rhs.currentLength;
        return *this;
} // operator =
```

operator = is the assignment operator in C++, and its job is to make a copy of the variable that appears on the right hand side (rhs) of the equals sign (=) and place the copy in the variable on the left hand side (lhs)

```
{
   // Example client of BoundedQueue1
   typedef BoundedQueue1<Integer> IntegerQueue;
   IntegerQueue q1, q2;

   q2 = q1;
   // Or in C++ we could have written:
   q2.operator=(q1);
   // Both do the same thing, and both compile
}
```

- Queue's layered *operator =* is implemented by:
  - By calling through to StaticArray's *operator =*
  - By calling through to Integer's *operator =*

  - Again, we have to use C++'s dot operator to gain access to the StaticArray and Integer inside of the parameter rhs

- `return *this;`

  In C++, the return statement is required so that clients can write code containing multiple assignments on one line, for example: `q2 = q1 = q3;`

```
// Filename: BoundedQueue1.hpp
#pragma once
#include "StaticArray\StaticArray1.hpp"
template <class T, int maxLength>
class BoundedQueue1
{
   // Queue1 Specific Operations
   void enqueue(T& x);
   void dequeue(T& x);

private: // internal representation
   enum {lowerBound = 0, upperBound = (maxLength - 1)};
   typedef StaticArray1 <T, lowerBound, upperBound> ArrayOfT;

   ArrayOfT contents;
   Integer currentLength;
};
```

```
template <class T, int maxLength>
void BoundedQueue1<T, maxLength>::enqueue (T& x)
{
   x.clear();
   // replaceFront
}
template <class T, int maxLength>
void BoundedQueue1<T, maxLength>::dequeue(T& x)
{
   x.clear();
   // replaceFront
}
template <class T, int maxLength>
void BoundedQueue1<T, maxLength>::replaceFront (T& x)
{
   x.clear();
   // replaceFront
}
template <class T, int maxLength>
T& BoundedQueue1<T, maxLength>::front (void)
{
   x.clear();
   // replaceFront
}
template <class T, int maxLength
Integer_BoundedQueue1<T, maxLength>::length (void)
{
   x.clear();
   // replaceFront
}
template <class T, int maxLength>
Integer_BoundedQueue1<T, maxLength>::remainingCapacity(void)
{
   x.clear();
   // replaceFront
}
```

Component Specific Operations

- Queue's component specific operations are:

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

enqueue – conceptual value

```
template <class T, int maxLength>
void BoundedQueue1<T, maxLength>::enqueue (T& x)
   //! alters self
   //! consumes x
   //! requires: |self| + 1 <= maxLength
   //! ensures: self = #self * <#x>
{
   contents[currentLength].transferFrom(x);
   currentLength++;
}  // enqueue
```

enqueue allows the client program to insert an item at the rear of the queue

```
{  // Example client of BoundedQueue1

   typedef BoundedQueue1<Integer> IntegerQueue;
   IntegerQueue q1;
   Integer y2;

   // Some code not shown to enqueue 3 items onto q1
   // Incoming: q1 = <18,15,27>  y2 = 5
   q1.enqueue(y2);
   // Outgoing: q1 = <18,15,27,5>  y2 = 0
}
```

1. *enqueue's* ensures clause
   `ensures: self = #self * <#x>`

2. *enqueue's* ensures clause after *variable* substitution is:
   `ensures: q1 = #q1 * <#y2>`

   Where: `#q1 = <18,15,27> and #y2 = 5`

3. *enqueue's* ensures clause after *value* substitution is:
   `q1 = <18,15,27> * <5>`
   `= <18,15,27,5>`

enqueue – Internal Changes

```
template <class T, int maxLength>
void BoundedQueue1<T, maxLength>::enqueue (T& x)
    //! alters self
    //! consumes x
    //! requires: |self| + 1 <= maxLength
    //! ensures: self = #self * <#x>
{
    contents[currentLength].transferFrom(x);
    currentLength++;
}   // enqueue
```

What value does `contents` (inside q1) have after enqueue?
Answer: `contents = [18,15,27,5,0,0,0,0,0,0]`

Explanation

Before the call:

`contents = [18,15,27,0,0,0,0,0,0,0]`
`currentLength = 3`

`contents[currentLength].transferFrom(x);`
gave `contents` the value:
`contents = [18,15,27,5,0,0,0,0,0,0]`
gave the parameter `x` the value:
`x = 0`

`currentLength++;`
gave `currentLength` the value:
`currentLength = 4`

enqueue – correspondence

```
template <class T, int maxLength>
void BoundedQueue1<T, maxLength>::enqueue (T& x)
    //! alters self
    //! consumes x
    //! requires: |self| + 1 <= maxLength
    //! ensures: self = #self * <#x>
{
    contents[currentLength].transferFrom(x);
    currentLength++;
}   // enqueue
```

And using the *correspondence* we get:

```
self = IteratedConcatenation (
       z:Integer 0 <= z < currentLength, <contents(z)>)

  = <18,15,27,5>
```

Explanation

After the call:

```
contents = [18,15,27,5,0,0,0,0,0,0]
currentLength = 4
```

Applying the correspondence and its IteratedConcatenation to the array `contents` we get:

```
self = IteratedConcatenation (
       z:Integer 0 <= z < 4, <contents(z)>)

  = <contents(0)>*<contents(1)*<contents(2)>*<contents(3)>
  = <18> * <15> * <27> * <5>
  = <18,15,27,5>
```

Which correctly matches *enqueue's* ensures clause

Note: when applying the correspondence to the internal data members if we do not come up with the same value as the ensures clause, then that means there is a problem with the implementation of the operation

dequeue – conceptual value

dequeue allows the client program to remove an item from the front of a non-empty queue

```
{  // Example client of BoundedQueue1

   typedef BoundedQueue1<Integer> IntegerQueue;
   IntegerQueue q1;
   Integer y2;

   // Incoming: q1 = <18,15,27>  y2 = 100
   q1.dequeue(y2);
   // Outgoing: q1 = <15,27>  y2 = 18
}
```

```
template <class T, int maxLength>
void BoundedQueue1<T, maxLength>::dequeue(T& x)
  //! alters self
  //! produces x
  //! requires: self /= < >
  //! ensures: <x> = #self[0,1) and
  //!          self = #self[1, |#self|)
{
   static Integer locationZero = 0;
   x.transferFrom(contents[locationZero]);
   for (int k = 0, z = (currentLength - 1); k < z; k++) {
      contents[k].transferFrom(contents[k + 1]);
   } // end for
   currentLength--;
}  // dequeue
```

1. *dequeue's* ensures clause:

   ```
   ensures: <x> = #self[0,1)  and
            self = #self[1, |#self|)
   ```

2. *dequeue's* ensures clause after *variable* substitution is:

   ```
   ensures: <y2> = #q1[0,1) and
            q1 = #q1[1,|#q1|)
   ```

   Where: `#q1 = <18,15,27> and #y2 = 100`

3. *dequeue's* ensures clause after *value* substitution is:

   ```
   ensures: <y2> = <18,15,27>[0,1) and
            q1 = <18,15,27>[1,3)
   ```

   So: `y2 = 18 and q1 = <15,27>`

dequeue – Internal Changes

```
template <class T, int maxLength>
void BoundedQueue1<T, maxLength>::dequeue(T& x)
  //! alters self
  //! produces x
  //! requires: self /= < >
  //! ensures: <x> = #self[0,1) and
  //!          self = #self[1, |#self|)
{
   static Integer locationZero = 0;
   x.transferFrom(contents[locationZero]);
   for (int k = 0, z = (currentLength - 1); k < z; k++) {
      contents[k].transferFrom(contents[k + 1]);
   } // end for
   currentLength--;
}  // dequeue
```

What value does `contents` (inside q1) have after dequeue?
Answer: `contents = [15,27,0,0,0,0,0,0,0,0]`

Explanation

Before the call:

`contents = [18,15,27,0,0,0,0,0,0,0]`
`currentLength = 3`

`x.transferFrom.contents[locationZero];`
gave parameter `x` the value:
`x = 18`
and array `contents` the value:
`contents = [0,15,27,0,0,0,0,0,0,0]`

The for loop moves the remaining items 1 array location toward the front:

`contents = [0,15,27,0,0,0,0,0,0,0]`

`contents = [15,27,0,0,0,0,0,0,0,0]`

`currentLength--;`
gave `currentLength` the value:
`currentLength = 2`

dequeue – correspondence

```
template <class T, int maxLength>
void BoundedQueue1<T, maxLength>::dequeue(T& x)
  //! alters self
  //! produces x
  //! requires: self /= < >
  //! ensures: <x> = #self[0,1) and
  //!          self = #self[1, |#self|)
{
   static Integer locationZero = 0;
   x.transferFrom(contents[locationZero]);
   for (int k = 0, z = (currentLength - 1); k < z; k++) {
      contents[k].transferFrom(contents[k + 1]);
   } // end for
   currentLength--;
}  // dequeue
```

And using the *correspondence* we get:

```
self = IteratedConcatenation (
       z:Integer 0 <= z < currentLength, <contents(z)>)

     = <15,27>
```

Explanation

After the call:

```
contents = [15,27,0,0,0,0,0,0,0,0]
currentLength = 3
```

Applying the correspondence and its IteratedConcatenation to the array `contents` we get:

```
self = IteratedConcatenation (
       z:Integer 0 <= z < 3, <contents(z)>)

  = <contents(0)> * <contents(1)>
  = <15> * <27>
  = <15,27>
```

Which correctly matches *dequeue's* ensures clause

replaceFront – conceptual value

```
template <class T, int maxLength>
void BoundedQueue1<T, maxLength>::replaceFront (T& x)
   //! alters self, x
   //! requires: self /= < >
   //!  ensures: x = #self[0,1) and
   //!           self = <#x> * #self[1, |#self|)
{
   static Integer locationZero = 0
   T temp;

   temp.transferFrom(contents[locationZero]);
   contents[locationZero].transferFrom(x);
   x.transferFrom(temp);
}  // replaceFront
```

replaceFront allows the client program to simultaneously replace the item at the front of the queue and also obtain the value that was at the front

```
{  // Example client of BoundedQueue1

   typedef BoundedQueue1<Integer> IntegerQueue;
   IntegerQueue q1;
   Integer y2;

   // Incoming: q1 = <18,15,27>  y2 = 100
   q1.replaceFront(y2);
   // Outgoing: q1 = <100,15,27>  y2 = 18
}
```

1. *replaceFront's* ensures clause:

   ```
   ensures: <x> = #self[0,1)  and
            self = <#x> * #self[1, |#self|)
   ```

2. *replaceFront's* ensures clause after *variable* substitution is:

   ```
   ensures: <y2> = #q1[0,1) and
            q1 = <#y2> * #q1[1,|#q1|)
   ```

   Where: `#q1 = <18,15,27> and #y2 = 100`

3. *replaceFront's* ensures clause after *value* substitution is:

   ```
   ensures: <y2> = <18,15,27>[0,1) and
            q1 = <100> * <18,15,27>[1,3)
   ```

   So: `y2 = 18 and q1 = <100,15,27>`

replaceFront – Internal Changes

```
template <class T, int maxLength>
void BoundedQueue1<T, maxLength>::replaceFront (T& x)
    //! alters self, x
    //! requires: self /= < >
    //!  ensures: x = #self[0,1) and
    //!           self = <#x> * #self[1, |#self|)
{
    static Integer locationZero = 0
    T temp;

    temp.transferFrom(contents[locationZero]);
    contents[locationZero].transferFrom(x);
    x.transferFrom(temp);
}   // replaceFront
```

What value does `contents` (inside q1) have after replaceFront?
Answer: `contents = [100,15,27,0,0,0,0,0,0,0]`

Explanation

Before the call:

`contents = [18,15,27,0,0,0,0,0,0,0]`
`currentLength = 3`

`temp.transferFrom.contents[locationZero];`
gave local variable `temp` the value:
`temp = 18`
and array `contents` the value:
`contents = [0,15,27,0,0,0,0,0,0,0]`

`contents[locationZero].transferFrom(x);`
gave the parameter `x` the value:
`x = 0`
and array `contents` the value:
`contents = [100,15,27,0,0,0,0,0,0,0]`

`x.transferFrom(temp);`
gave the parameter `x` the value:
`x = 18`
and local variable `temp` the value:
`temp = 0`

replaceFront – correspondence

```
template <class T, int maxLength>
void BoundedQueue1<T, maxLength>::replaceFront (T& x)
   //! alters self, x
   //! requires: self /= < >
   //!  ensures: x = #self[0,1) and
   //!           self = <#x> * #self[1, |#self|)
{
   static Integer locationZero = 0
   T temp;

   temp.transferFrom(contents[locationZero]);
   contents[locationZero].transferFrom(x);
   x.transferFrom(temp);
}  // replaceFront
```

And using the *correspondence* we get:

```
self = IteratedConcatenation (
       z:Integer 0 <= z < currentLength, <contents(z)>)

     = <100,15,27>
```

Explanation

After the call:

```
contents = [100,5,27,0,0,0,0,0,0,0]
currentLength = 3
```

Applying the correspondence and its IteratedConcatenation to the array `contents` we get:

```
self = IteratedConcatenation (
       z:Integer 0 <= z < 3, <contents(z)>)

  = <contents(0)> * <contents(1)> *
<contents(2)>
  = <100> * <15> * <27>
  = <100,15,27>
```

Which correctly matches *replaceFront's* ensures clause

front – conceptual value

```
template <class T, int maxLength>
T& BoundedQueue1<T, maxLength>::front (void)
   //! preserves self
   //! requires: self /= < >
   //!  ensures: <front> = self[0,1)
{
   static Integer locationZero = 0;

   return contents[locationZero];
}  // front
```

front allows the client program to inspect the value at the front of the queue

```
{  // Example client of BoundedQueue1

   typedef BoundedQueue1<Integer> IntegerQueue;
   IntegerQueue q1;
   Integer z;

   // Incoming: q1 = <111,44>
   cout << q1.front();
   // Outgoing: q1 = <111,44>
}
```

1. *front's* ensures clause:
   `ensures: <front> = self[0,1)`

2. *front's* ensures clause after *variable* substitution is:
   `ensures: <front> = q1[0,1)`

   Where: `q1 = <111,44>`

3. *front's* ensures clause after *value* substitution is:
   `ensures: <front> = <111,44>[0,1)`

   `So: front = 111`

length – conceptual value

```
template <class T, int maxLength
Integer BoundedQueue1<T, maxLength>::length (void)
   //! preserves self
   //! ensures: length = |self|
{
   return currentLength;
}  // length
```

length allows the client program to determine how many items are currently in the queue

```
{  // Example client of BoundedQueue1

   typedef BoundedQueue1<Integer> IntegerQueue;
   IntegerQueue q1;
   Integer z;

   // Incoming: q1 = <18,15,27>  z = 0
   z = q1.length();
   // Outgoing: q1 = <18,15,27>  z = 3
}
```

1. *length's* ensures clause:
   `ensures: length = |self|`

2. *length's* ensures clause after *variable* substitution is:
   `ensures: length = |q1|`

   Where: `q1 = <18,15,27>`

3. *length's* ensures clause after *value* substitution is:
   `ensures: length = |<18,15,27>|`

   `So: length = 3`

remainingCapacity – conceptual value

```
template <class T, int maxLength>
Integer BoundedQueue1<T, maxLength>::remainingCapacity(void)
   //! preserves self
   //! ensures: remainingCapacity  = maxLength - |self|
{
   return (maxLength - currentLength);
}  // remainingCapacity
```

remainingCapacity allows the client program to determine how many more items can be enqueued

```
{  // Example client of BoundedQueue1

   typedef BoundedQueue1<Integer> IntegerQueue;
   IntegerQueue q1;
   Integer z;

   // Incoming: q1 = <18,15,27>  z = 0
   z = q1.remainingCapacity();
   // Outgoing: q1 = <18,15,27>  z = 7
}
```

1. *remainingCapacity's* ensures clause:
   `ensures: remainingCapacity = maxLength - |self|`

2. *remainingCapacity's* ensures clause after *variable* substitution is:
   `ensures: remainingCapacity = maxLength - |q1|`

   Where: `maxLength = 10 and q1 = <18,15,27>`

3. *remainingCapacity's* ensures clause after *value* substitution is:
   `ensures: remainingCapacity = 10 - 3`

   `So: remainingCapacity = 7`