

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)>  
)
```

```

PROGRAM CS10000000
  REAL*8 X(1000000)
  I
  X=1.0/1000000
  PRINT X(1000000)
  STOP
END

PROGRAM CS10000000
  REAL*8 X(1000000)
  I
  X=1.0/1000000
  PRINT X(1000000)
  STOP
END

PROGRAM CS10000000
  REAL*8 X(1000000)
  I
  X=1.0/1000000
  PRINT X(1000000)
  STOP
END

```

- 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:

2


```
// 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
};
```

```

PROGRAM SUMM TO
      DIMENSION A(10),B(10)
      DATA A / 1,2,3,4,5,6,7,8,9,10 /
      DATA B / 10,9,8,7,6,5,4,3,2,1 /
      DO 10 I=1,N
        S=A(I)+B(I)
      10 CONTINUE
      PRINT *,S
      STOP
END

```

```

    require('fs').readFile(
      path.resolve(__dirname, 'index.html'),
      function(err, html) {
        if (err) return console.log(err);
        // ...
      }
    );

```

RESEARCH DESIGN

```

1000  IF (N1.EQ.0) GO TO 1005
1010  CALL SUBROUTINE (N1,N2,N3,N4,N5,N6,N7,N8,N9,N10,N11,N12,N13,N14,N15,N16,N17,N18,N19,N20,N21,N22,N23,N24,N25,N26,N27,N28,N29,N30,N31,N32,N33,N34,N35,N36,N37,N38,N39,N40,N41,N42,N43,N44,N45,N46,N47,N48,N49,N50,N51,N52,N53,N54,N55,N56,N57,N58,N59,N60,N61,N62,N63,N64,N65,N66,N67,N68,N69,N70,N71,N72,N73,N74,N75,N76,N77,N78,N79,N80,N81,N82,N83,N84,N85,N86,N87,N88,N89,N90,N91,N92,N93,N94,N95,N96,N97,N98,N99,N100,N101,N102,N103,N104,N105,N106,N107,N108,N109,N110,N111,N112,N113,N114,N115,N116,N117,N118,N119,N120,N121,N122,N123,N124,N125,N126,N127,N128,N129,N130,N131,N132,N133,N134,N135,N136,N137,N138,N139,N140,N141,N142,N143,N144,N145,N146,N147,N148,N149,N150,N151,N152,N153,N154,N155,N156,N157,N158,N159,N160,N161,N162,N163,N164,N165,N166,N167,N168,N169,N170,N171,N172,N173,N174,N175,N176,N177,N178,N179,N180,N181,N182,N183,N184,N185,N186,N187,N188,N189,N190,N191,N192,N193,N194,N195,N196,N197,N198,N199,N200,N201,N202,N203,N204,N205,N206,N207,N208,N209,N210,N211,N212,N213,N214,N215,N216,N217,N218,N219,N220,N221,N222,N223,N224,N225,N226,N227,N228,N229,N230,N231,N232,N233,N234,N235,N236,N237,N238,N239,N240,N241,N242,N243,N244,N245,N246,N247,N248,N249,N250,N251,N252,N253,N254,N255,N256,N257,N258,N259,N260,N261,N262,N263,N264,N265,N266,N267,N268,N269,N270,N271,N272,N273,N274,N275,N276,N277,N278,N279,N280,N281,N282,N283,N284,N285,N286,N287,N288,N289,N290,N291,N292,N293,N294,N295,N296,N297,N298,N299,N300,N301,N302,N303,N304,N305,N306,N307,N308,N309,N310,N311,N312,N313,N314,N315,N316,N317,N318,N319,N320,N321,N322,N323,N324,N325,N326,N327,N328,N329,N330,N331,N332,N333,N334,N335,N336,N337,N338,N339,N340,N341,N342,N343,N344,N345,N346,N347,N348,N349,N350,N351,N352,N353,N354,N355,N356,N357,N358,N359,N360,N361,N362,N363,N364,N365,N366,N367,N368,N369,N370,N371,N372,N373,N374,N375,N376,N377,N378,N379,N380,N381,N382,N383,N384,N385,N386,N387,N388,N389,N390,N391,N392,N393,N394,N395,N396,N397,N398,N399,N400,N401,N402,N403,N404,N405,N406,N407,N408,N409,N410,N411,N412,N413,N414,N415,N416,N417,N418,N419,N420,N421,N422,N423,N424,N425,N426,N427,N428,N429,N430,N431,N432,N433,N434,N435,N436,N437,N438,N439,N440,N441,N442,N443,N444,N445,N446,N447,N448,N449,N450,N451,N452,N453,N454,N455,N456,N457,N458,N459,N460,N461,N462,N463,N464,N465,N466,N467,N468,N469,N470,N471,N472,N473,N474,N475,N476,N477,N478,N479,N480,N481,N482,N483,N484,N485,N486,N487,N488,N489,N490,N491,N492,N493,N494,N495,N496,N497,N498,N499,N500,N501,N502,N503,N504,N505,N506,N507,N508,N509,N510,N511,N512,N513,N514,N515,N516,N517,N518,N519,N520,N521,N522,N523,N524,N525,N526,N527,N528,N529,N530,N531,N532,N533,N534,N535,N536,N537,N538,N539,N540,N541,N542,N543,N544,N545,N546,N547,N548,N549,N550,N551,N552,N553,N554,N555,N556,N557,N558,N559,N560,N561,N562,N563,N564,N565,N566,N567,N568,N569,N570,N571,N572,N573,N574,N575,N576,N577,N578,N579,N580,N581,N582,N583,N584,N585,N586,N587,N588,N589,N590,N591,N592,N593,N594,N595,N596,N597,N598,N599,N600,N601,N602,N603,N604,N605,N606,N607,N608,N609,N610,N611,N612,N613,N614,N615,N616,N617,N618,N619,N620,N621,N622,N623,N624,N625,N626,N627,N628,N629,N630,N631,N632,N633,N634,N635,N636,N637,N638,N639,N640,N641,N642,N643,N644,N645,N646,N647,N648,N649,N650,N651,N652,N653,N654,N655,N656,N657,N658,N659,N660,N661,N662,N663,N664,N665,N666,N667,N668,N669,N670,N671,N672,N673,N674,N675,N676,N677,N678,N679,N680,N681,N682,N683,N684,N685,N686,N687,N688,N689,N690,N691,N692,N693,N694,N695,N696,N697,N698,N699,N700,N701,N702,N703,N704,N705,N706,N707,N708,N709,N710,N711,N712,N713,N714,N715,N716,N717,N718,N719,N720,N721,N722,N723,N724,N725,N726,N727,N728,N729,N730,N731,N732,N733,N734,N735,N736,N737,N738,N739,N740,N741,N742,N743,N744,N745,N746,N747,N748,N749,N750,N751,N752,N753,N754,N755,N756,N757,N758,N759,N760,N761,N762,N763,N764,N765,N766,N767,N768,N769,N770,N771,N772,N773,N774,N775,N776,N777,N778,N779,N780,N781,N782,N783,N784,N785,N786,N787,N788,N789,N790,N791,N792,N793,N794,N795,N796,N797,N798,N799,N800,N801,N802,N803,N804,N805,N806,N807,N808,N809,N810,N811,N812,N813,N814,N815,N816,N817,N818,N819,N820,N821,N822,N823,N824,N825,N826,N827,N828,N829,N830,N831,N832,N833,N834,N835,N836,N837,N838,N839,N840,N841,N842,N843,N844,N845,N846,N847,N848,N849,N850,N851,N852,N853,N854,N855,N856,N857,N858,N859,N860,N861,N862,N863,N864,N865,N866,N867,N868,N869,N870,N871,N872,N873,N874,N875,N876,N877,N878,N879,N880,N881,N882,N883,N884,N885,N886,N887,N888,N889,N890,N891,N892,N893,N894,N895,N896,N897,N898,N899,N900,N901,N902,N903,N904,N905,N906,N907,N908,N909,N910,N911,N912,N913,N914,N915,N916,N917,N918,N919,N920,N921,N922,N923,N924,N925,N926,N927,N928,N929,N930,N931,N932,N933,N934,N935,N936,N937,N938,N939,N940,N941,N942,N943,N944,N945,N946,N947,N948,N949,N950,N951,N952,N953,N954,N955,N956,N957,N958,N959,N960,N961,N962,N963,N964,N965,N966,N967,N968,N969,N970,N971,N972,N973,N974,N975,N976,N977,N978,N979,N980,N981,N982,N983,N984,N985,N986,N987,N988,N989,N990,N991,N992,N993,N994,N995,N996,N997,N998,N999,

```

[illegible]

1. *Chlorophyll a* and *b* contents were determined by the method of Arar and Collins (1971).

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

- 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

```
// BoundedQueue1.cpp
#include "BoundedQueue1.hpp"
#include "StaticArray\StaticArray1.hpp"

// Standard Operations
BoundedQueue1::BoundedQueue1()
{
    clear();
}

BoundedQueue1::~BoundedQueue1()
{
}

void BoundedQueue1::clear(void)
{
    currentLength = 0;
}

void BoundedQueue1::transferFrom(BoundedQueue1& source)
{
    // ...
}

BoundedQueue1& BoundedQueue1::operator =(BoundedQueue1& rhs)
{
    // ...
}

// BoundedQueue1 Specific Operations
void BoundedQueue1::enqueue(T& x)
{
    // ...
}

void BoundedQueue1::dequeue(T& x)
{
    // ...
}

void BoundedQueue1::replaceFront(T& x)
{
    // ...
}

Integer BoundedQueue1::length(void)
{
    return currentLength;
}

T& BoundedQueue1::front(void)
{
    // ...
}

Integer BoundedQueue1::remainingCapacity(void)
{
    return maxLength - currentLength;
}
```

```
// BoundedQueue1.cpp
#include "BoundedQueue1.hpp"
#include "StaticArray\StaticArray1.hpp"

// Standard Operations
BoundedQueue1::BoundedQueue1()
{
    clear();
}

BoundedQueue1::~BoundedQueue1()
{
}

void BoundedQueue1::clear(void)
{
    currentLength = 0;
}

void BoundedQueue1::transferFrom(BoundedQueue1& source)
{
    // ...
}

BoundedQueue1& BoundedQueue1::operator =(BoundedQueue1& rhs)
{
    // ...
}

// BoundedQueue1 Specific Operations
void BoundedQueue1::enqueue(T& x)
{
    // ...
}

void BoundedQueue1::dequeue(T& x)
{
    // ...
}

void BoundedQueue1::replaceFront(T& x)
{
    // ...
}

Integer BoundedQueue1::length(void)
{
    return currentLength;
}

T& BoundedQueue1::front(void)
{
    // ...
}

Integer BoundedQueue1::remainingCapacity(void)
{
    return maxLength - currentLength;
}
```



```
// 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
};
```

```
// Standard Operations
// BoundedQueue1::BoundedQueue1()
// BoundedQueue1::~BoundedQueue1()
// BoundedQueue1::clear()
// BoundedQueue1::transferFrom()
```

```
// BoundedQueue1 Specific Operations
// BoundedQueue1::enqueue()
// BoundedQueue1::dequeue()
// BoundedQueue1::replaceFront()
// BoundedQueue1::length()
// BoundedQueue1::front()
// BoundedQueue1::remainingCapacity()
```

```
// Standard Operations
// BoundedQueue1::BoundedQueue1()
// BoundedQueue1::~BoundedQueue1()
// BoundedQueue1::clear()
// BoundedQueue1::transferFrom()
```

```
// BoundedQueue1 Specific Operations
// BoundedQueue1::enqueue()
// BoundedQueue1::dequeue()
// BoundedQueue1::replaceFront()
// BoundedQueue1::length()
// BoundedQueue1::front()
// BoundedQueue1::remainingCapacity()
```

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

[illegible]

- 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

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

```
// 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 ()
{
    // clear()
    // enqueue
}

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

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

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


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

```
template <class T, int maxLength>
void BoundedQueue1<T, maxLength>::
    enqueue(T& x)
{
    // enqueue
}

template <class T, int maxLength>
void BoundedQueue1<T, maxLength>::
    dequeue(T& x)
{
    // dequeue
}
```

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: $0 \leq z < \text{currentLength}$ 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]`

In this case $0 \leq z < 3$ 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

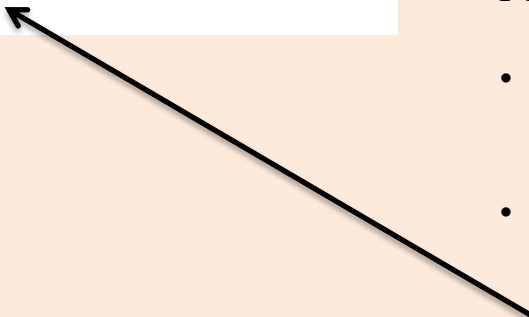
```
{ // Example client of BoundedQueue1

    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'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);
```

transferFrom works on two StaticArrays:

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

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
{
    // Queue's 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;
};
```

```
// Queue's Specific Operations
// enqueue(T& x)
// dequeue(T& x)
// replaceFront(T& x)
// front(void)
// length(void)
// remainingCapacity(void)
```

```
template <class T, int maxLength>
void BoundedQueue1<T, maxLength>::enqueue (T& x)
{
    // enqueue
}

template <class T, int maxLength>
void BoundedQueue1<T, maxLength>::dequeue(T& x)
{
    // dequeue
}

template <class T, int maxLength>
void BoundedQueue1<T, maxLength>::replaceFront (T& x)
{
    // replaceFront
}

template <class T, int maxLength>
T& BoundedQueue1<T, maxLength>::front (void)
{
    // front
}

template <class T, int maxLength>
Integer BoundedQueue1<T, maxLength>::length (void)
{
    // length
}

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

Component Specific Operations

- Queue's component specific operations are:
 1. enqueue
 2. dequeue
 3. replaceFront
- 1. front
- 2. length
- 3. remainingCapacity

enqueue – conceptual value

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

```
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
```

```
{ // 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

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:

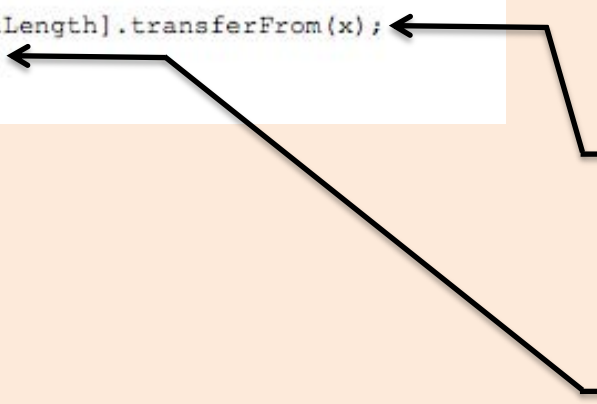
x = 0

currentLength++;

gave currentLength the value:

currentLength = 4

```
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 – correspondence

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

```
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
```

dequeue – conceptual value

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

```
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
```

```
{ // 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
}
```

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

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

```
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
```

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

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

```
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
```

```
{ // 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: $\langle x \rangle = \#self[0,1)$ and
 $self = \langle \#x \rangle * \#self[1, | \#self |)$
2. *replaceFront*'s ensures clause after *variable* substitution is:
ensures: $\langle y2 \rangle = \#q1[0,1)$ and
 $q1 = \langle \#y2 \rangle * \#q1[1, | \#q1 |)$

Where: $\#q1 = \langle 18, 15, 27 \rangle$ and $\#y2 = 100$

3. *replaceFront*'s ensures clause after *value* substitution is:
ensures: $\langle y2 \rangle = \langle 18, 15, 27 \rangle[0,1)$ and
 $q1 = \langle 100 \rangle * \langle 18, 15, 27 \rangle[1, 3)$

So: $y2 = 18$ and $q1 = \langle 100, 15, 27 \rangle$

replaceFront – Internal Changes

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

```
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
```

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

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

```
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
```

```
{ // 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>

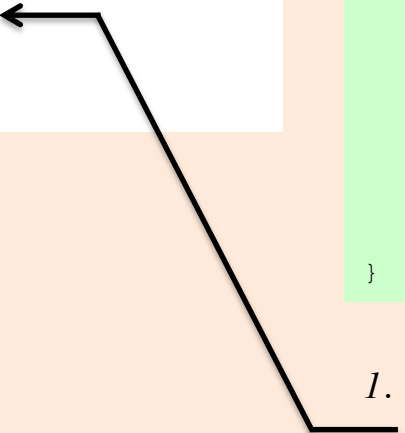
3. *front*'s ensures clause after *value* substitution is:
ensures: <front> = <111,44>[0,1)

So: front = 111

length – conceptual value

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

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



```
{ // 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

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

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

```
{ // 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