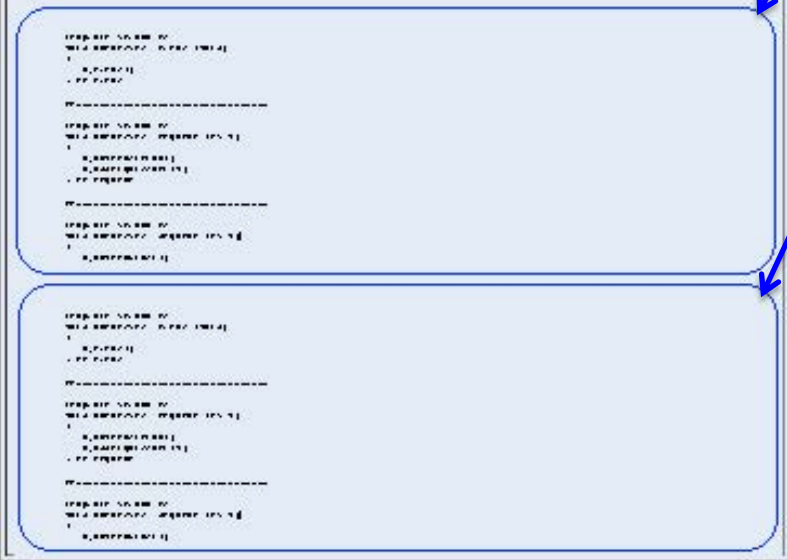
Member Function Implementations Of a Layered Component

Queue Layered on List Implementation #1

Illustrating the *correspondence*

`self = s.left * s.right`

```
// 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
};
```



Member Function Implementations

- Are placed at the bottom of the .hpp file

  There are two parts:

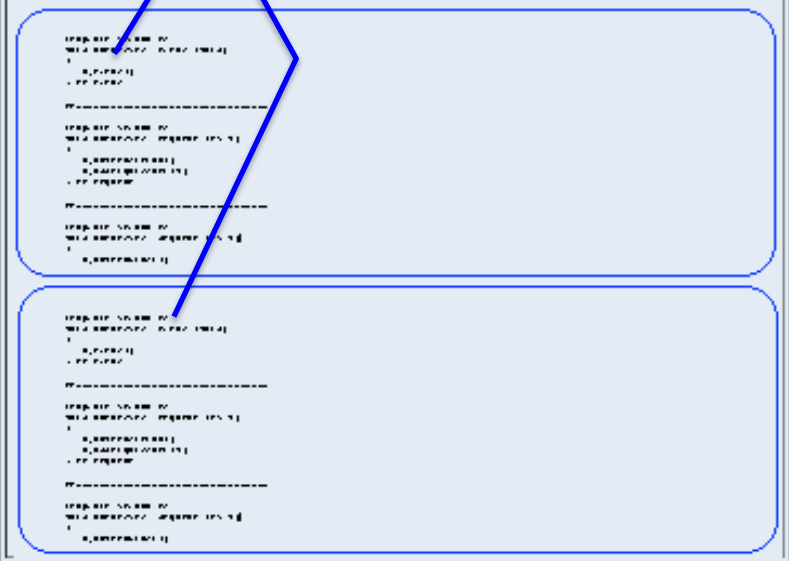
  1. Standard Operations Part

     The member functions that implement the 5 standard operations

  2. Component Specific Operations Part

     The member functions that implement the component specific operations

```
// Filename: 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
};
```



Member Function Implementations

- Recall: all member function implementations:
  - Work with the concrete internal representation
  - For this example using Queue, the member functions work with variable *s*, declared from ListOfT

```
// 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);
 int length();
 T frontValue();

private: // Internal Representation
   typedef List1<T> ListOfT;
   ListOfT s;
};
```

```
template <class T>
Queue2<T>::Queue2 ()
{
   s.clear();
} // clear
template <class T>
Queue2<T>::~Queue2 ()
{
   s.clear();
} // enqueue
template <class T>
void Queue2<T>::clear (void)
{
   s.clear();
} // enqueue
template <class T>
void Queue2<T>::transferFrom (Queue2& source)
{
   s.clear();
} // dequeue
template <class T>
Queue2<T>& Queue2<T>::operator = (Queue2& rhs)
{
   s.clear();
} // replaceFront
```

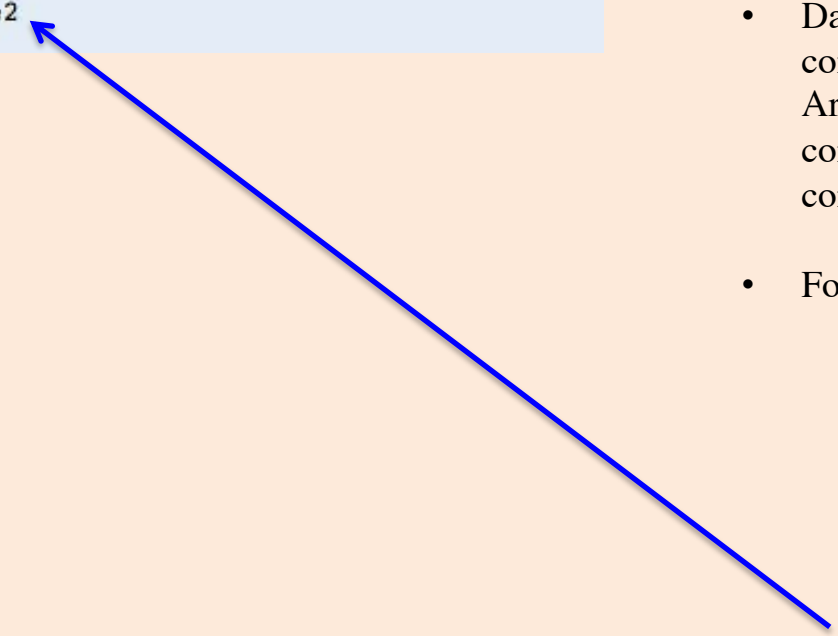
```
template <class T>
{
}

//-------------------------

template <class T>
```

Standard Operations Part

- The 5 Standard Operations are:
  1. Queue2 – the constructor
  2. ~Queue2 – the destructor
  3. transferFrom
  1. operator = – the assignment operator
  2. clear

The Constructor

```
template <class T>
Queue2<T>::Queue2 ()
   //! replaces self
   //! ensures: self = < >
{
} // Queue2
```



- Has the same name as the component
- The constructor's code initializes the data members
- Data members declared from layered upon components, e.g., *s* declared from ListOfT:
  Are automatically initialized by their own constructors, the C++ compiler guarantees that these constructors will be called
- For Queue layered on List:
  - List's constructor initializes *s* to:
    `s = (<>,<>)`
  - Recall the *correspondence*:
    `self = s.left * s.right`
  - So no code is required for Queue's constructor, because the ensures clause for Queue's constructor is:
    `self = <>`
    and
    ```
    self = s.left * s.right
         = <> * <>
         = <>
    ```

The Destructor

```
template <class T>
Queue2<T>::~Queue2 ()
{
} // ~Queue2
```

- Has the same name as the constructor with a tilde prepended to the name
- The job of the destructor is to return dynamically allocated resources back to the system
- All 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 List, no code is required for Queue's destructor, because List's destructor will automatically get called for data member *s*

Queue() automatically called when q1 & q2 are declared

~Queue() automatically called as q1 & q2 go out of scope, so q1.s's destructor is called and q2.s's destructor is called

Example client:

```
{
1  typedef Queue2<Integer> IntegerQueue;
2  IntegerQueue q1, q2;
3
4  // client code manipulating q1 and q2
5  //        code is not shown
}
```

clear

```
template <class T>
void Queue2<T>::clear (void)
   //! clears self
{
   s.clear();
} // clear
```

- The job of the *clear* operation is to reset the value of the variable back to its initial value
- For Queue layered on List:
  - The spec for *clear* is:
    `clears self`
  - Recall the *correspondence*:
    `self = s.left * s.right`
  - Queue's *clear* has to only call through to List's *clear* for the data member *s*:

    After calling `s.clear();`
    `s = (<>,<>)`

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

  - 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 List's *clear*

transferFrom

```
template <class T>
void Queue2<T>::transferFrom (Queue2& source)
   //! replaces self
   //! clears source
   //! ensures: self = #source
{
   s.transferFrom(source.s);
}
```

- The job of *transferFrom* is to transfer the value from the source variable to the controlling variable, i.e., the variable front of the dot
- For Queue layered on List:
  - This is easily accomplished by calling through to List's *transferFrom*
    - Queue2's *transferFrom* has parameter `source`
      - It is of type Queue2
      - List's *transferFrom* cannot be called as follows: `s.tranferFrom(source);`

        This would be a type mismatch because List's *transferFrom* works on two Lists

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

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

transferFrom works on two Lists:

1. *s* in front of the dot
2. source.s passed in as a parameter

operator =

```
template <class T>
Queue2<T>& Queue2<T>::operator = (Queue2& rhs)
    //! replaces self
    //! restores rhs
    //! ensures: self = rhs
{
    s = rhs.s;
    return *this;
} // operator =
```

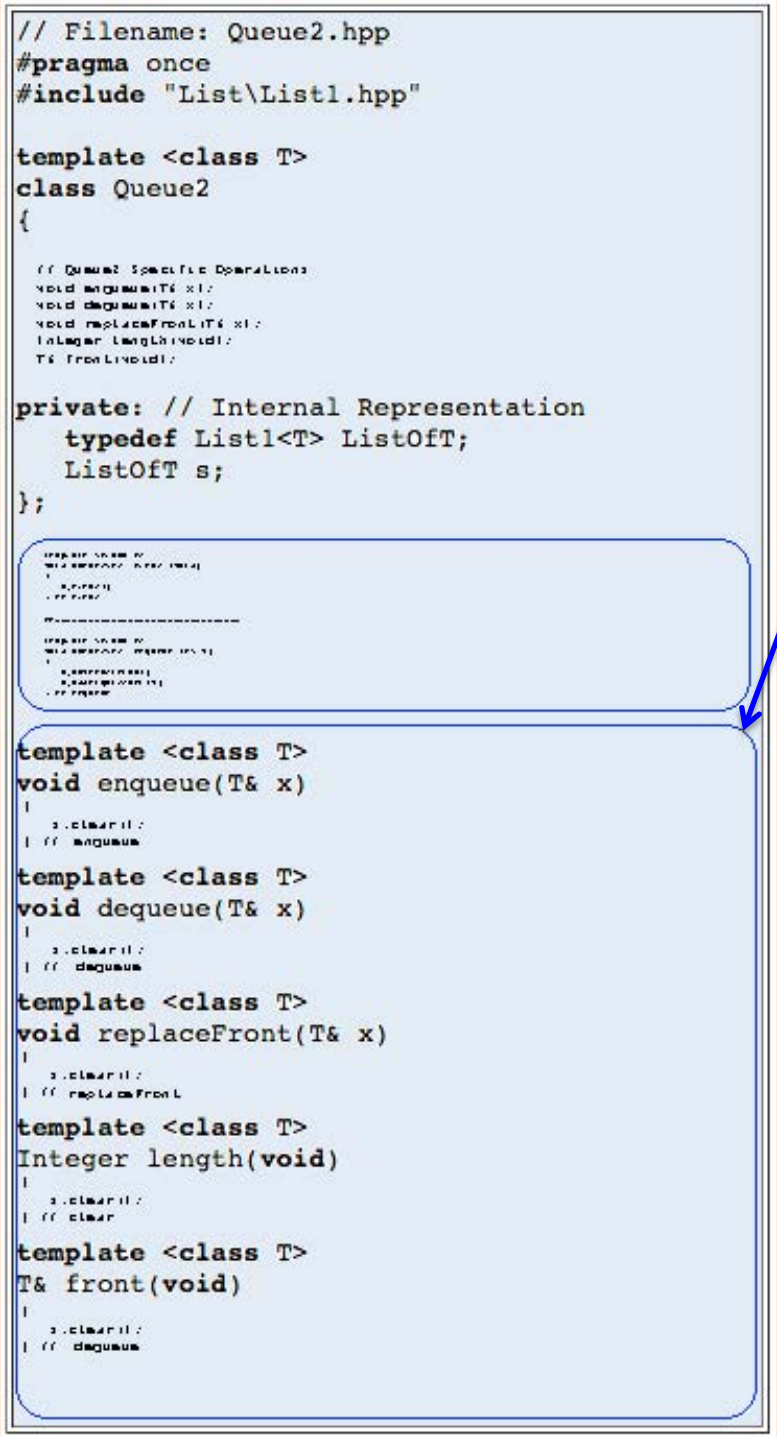
- In C++ *operator* = is the assignment operator, and its job is to make a copy of the variable that appears on the right-hand side (rhs) of the equals sign (=) and place the copy in the variable on the left-hand side (lhs)

Example client:

```
{
1  typedef Queue2<Integer> IntegerQueue;
2  IntegerQueue q1, q2;
3
4  q2 = q1;
5  // Or in C++ we could have written:
6  q2.operator=(q1);
7  // Both do the same thing, and both compile
}
```

- Queue's layered *operator* = is implemented by:
  - By calling through to List's *operator* =
    - Again, we have to use C++'s dot operator to gain access to the List 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;`



Component Specific Operations

- Queue's component specific operations are:

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

enqueue

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

Example client:

```
{
1   typedef Queue2<Integer> IntegerQueue;
2   IntegerQueue q1;
3   Integer y2;
4
5   // Code not shown - enqueues 3 items onto q1
6   // Incoming: q1 = <18,15,27>  y2 = 5
7   q1.enqueue(y2);
8   // Outgoing: q1 = <18,15,27,5>  y2 = 0
}
```

- *enqueue*'s ensures clause after substitution is:
  `q1 = #q1 * <#y2> = <18,15,27> * <5>`
  `= <18,15,27,5>`

- What value does *s* (inside q1) contain after *enqueue*?
  Answer: `s = (<18,15,27>,<5>)`

- How?
  `s.moveToFinish();` gave *s* the value:
  `s = (<18,15,27>,<>)`
  `s.addRightFront(x);` gave *s* the value:
  `s = (<18,15,27>,<5>)`

- And using the *correspondence* we get:
  `self = s.left * s.right`
  `= <18,15,27> * <5>`
  `= <18,15,27,5>`

dequeue

```
template <class T>
void dequeue(T& x)
   //! updates self
   //! replaces x
   //! requires: self /= <>
   //!  ensures: <x> = #self[0,1) and
   //!           self = #self[1, |#self|)
{
   s.moveToStart();
   s.removeRightFront(x);
} // dequeue
```

Example client:

```
{
1   typedef Queue2<Integer> IntegerQueue;
2   IntegerQueue q1;
3   Integer y2;
4
5   // Incoming: q1 = <18,15,27>  y2 = 100
6   q1.dequeue(y2);
7   // Outgoing: q1 = <15,27>  y2 = 18
}
```

- *dequeue*'s ensures clause after substitution is:
  `<y2> = <18,15,27>[0,1) and`
  `q1 = <18,15,27>[1,3)`
  `= <15,27>`

- What value does *s* (inside q1) contain after *dequeue*?
  Answer: `s = (<>,<15,27>)`

- How?
  `s.moveToStart();` gave *s* the value:
  `s = (<>,<18,15,27>)`
  `s.removeRightFront(x);` gave *s* the value:
  `s = (<>,<15,27>)`

- And using the *correspondence* we get:
  `self = s.left * s.right`
  `= <> * <15,27>`
  `= <15,27>`

replaceFront

```
template <class T>
void replaceFront(T& x)
   //! updates self, x
   //! requires: self /= < >
   //!  ensures: <x>  = #self[0,1) and
   //!           self = <#x> * #self[1, |#self|)
{
   s.moveToStart();
   s.replaceRightFront(x);
} // replaceFront
```

Example client:

```
{
1   typedef Queue2<Integer> IntegerQueue;
2   IntegerQueue q1;
3   Integer y2;
4
5   // Incoming: q1 = <18,15,27>  y2 = 100
6   q1.replaceFront(y2);
7   // Outgoing: q1 = <100,15,27>  y2 = 18
}
```

- *replaceFront*'s ensures clause after substitution is:

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

- How?

  `s.moveToStart();` gave *s* the value:
  `s = (<>,<18,15,27>)`
  `s.replaceRightFront(x);` gave *s* the value:
  `s = (<>,<100,15,27>)`

- And using the *correspondence* we get:

  `self = s.left * s.right`
  `= <> * <100,15,27>`
  `= <100,15,27>`

front

```
template <class T>
T& front(void)
   //! restores self
   //! requires: self /= < >
   //!  ensures: <front> = self[0,1)
{
   s.moveToStart();
   return s.rightFront();
} // front
```

Example client:

```
{
1  typedef Queue2<Integer> IntegerQueue;
2  IntegerQueue q1;
3
4  // Incoming: q1 = <111,44>
5  cout << q1.front();
6  // Outgoing: q1 = <111,44> and 111 is output
}
```

- *front*'s ensures clause after substitution is:

  `<front> = <111,44>[0,1)`

  `<front> = <111>`

  `front = 111`

- Hand executing *front*'s code:

  `s.moveToStart();` gave *s* the value:

  `s = (<>,<111,44>)`

  `s.rightFront();` returns a reference to 111

length

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

Example client:

```
{
1   typedef Queue2<Integer> IntegerQueue;
2   IntegerQueue q1;
3   Integer z;
4
5   // Incoming: q1 = <18,15,27>  z = 0
6   z = q1.length();
7   // Outgoing: q1 = <18,15,27>  z = 3
}
```

- *length*'s ensures clause after substitution is:

  ```
  length = |<18,15,27>|
         = 3
  ```

- Because of the *correspondence* we know:

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

  and we know:

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

  so adding |s.left| and |s.right| must give us 3:

  ```
  s.leftLength() + s.rightLength() = 3
  ```

- In this example *s* could have anyone of 4 values:

  1. s = (<>,<18,15,27>)
  2. s = (<18>,<15,27>)
  3. s = (<18,15>,<27>)
  4. s = (<18,15,27>,<>)