Queue

front

Inspecting the front of a Queue

One of the 5 Queue Specific Operations

The Queue Component

```
template <class T>
class Queue1

{

public: // Standard Operations

   Queue1();

   ~Queue1();

   void clear (void);

   void transferFrom (Queue1& source);

   Queue1& operator = (Queue1& rhs);

// Queue1 Specific Operations

   void enqueue (T& x);

   void dequeue (T& x);

   void replaceFront (T& x);

   T& front (void);

   Integer length (void);

private: // representation

   // ...

};
```

Let’s look at the *front* operation

Many C++ *container* components have an operation that allows the client to examine some part of the data stored in the container, for Queue this operation is *front*

```
template <class T>
class Queue1

{

public: // Standard Operations

   Queue1();

   ~Queue1();

   void clear (void);

   void transferFrom (Queue1& source);

   Queue1& operator = (Queue1& rhs);

// Queue1 Specific Operations

   void enqueue (T& x);

   void dequeue (T& x);

   void replaceFront (T& x);

   T& front (void);
      //! restores self
      //! requires: self /= <>
      //! ensures:
      //! <front> is prefix of self

   Integer length (void);

private: // representation

   // ...

};
```

front

The job of *front* is to return a reference to the value stored at the front of the queue

front

```
template <class T>
class Queue1

{

public: // Standard Operations

   Queue1();

   ~Queue1();

   void clear (void);

   void transferFrom (Queue1& source);

   Queue1& operator = (Queue1& rhs);

// Queue1 Specific Operations

   void enqueue (T& x);

   void dequeue (T& x);

   void replaceFront (T& x);

   T& front (void);
      //! restores self
      //! requires: self /= <>
      //! ensures:
      //! <front> is prefix of self

   Integer length (void);

private: // representation

   // ...

};
```

front's ensures clause indicates:

- That the reference returned is the value stored at the front of #self (the incoming queue)

front

```
template <class T>
class Queue1

{

public: // Standard Operations

   Queue1();

   ~Queue1();

   void clear (void);

   void transferFrom (Queue1& source);

   Queue1& operator = (Queue1& rhs);

// Queue1 Specific Operations

   void enqueue (T& x);

   void dequeue (T& x);

   void replaceFront (T& x);

   T& front (void);
      //! restores self
      //! requires: self /= <>
      //! ensures:
      //! <front> is prefix of self

   Integer length (void);

private: // representation

   // ...

};
```

restores self

- Is concise notation for: *self = #self*
- Without this concise notation, the ensures clause would be written as follows:

  ensures: <front> is prefix of self and
  self = #self

front

```
template <class T>
class Queue1

{

public: // Standard Operations

   Queue1();

   ~Queue1();

   void clear (void);

   void transferFrom (Queue1& source);

   Queue1& operator = (Queue1& rhs);

// Queue1 Specific Operations

   void enqueue (T& x);

   void dequeue (T& x);

   void replaceFront (T& x);

   T& front (void);
      //! restores self
      //! requires: self /= <>
      //! ensures:
      //! <front> is prefix of self

   Integer length (void);

private: // representation

   // ...

};
```

front is called in the client below and the lines following the call contain comments based on *front*'s spec

Example client:

```
{
 typedef Queue1<Integer> IntegerQueue;
 IntegerQueue q1;

 // ...
 // Suppose q1 = <5,10,15>
 cout << q1.front();
 // <front> is prefix of self
 // self = #self
}
```

front

```
template <class T>
class Queue1

{

public: // Standard Operations

   Queue1();

   ~Queue1();

   void clear (void);

   void transferFrom (Queue1& source);

   Queue1& operator = (Queue1& rhs);

// Queue1 Specific Operations

   void enqueue (T& x);

   void dequeue (T& x);

   void replaceFront (T& x);

   T& front (void);
      //! restores self
      //! requires: self /= <>
      //! ensures:
      //! <front> is prefix of self

   Integer length (void);

private: // representation

   // ...

};
```

Substitute:

- q1 for *self*

This gives us

Example client:

```
{
 typedef Queue1<Integer> IntegerQueue;
 IntegerQueue q1;

 // ...
 // Suppose q1 = <5,10,15>
 wcout << q1.front();
 // <front> is prefix of q1
 // q1 = #q1
}
```

front

```
template <class T>
class Queue1
{
public: // Standard Operations
   Queue1();
   ~Queue1();
   void clear (void);
   void transferFrom (Queue1& source);
   Queue1& operator = (Queue1& rhs);
// Queue1 Specific Operations
   void enqueue (T& x);
   void dequeue (T& x);
   void replaceFront (T& x);
   T& front (void);
      //! restores self
      //! requires: self /= <>
      //! ensures:
      //! <front> is prefix of self
   Integer length (void);
private: // representation
   // ...
};
```

Now substitute:

- <5,10,15> for #q1

This gives us

Example client:

```
{
 typedef Queue1<Integer> IntegerQueue;
 IntegerQueue q1;

 // ...
 // Suppose q1 = <5,10,15>
 wcout << q1.front();
 // <front> is prefix of <5,10,15>
 // q1 = <5,10,15>
}
```

```
template <class T>
class Queue1

{

public: // Standard Operations

   Queue1();

   ~Queue1();

   void clear (void);

   void transferFrom (Queue1& source);

   Queue1& operator = (Queue1& rhs);

// Queue1 Specific Operations

   void enqueue (T& x);

   void dequeue (T& x);

   void replaceFront (T& x);

   T& front (void);
      //! restores self
      //! requires: self /= <>
      //! ensures:
      //! <front> is prefix of self

   Integer length (void);

private: // representation

   // ...

};
```

front

Evaluate: <front> is prefix of <5,10,15>

front returns a reference to: 5

wcout outputs 5
And q1's value remains unchanged

Example client:

```
{
 typedef Queue1<Integer> IntegerQueue;
 IntegerQueue q1;

 // ...
 // Suppose q1 = <5,10,15>
 wcout << q1.front();
 // <front> is prefix of <5,10,15>
 // q1 = <5,10,15>
}
```

front

```
template <class T>
class Queue1

{

public: // Standard Operations

   Queue1();

   ~Queue1();

   void clear (void);

   void transferFrom (Queue1& source);

   Queue1& operator = (Queue1& rhs);

// Queue1 Specific Operations

   void enqueue (T& x);

   void dequeue (T& x);

   void replaceFront (T& x);

   T& front (void);
      //! restores self
      //! requires: self /= <>
      //! ensures:
      //! <front> is prefix of self

   Integer length (void);

private: // representation

   // ...

};
```

Now examine *front*’s requires clause

front

```
template <class T>
class Queue1
{
public: // Standard Operations
   Queue1();
   ~Queue1();
   void clear (void);
   void transferFrom (Queue1& source);
   Queue1& operator = (Queue1& rhs);
// Queue1 Specific Operations
   void enqueue (T& x);
   void dequeue (T& x);
   void replaceFront (T& x);
   T& front (void);
      //! restores self
      //! requires: self /= <>
      //! ensures:
      //! <front> is prefix of self
   Integer length (void);
private: // representation
   // ...
};
```

front’s requires clause indicates the incoming value of *self* must not be empty

We must check the client’s call to *front* to make sure it satisfies *front*’s requires clause

Example client:

```
{
  typedef Queue1<Integer> IntegerQueue;
  IntegerQueue q1;

  // ...
  // Suppose q1 = <5,10,15>
  wcout << q1.front();
}
```

front

```
template <class T>
class Queue1
{
public: // Standard Operations
   Queue1();
   ~Queue1();
   void clear (void);
   void transferFrom (Queue1& source);
   Queue1& operator = (Queue1& rhs);
// Queue1 Specific Operations
   void enqueue (T& x);
   void dequeue (T& x);
   void replaceFront (T& x);
   T& front (void);
      //! restores self
      //! requires: self /= <>
      //! ensures:
      //! <front> is prefix of self
   Integer length (void);
private: // representation
   // ...
};
```

In the client below a comment containing the *front*'s requires clause has been inserted prior to the call to *front*

Example client:

```
{
  typedef Queue1<Integer> IntegerQueue;
  IntegerQueue q1;

  // ...
  // Suppose q1 = <5,10,15>
  // self /= <>
  wcout << q1.front();
}
```

```
template <class T>
class Queue1
{
public: // Standard Operations
   Queue1();
   ~Queue1();
   void clear (void);
   void transferFrom (Queue1& source);
   Queue1& operator = (Queue1& rhs);
// Queue1 Specific Operations
   void enqueue (T& x);
   void dequeue (T& x);
   void replaceFront (T& x);
   T& front (void);
      //! restores self
      //! requires: self /= <>
      //! ensures:
      //! <front> is prefix of self
   Integer length (void);
private: // representation
   // ...
};
```

front

Substitute:

- q1 for *self*

This gives us

Example client:

```
{
typedef Queue1<Integer> IntegerQueue;
IntegerQueue q1;

// ...
// Suppose q1 = <5,10,15>
// q1 /= <>
wcout << q1.front();
}
```

```
template <class T>
class Queue1
{
public: // Standard Operations
   Queue1();
   ~Queue1();
   void clear (void);
   void transferFrom (Queue1& source);
   Queue1& operator = (Queue1& rhs);
// Queue1 Specific Operations
   void enqueue (T& x);
   void dequeue (T& x);
   void replaceFront (T& x);
   T& front (void);
      //! restores self
      //! requires: self /= <>
      //! ensures:
      //! <front> is prefix of self
   Integer length (void);
private: // representation
   // ...
};
```

front

Now substitute:

- <5,10,15> for q1

This gives us

front's requires clause allows us to reason that the incoming queue q1 is not empty

Example client:

```
{
  typedef Queue1<Integer> IntegerQueue;
  IntegerQueue q1;
  Integer y2;
  // ...
  // Suppose q1 = <5,10,15>
  // <5,10,15> /= <>
  wcout << q1.front();
}
```