16.5 Implementing the STL `list`

This section implements the STL `list` class discussed in Section 7.6. We will see lots of code, but most of the techniques were seen earlier in this chapter.

```
1  // Incomplete class declarations for
2  // the const_iterator, iterator, and list,
3  // because all these classes refer to each other.
4  template <class Object>
5  class ConstListItr;
6
7  template <class Object>
8  class ListItr;
9
10 template <class Object>
11 class list;
12
13 // The basic doubly-linked list node.
14 // Everything is private, and is accessible
15 // only by the iterators and list classes.
16 template <class Object>
17 class ListNode
18 {
19     Object data;
20     ListNode *prev;
21     ListNode *next;
22
23     ListNode( const Object & d = Object( ),
24               ListNode * p = NULL, ListNode * n = NULL )
25       : data( d ), prev( p ), next( n ) { }
26
27     friend class ConstListItr<Object>;
28     friend class ListItr<Object>;
29     friend class list<Object>;
30 };
```

Figure 16.23 `ListNode` class and declarations for other classes

```
 1 template <class Object>
 2 class list
 3 {
 4   public:
 5     typedef ListItr<Object>      iterator;
 6     typedef ConstListItr<Object> const_iterator;
 7
 8     list( );
 9     ~list( );
10
11     list( const list & rhs );
12     const list & operator= ( const list & rhs );
13
14     iterator begin( );
15     const_iterator begin( ) const;
16
17     iterator end( );
18     const_iterator end( ) const;
19
20     int size( ) const;
21     bool empty( ) const;
22
23     Object & front( );
24     const Object & front( ) const;
25
26     Object & back( );
27     const Object & back( ) const;
28
29     void push_front( const Object & x );
30     void push_back( const Object & x );
31     void pop_front( );
32     void pop_back( );
33
34     iterator insert( iterator itr, const Object & x );
35     iterator erase( iterator itr );
36     iterator erase( iterator start, iterator end );
37
38     friend class ConstListItr<Object>;
39     friend class ListItr<Object>;
40
41   private:
42     typedef ListNode<Object> node;
43
44     int   theSize;
45     node *head;
46     node *tail;
47
48     void init( );
49     void makeEmpty( );
50 };
```

Figure 16.24 `list` class interface

Our code already uses many pages, so to save some space, we do not show all the include directives, and we occasionally skimp on the commenting.

As we saw previously, we will need a class to store the basic list node, a class for the iterator, and a class for the list itself. The STL provides two types of iterators: the `const_iterator` and `iterator`, so we will have two iterator classes, for a grand total of four classes. As we will see, `const_iterator` and `iterator` will be typedefs representing the two iterator class templates. The technique was seen in the `vector` implementation in Figure 7.10. Because the iterators are bidirectional, we will need a doubly-linked list. Finally, our implementation will be safer than the STL's in that the use of bad iterators (for instance, advancing past the endmarker) will cause an exception to be thrown.

The four classes are: `list`, `ListItr`, `ConstListItr`, and `ListNode`. We begin by discussing `ListNode`. Then we look at `list`, and finally we look at the two iterator classes.

Figure 16.23 (page 838) shows the `ListNode` class. This is very similar to the `LListNode` class that we wrote in Figure 16.7. The main difference is that since we use a doubly linked list, we have both `prev` and `next` pointers. As we saw before, we use the technique of making all the members private, and then making the three other class friends. And so we also see the technique of using an incomplete class declaration.

Next we see the `list` class interface in Figure 16.24. As advertised, lines 5 and 6 are typedefs for the `iterator` and `const_iterator`. At line 42, we use another typedef, simply for convenience. When writing member functions, instead of using `ListNode<Object>`, we can now use `node`. We see this immediately in the declaration of `head` and `tail` at lines 45 and 46. Notice also that the list class keeps track of its size in a data member declared at line 44. We do this so that the `size` method can be performed in constant time.

Almost all of the member functions use signatures that we've seen before. So for instance, there are two versions of `begin` and `end`, just like in the `vector` class in Figure 7.10.

Some unusual lines of code occur at lines 34 to 39. The three member functions (`insert` and both `erase` methods) pass an iterator using call by value instead of by the normal (for non-primitive objects) constant reference. This is safe, because as we will see, the iterator is a small object. Also unusual is that the friend declarations do not use `const_iterator` and `iterator` directly. One of our compilers didn't like it, so we have played it safe. This illustrates the typical C++ problem of combining too many features at the same time. Often, you will run into compiler bugs.

The implementation of `list` begins in Figure 16.25 (page 838), where we see a constructor and the big three. `makeEmpty` and `init` are private helpers. `init` contains the basic functionality of the zero-parameter constructor, but is a separate function so that the copy constructor can be implemented by logically using a zero-parameter constructor and `operator=`. All in all, there is little new here. We combine a lot of the `LList` code with concepts in Section 16.3.

```
 1 template <class Object>
 2 list<Object>::list( )
 3 {
 4     init( );
 5 }
 6
 7 template <class Object>
 8 void list<Object>::init( )
 9 {
10     theSize = 0;
11     head = new node;
12     tail = new node;
13     head->next = tail;
14     tail->prev = head;
15 }
16
17 template <class Object>
18 list<Object>::~list( )
19 {
20     makeEmpty( );
21     delete head;
22     delete tail;
23 }
24
25 template <class Object>
26 void list<Object>::makeEmpty( )
27 {
28     while( !empty( ) )
29         pop_front( );
30 }
31
32 template <class Object>
33 list<Object>::list( const list<Object> & rhs )
34 {
35     init( );
36     *this = rhs;
37 }
38
39 template <class Object>
40 const list<Object> &
41 list<Object>::operator= ( const list & rhs )
42 {
43     if( this == & rhs )
44         return *this;
45
46     makeEmpty( );
47     const_iterator itr = rhs.begin( );
48     while( itr != rhs.end( ) )
49         push_back( *itr++ );
50     return *this;
51 }
```

Figure 16.25 Constructor and big three for `list` class

```
 1 // Return iterator representing beginning of list.
 2 // Mutator version is first, then accessor version.
 3 template <class Object>
 4 list<Object>::iterator list<Object>::begin( )
 5 {
 6     iterator itr( *this, head );
 7     return ++itr;
 8 }
 9
10 template <class Object>
11 list<Object>::const_iterator list<Object>::begin( ) const
12 {
13     const_iterator itr( *this, head );
14     return ++itr;
15 }
16
17 // Return iterator representing endmarker of list.
18 // Mutator version is first, then accessor version.
19 template <class Object>
20 list<Object>::iterator list<Object>::end( )
21 {
22     return iterator( *this, tail );
23 }
24
25 template <class Object>
26 list<Object>::const_iterator list<Object>::end( ) const
27 {
28     return const_iterator( *this, tail );
29 }
30
31 // Return number of elements currently in the list.
32 template <class Object>
33 int list<Object>::size( ) const
34 {
35     return theSize;
36 }
37
38 // Return true if the list is empty, false otherwise.
39 template <class Object>
40 bool list<Object>::empty( ) const
41 {
42     return size( ) == 0;
43 }
```

Figure 16.26 `begin`, `end`, `size`, and `empty` for `list` class

Figure 16.26 (page 842) contains methods `begin`, `end`, `size`, and `empty`. `begin` looks much like the `LList` method `zeroth` (Figure 16.10), except that the iterators are constructed by passing not only a pointer to a node, but also a reference to the list that contains the node. This allows additional error checking for the `insert` and `erase` methods. However, `zeroth` returns the header, but we

want the first node, so we advance the iterator with operator++, and use its new value as the return value. end, size, and empty are one-liners.

Figure 16.27 (page 844) contains the double-ended queue operations. All are one-liners that combine calls to begin, end, operator*, operator--, insert, and erase. Recall that the insert method inserts prior to a position. So, push_back inserts prior to the endmarker, as required. In pop_back, note that erase(--end()) creates an iterator corresponding to the endmarker, retreats, and uses that iterator to erase. Similar behavior is seen in back.

insert and erase are shown in Figure 16.28. assertIsValid, called at line 6 throws an exception if itr is not at an insertable location. This could occur if it was never initialized. At line 7 we test that itr belongs to this list, and at line 8 we throw an exception if it does not. The rest of the code is the usual splicing already discussed for doubly linked list. At line 14, an iterator representing the newly inserted item is returned.

In the first version of erase, we see an additional error check. Afterwards, we perform the standard deletion in a doubly linked list; we return an iterator representing the item after the deleted element. Like insert, erase must update theSize. The second version of erase simply uses an iterator to call the first version of erase. Note carefully, that we cannot simply use itr++ in the for loop at line 41 and ignore the return value of erase at line 42. The value of itr is stale immediately after the call to erase, and this is why erase returns an iterator.

The value of itr is stale immediately after the call to erase, and this is why erase returns an iterator.

Figure 16.29 has the class interface for ConstListItr and ListItr. The iterators store the current position and a pointer to the header. Of most interest is the use of inheritance. We want to be able to send an iterator to any method that accepts a const_iterator, but not vice-versa. So an iterator IS-A const_iterator. As a result, the private assert methods only need to be written once for the two classes. At line 6, the base class destructor is declared virtual, as is normal for base classes. operator* is also declared virtual.

However, operator++ and operator-- are not virtual, mostly because their return types change. The versions that do not return references cannot be virtual, while those that return references can be virtual because they have compatible return types (the return type changes from a reference to a base class to a reference to a derived class Section 4.4.4). However, our compilers did not agree with the newer rules.

The iterator classes each declare operator++ and operator-- to mirror advance and retreat (see Section 2.3.3 for a discussion of operator overloading). The public comparison operators and the private helper retrieve are declared in const_iterator and inherited unchanged.

```
// front, back, push_front, push_back, pop_front, and pop_back
// are the basic double-ended queue operations.
template <class Object>
Object & list<Object>::front( )
{
    return *begin( );
}

template <class Object>
const Object & list<Object>::front( ) const
{
    return *begin( );
}

template <class Object>
Object & list<Object>::back( )
{
    return *--end( );
}

template <class Object>
const Object & list<Object>::back( ) const
{
    return *--end( );
}

template <class Object>
void list<Object>::push_front( const Object & x )
{
    insert( begin( ), x );
}

template <class Object>
void list<Object>::push_back( const Object & x )
{
    insert( end( ), x );
}

template <class Object>
void list<Object>::pop_front( )
{
    erase( begin( ) );
}

template <class Object>
void list<Object>::pop_back( )
{
    erase( --end( ) );
}
```

Figure 16.27 Double-ended queue `list` operations

```
// Insert x before itr.
template <class Object>
list<Object>::iterator
list<Object>::insert( iterator itr, const Object & x )
{
    itr.assertIsValid( );
    if( itr.head != head )  // itr is not in this list
        throw IteratorMismatchException( );

    node *p = itr.current;
    p->prev->next = new node( x, p->prev, p );
    p->prev = p->prev->next;
    theSize++;
    return iterator( *this, p->prev );
}

// Erase item at itr.
template <class Object>
list<Object>::iterator list<Object>::erase( iterator itr )
{
    itr.assertIsValid( );
    if( itr == end( ) )     // can't erase endmarker
        throw IteratorOutOfBoundsException( );
    if( itr.head != head )  // itr is not in this list
        throw IteratorMismatchException( );

    node *p = itr.current;
    iterator retVal( *this, p->next );
    p->prev->next = p->next;
    p->next->prev = p->prev;
    delete p;
    theSize--;
    return retVal;
}

// Erase items in the range [from,to).
template <class Object>
list<Object>::iterator
list<Object>::erase( iterator from, iterator to )
{
    for( iterator itr = from; itr != to; )
         itr = erase( itr );
    return to;
}
```

Figure 16.28 Methods for insertion and removal from the `list`

```
 1 template <class Object>
 2 class ConstListItr
 3 {
 4   public:
 5     ConstListItr( );
 6     virtual ~ConstListItr( ) { }
 7
 8     virtual const Object & operator* ( ) const;
 9     ConstListItr & operator++ ( );
10     ConstListItr operator++ ( int );
11     ConstListItr & operator-- ( );
12     ConstListItr operator-- ( int );
13
14     bool operator== ( const ConstListItr & rhs ) const;
15     bool operator!= ( const ConstListItr & rhs ) const;
16
17   protected:
18     typedef ListNode<Object> node;
19     node *head;
20     node *current;
21
22     friend class list<Object>;
23     void assertIsInitialized( ) const;
24     void assertIsValid( ) const;
25     void assertCanAdvance( ) const;
26     void assertCanRetreat( ) const;
27     Object & retrieve( ) const;
28
29     ConstListItr( const list<Object> & source, node *p );
30 };
31
32 template <class Object>
33 class ListItr : public ConstListItr<Object>
34 {
35   public:
36     ListItr( );
37
38     Object & operator* ( );
39     const Object & operator* ( ) const;
40     ListItr & operator++ ( );
41     ListItr operator++ ( int );
42     ListItr & operator-- ( );
43     ListItr operator-- ( int );
44
45   protected:
46     typedef ListNode<Object> node;
47     friend class list<Object>;
48
49     ListItr( const list<Object> & source, node *p );
50 };
```

Figure 16.29 Class interface for two `list` iterators

```
// Public constructor for const_iterator.
template <class Object>
ConstListItr<Object>::ConstListItr( )
  : head( NULL ), current( NULL )
{
}

// Protected constructor for const_iterator.
// Expects the list that owns the iterator and a
// pointer that represents the current position.
template <class Object>
ConstListItr<Object>::
ConstListItr( const list<Object> & source, node *p )
  : head( source.head ), current( p )
{
}

// Public constructor for iterator.
// Calls the base-class constructor.
// Must be provided because the private constructor
// is written; otherwise zero-parameter constructor
// would be disabled.
template <class Object>
ListItr<Object>::ListItr( )
{
}

// Protected constructor for iterator.
// Expects the list that owns the iterator and a
// pointer that represents the current position.
template <class Object>
ListItr<Object>::
ListItr( const list<Object> & source, node *p )
  : ConstListItr<Object>( source, p )
{
}
```

Figure 16.30 Constructors for `list` iterators

The iterator constructors are shown in Figure 16.30. They are straightforward. As we mentioned earlier, the zero-parameter constructor is public, while the two parameter constructor, that sets the current position and the header position, is private. Various assertion methods are shown in Figure 16.31. These all test the validity of an iterator and throw an exception if the iterator is determined to be invalid. Otherwise, these methods return safely.

```
// Throws an exception if this iterator is obviously
// uninitialized. Otherwise, returns safely.
template <class Object>
void ConstListItr<Object>::assertIsInitialized( ) const
{
    if( head == NULL || current == NULL )
        throw IteratorUninitializedException( );
}

// Throws an exception if the current position is
// not somewhere in the range from begin to end, inclusive.
// Otherwise, returns safely.
template <class Object>
void ConstListItr<Object>::assertIsValid( ) const
{
    assertIsInitialized( );
    if( current == head )
        throw IteratorOutOfBoundsException( );
}

// Throws an exception if operator++ cannot be safely applied
// to the current position. Otherwise, returns safely.
template <class Object>
void ConstListItr<Object>::assertCanAdvance( ) const
{
    assertIsInitialized( );
    if( current->next == NULL )
        throw IteratorOutOfBoundsException( );
}

// Throws an exception if operator-- cannot be safely applied
// to the current position. Otherwise, returns safely.
template <class Object>
void ConstListItr<Object>::assertCanRetreat( ) const
{
    assertIsValid( );
    if( current->prev == head )
        throw IteratorOutOfBoundsException( );
}
```

Figure 16.31 Various assertions that throw exceptions if the assertion fails

In Figure 16.32, we see three versions of `operator*`, which is used to get the `Object` stored at the current position. Recall that we have an accessor method that returns a constant reference, and a mutator method that returns a reference (through which the `Object` can be changed). The mutator method cannot be made available for `const_iterator`, so we have only three methods. All are identical except for the return type, and simply call the `retrieve` method.

```
// Return the object stored at the current position.
// For const_iterator, this is an accessor with a
// const reference return type.
template <class Object>
const Object & ConstListItr<Object>::operator* ( ) const
{
    return retrieve( );
}

// Return the object stored at the current position.
// For iterator, there is an accessor with a
// const reference return type and a mutator with
// a reference return type. The accessor is shown first.
template <class Object>
const Object & ListItr<Object>::operator* ( ) const
{
    return ConstListItr<Object>::operator*( );
}

template <class Object>
Object & ListItr<Object>::operator* ( )
{
    return retrieve( );
}

// Protected helper in const_iterator that returns the object
// stored at the current position. Can be called by all
// three versions of operator* without any type conversions.
template <class Object>
Object & ConstListItr<Object>::retrieve( ) const
{
    assertIsValid( );
    if( current->next == NULL )
        throw IteratorOutOfBoundsException( );

    return current->data;
}
```

Figure 16.32 Various `operator*` implementations

```
 1 template <class Object>   // prefix
 2 ConstListItr<Object> & ConstListItr<Object>::operator++ ( )
 3 {
 4     assertCanAdvance( );
 5     current = current->next;
 6     return *this;
 7 }
 8
 9 template <class Object>   // postfix
10 ConstListItr<Object> ConstListItr<Object>::operator++ ( int )
11 {
12     ConstListItr<Object> old = *this;
13     ++( *this );
14     return old;
15 }
16
17 template <class Object>   // prefix
18 ListItr<Object> & ListItr<Object>::operator++ ( )
19 {
20     assertCanAdvance( );
21     current = current->next;
22     return *this;
23 }
24
25 template <class Object>   // postfix
26 ListItr<Object> ListItr<Object>::operator++ ( int )
27 {
28     ListItr<Object> old = *this;
29     ++( *this );
30     return old;
31 }
32
33 template <class Object>   // prefix
34 ConstListItr<Object> & ConstListItr<Object>::operator-- ( )
35 {
36     assertCanRetreat( );
37     current = current->prev;
38     return *this;
39 }
40
41 template <class Object>   // postfix
42 ConstListItr<Object> ConstListItr<Object>::operator-- ( int )
43 {
44     ConstListItr<Object> old = *this;
45     --( *this );
46     return old;
47 }
```

Figure 16.33 Four versions of `operator++` (two for each iterator), and two versions of operator-- (for `ConstListItr`); two additional versions of `operator--` are similar and not shown

```
 1 template <class Object>
 2 bool ConstListItr<Object>::
 3 operator== ( const ConstListItr & rhs ) const
 4 {
 5     return current == rhs.current;
 6 }
 7
 8 template <class Object>
 9 bool ConstListItr<Object>::
10 operator!= ( const ConstListItr & rhs ) const
11 {
12     return !( *this == rhs );
13 }
```

Figure 16.34 Equality operators for list iterators; these are inherited unchanged by `ListItr`

The various implementations of `operator++` are shown in In Figure 16.33 (page 850). The postfix version (`itr++`) is implemented in terms of the prefix version (`++itr`), the derived class versions are identical to the base class versions (except for class name), and `operator--` uses the same logic as `operator++`. Consequently, we omit two versions of `operator--`. Finally, Figure 16.34 shows a straightforward implementation of the equality operators.

All in all, there is a large amount of code, but it simply embellishes the basics seen in the original implementation of `LList` in Section 16.2.