CHAPTER 6

Lists

In this chapter we continue our study of the Standard Template Library's data structures by introducing another sequential container class: the list class. There are some significant performance differences between lists and vectors (or deques). For example, lists lack the random-access feature of vectors: to access a list's item from an index requires a loop that starts at the beginning or end of the list, whichever is closer to the index. But lists allow constant-time insertions and deletions, once an iterator is positioned where the insertion or deletion should be made. This fact makes iterators essential for almost all list applications, and furthermore, the list class lacks an index operator, **operator**[ ].

After we define what a list is, we enumerate some of the method interfaces for the list class and its associated iterator class. This *user's* view is all that the Standard Template Library specifies. We then provide an outline of the Hewlett-Packard design and implementation, and suggest simpler (but less efficient) designs. The application of lists, a simple line editor, takes advantage of a list's ability to quickly make consecutive insertions and deletions anywhere in the list. ■

CHAPTER OBJECTIVES

1. Understand the list class, both from the user's perspective and from the developer's perspective.
2. Given an application that requires a sequential container class, be able to decide whether a list, vector, or deque would be more appropriate.
3. Compare the Hewlett-Packard design of the list class with a singly linked design and a doubly linked design with head and tail fields.

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

6.1 | LISTS

In everyday life we construct lists as a way to impose order on reality: grocery lists, sign-up sheets, telephone directories, class rosters, TV schedules, and so on. For this reason, problems are often stated in terms of lists:

> Given a list of test scores, sort them into increasing order.
>
> Print out a list of all club members whose dues are overdue.

A ***list,*** sometimes called a ***linked list,*** is a finite sequence of items such that

1. Accessing or modifying an arbitrary item in the sequence takes linear time.
2. Given an iterator at a position in the sequence, inserting or deleting one item at that position takes constant time.

Starting in Section 6.1.1, we will design and implement a list class corresponding to this data structure. How do the two list properties compare with the behavior of vector objects? Recall that to access or modify the item in position k for a vector vec, we apply the index operator

```
vec [k]
```

The index operator is also used with a deque. With a list, we must use an iterator. Suppose lis is an instance of the list class, and we want to access the item k positions from the beginning of lis. We proceed sequentially either from the start of lis up to position k or from the end of lis down to position k, whichever route is shorter:

```
if (k < lis.size( ) / 2)
{
        // loop forward from beginning of lis:
        itr = lis.begin( );
        for (int i = 0; i < k; i++)
                itr++;
} // if
else
{
        // loop backward from end of lis:
        itr = lis.end( );
        for (int i = lis.size( ); i > k; i--)
                itr--;
} // else
```

The time for this access is proportional to k. Let n represent the number of items in the list. In the worst case, when k $= n$, the number of loop iterations is $n/2$, which is linear in n. The average distance from k to the beginning or end of the list is $n/4$, which is also linear in n.

The reason we lose the constant-time access we had for vectors and deques is that list iterators are not random-access iterators, but merely *bidirectional iterators.*

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

That means from a given position in the list, an iterator can go forward one position or go backward one position. Recall that, in constrast, a random-access iterator could immediately go forward or backward any number of positions.

But once an iterator is correctly positioned, an insertion or deletion at that position in a list takes only constant time, versus linear time for a vector or deque. This provides the major motivation for using a list instead of a vector (or deque): when the application entails a lot of insertions and/or deletions at positions other than the back (for a deque, front or back) of the container.

Lists can be spliced together in only constant time. For an example of what splicing means, suppose list1 contains the items

"television", "radio", "stereo", "CD player"

and the iterator itr is positioned at "radio". If list2 contains the items

"camcorder", "VCR", "laser disk player"

we can send the following message:

```
list1.splice (itr, list2);
```

As a result, the items from list2 will be removed from list2 and inserted into list1 in front of the item "radio". So list1 will contain

"television", "camcorder", "VCR", "laser disk player", "radio", "stereo", "CD player"

and list2 will be empty. You can probably see why splicing vectors or deques requires linear time proportional to the size of the container supplying the items to be spliced.

6.1.1 Method Interfaces for the list Class

The method interfaces for the list class and its associated iterator class are similar to those we saw for vectors and deques. We'll start with interfaces for the most widely used methods in the list class. Table 6.1 has a thumbnail sketch of these methods. The list class is a template class with template-parameter T, representing the type of the items in the list.

1.
```
// Postcondition: this list is empty.
list( );
```

Note This default constructor is usually invoked implicitly, for example,

```
list<Employee> employees;
```

makes employees an empty list, whose items will be of type Employee.

2.
```
// Postcondition: this list has been constructed and initialized to a copy of x.
//                The worstTime(n) is O(n), where n is the size of x.
list (const list<T>& x);
```

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

Table 6.1 | Brief description of some list methods (assume the following definition: list<double>::iterator itr;)

Method	Effect
list<**double**> x	x is an empty list
list<**double**> weights (x)	the list object weights contains a copy of the list object x
weights.push_front (8.3)	8.3 is inserted at the front of weights
weights.push_back (107.2)	107.2 is inserted at the back of weights
weights.insert (itr, 125.0)	125.0 is inserted where itr is positioned; items from the insertion point to the back of weights are moved up; returns an iterator positioned at the newly inserted item
weights.pop_front( )	The front item in weights has been deleted
weights.pop_back( )	The back item in weights has been deleted
weights.erase (itr)	The item where itr was positioned has been deleted; the only iterators and references invalidated are those positioned at the deleted item
weights.erase (itr1, itr2)	The items in weights, from where itr1 is positioned (inclusive) to where itr2 is positioned (exclusive) have been deleted
weights.size( )	Returns the number of items in weights
weights.empty( )	Returns **true** if weights has no items; otherwise, **false**
itr = weights.begin( )	itr is positioned at the item at the front of weights
itr == weight.end( )	Returns **true** if itr is positioned just beyond the back item in weights; otherwise, **false**
new_weights = weights	The previously defined list object new_weights contains a copy of weights
weights.splice (itr,old_weights)	All the items in old_weights are now in weights in front of where itr is positioned. The time for this method is constant no matter how many items were originally in weights or old_weights
weights.sort( )	The items in weights are in order according to **operator<**

Example Suppose that old_words, a list of strings, was defined earlier. If we write

```
list <string>new_words (old_words);
```

then new_words has been constructed and contains a copy of old_words.

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

Note Recall, from Chapter 5, that this kind of constructor is referred to as a ***copy constructor.***

3.
```
// Postcondition: x has been inserted at the front of this list.
void push_front (const T& x);
```

Note The vector class did not have a push_front method. Such a method might have given the impression that inserting at the front of a vector object was fast.

4.
```
// Postcondition: x has been inserted at the back of this list.
void push_back (const T& x);
```

5.
```
// Postcondition: x has been inserted in this list in front of the item that position
//                was positioned at before this call. An iterator positioned at x
//                has been returned.
iterator insert (iterator position, const T& x);
```

Note The worstTime(n) is constant. For the insert method in the vector class, worstTime(n) is $O(n)$.

6.
```
// Precondition: this list is not empty.
// Postcondition: the item that was at the front of this list before this call was
//                made has been deleted from this list.
void pop_front( );
```

7.
```
// Precondition: this list is not empty.
// Postcondition: the item that was at the back of this list before this call was
//                made has been deleted from this list.
void pop_back( );
```

8.
```
// Precondition: position is positioned at an item in this list.
// Postcondition: the item that position was positioned at before this call was
//                made has been deleted from this list.
void erase (iterator position);
```

Note The worstTime(n) is constant. For the erase method in the vector class, worstTime(n) is $O(n)$.

9.
```
// Precondition: first is positioned at some item in this list, and last is positioned
//               one past some item in this list.
```

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

```
// Postcondition: all the items that, before this call was made, were in the range
//                from first (inclusive) to last (exclusive) have been deleted from
//                this list.
void erase (iterator first, iterator last);
```

> ***Note*** The time for this method is proportional to the number of items removed. Recall that for the corresponding vector method, the time is proportional to the number of items *after* the last item removed (because those trailing items are moved down to fill up the deleted slots).

10.
```
// Postcondition: the number of items in this list has been returned.
unsigned size( ) const;
```

11.
```
// Postcondition: true has been returned if this list is empty. Otherwise, false
//                has been returned.
bool empty( ) const;
```

12.
```
// Postcondition: an iterator positioned at the front of this list has been returned.
iterator begin( );
```

13.
```
// Postcondition: an iterator positioned after the last item in this list has been
//                returned.
iterator end( );
```

> ***Note*** If the calling object list is empty, the iterator returned by the begin method is equal to the iterator returned by the end method.

14.
```
// Postcondition: this list contains a copy of x, and a reference to this list
//                has been returned.
list<T>& operator= (const list<T>& x);
```

> ***Note*** This assignment operator differs from the copy constructor (method number 2) in that the calling object for the copy constructor was being defined as well as being initialized to the parameter x.

15.
```
// Postcondition: The contents of x have been inserted, starting at position, into
//                this list, and x is empty.
void splice (iterator position, list<T>& x);
```

> ***Note*** This method takes constant time, no matter how big x is.

16.
```
// Precondition: operator< is defined for type T.
// Postcondition: the items in this list are in ascending order. The worstTime (n)
//                is O (n log n).
void sort( );
```

> ***Note*** We will study this method in Chapter 12.

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

There are also front and back methods, with the same method interfaces as the vector versions.

6.1.2 Iterator Interfaces

The list class supports bidirectional iterators, not random-access iterators. This is clearly seen by the absence of operator+. Here are the interfaces:

1.
```
// Postcondition: this iterator is now positioned at the next position in this list,
//                and a reference to this iterator has been returned.
iterator& operator++( );
```

> ***Note*** This is the *preincrement* operator; that is, the iterator advances and a reference to the newly positioned iterator is returned. For example, suppose that cities is a list object that contains the following list of cities:
>
> "Boston", "College Station", "Lansing", "Pasadena"
>
> If itr is a list iterator positioned at "College Station" and we write
>
> ```
> list<string>::iterator new_itr = ++itr;
> ```
>
> then both itr and new_itr are positioned at "Lansing".

2.
```
// Postcondition: this iterator is now positioned at the next position in this list,
//                and a copy of this iterator's previous value has been returned.
iterator operator++ (int)
```

> ***Note*** This is the *postincrement operator;* that is, the iterator advances, but the iterator's value before advancing is returned. The postincrement operator has an **int** parameter whose only purpose is to distinguish this operator from the preincrement operator. In fact, there is no argument corresponding to the **int** parameter. For example, suppose that cities is a list object that contains the following list of cities:
>
> "Boston", "College Station", "Lansing", "Pasadena"
>
> If itr is a list iterator positioned at "College Station" and we write
>
> ```
> list<string>::iterator old_itr = itr++;
> ```
>
> then itr is positioned at "Lansing", but old_itr is positioned at "College Station".

3.
```
// Postcondition: this iterator is now positioned at the previous position in this
//                list, and a reference to this iterator has been returned.
iterator& operator--( ); // pre-decrement
```

4.
```
// Postcondition: this iterator is now positioned at the previous position in this
//                list, and a copy of this iterator's previous value has been
```

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

```
//                  returned.
iterator operator--(int); // post-decrement
```

5.
```
// Precondition: this iterator is positioned at an item in this list.
// Postcondition: a reference to the item this iterator is positioned at has been
//                  returned.
T& operator*( );
```

Example Suppose that itr is positioned at the item "Lansing". If we write

```
cout << (*itr);
```

the output will be

```
Lansing
```

Note Because a reference is returned, we can use this operator to alter the value of an item in the list. For example,

```
*itr = "Detroit";
```

will change the value of the item itr is positioned at to "Detroit".

6.
```
// Postcondition: true has been returned if this iterator is positioned at the
//                  same place in this list x is positioned at. Otherwise, false has
//                  been returned.
bool operator== (const iterator& x);
```

Note There is also **operator**!=.

Here is a small program that illustrates several of the list methods and list-iterator operators:

```
#include <list>
#include <iostream>
#include <string>

using namespace std;

int main( )
{
      list<string> words;
      list<string>::iterator itr;

      words.push_back ("yes");
      words.push_back ("no");
      words.push_front ("maybe");
      words.push_front ("wow");
```

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

```
    cout << "size = " << words.size( ) << endl;

    cout << endl << "the list after 4 insertions:" << endl;
    for (itr = words.begin( ); itr != words.end( ); itr++)
        cout << (*itr) << endl;

    words.pop_front( );
    words.pop_back( );

    cout << endl << "the list after 2 deletions:" << endl;
    for (itr = words.begin( ); itr != words.end( ); itr++)
        cout << (*itr) << endl;
    cin.get( );

    return 0;

} // main
```

The output from this program is

```
size = 4

the list after 4 insertions:
wow
maybe
yes
no

the list after 2 deletions:
maybe
yes
```

Before we start looking at possible implementations of the list class, we should compare lists and vectors (or deques) from the user's point of view, that is, as data structures in the Standard Template Library. Section 6.1.3 explores the differences between lists and vectors as data structures.

6.1.3 Differences between List Methods and Vector or Deque Methods

The most significant difference between the list methods and the vector or deque methods is the absence of the index operator, operator[ ], from the list class. The omission is intended as a warning that lists lack the random-access property. We saw in Section 6.1 that we could simulate the effect of the index operator by looping from the front or back of a list, but the time would be linear in the number of items between the front (or back) item and the item at the given index.

Choose a vector (or deque) if the application entails accessing or modifying items at widely varying positions in a sequential container.

> So if the application entails a lot of accessing and/or modifying of items at widely varying positions in a sequential container, it will be completed much faster if a vector or deque is used instead of a list.

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

Choose a list if the application entails iterating through a sequential container and making insertions or deletions during the iterations.

On the other hand, to insert or erase the item that an iterator is positioned at takes only constant time with a list, versus linear time with a vector or deque.

> If a large part of an application consists of iterating through a sequential container and making insertions or erasures during the iterations, the application will be completed much faster if a list is used instead of either a vector or a deque.

Another difference between the list class and the vector or deque class is the extent of iterator invalidation as a result of insertions and deletions. Generally speaking, insertions and deletions in a list invalidate only the obvious iterators. For example, suppose that lis is a list object and itr1 and itr2 are iterators. If itr1 is positioned at item1 and itr2 is positioned at some later item, the message

```
lis.erase (itr1);
```

will invalidate itr1. That is, as you would expect, itr1 should no longer be depended on to point to item1. But itr2 would still be pointing to the same item it was pointing to before the erase message was sent.

Now suppose we had the same scenario for a vector vec and we sent the message

```
vec.erase (itr1);
```

Then itr2 will no longer be positioned at the item it was positioned at before the erase message was sent. The reason is that deletion in a vector causes a relocation of items beyond the point of erasure. A similar problem arises when deleting from a deque. And itr1 would also be invalidated because the erase method calls the destructor for the item itr1 was positioned at.

The status of iterators after inserting is similar. For a list, no items are moved, so iterators are still positioned where they were positioned before the insertion. For vectors, insertions can necessitate moving items, so iterators may be invalidated. Specifically, if the new size is greater than the old capacity, an expansion will occur, and all iterators and references are invalidated. Otherwise, only the iterators and references at or beyond the point of insertion are invalidated. The situation with deques is similar to vectors, except that invalidation applies from the insertion point to the front or back of the deque, whichever is closer to the insertion point.

6.1.4 Fields and Implementation of the list Class

In this section, we present an overview of one implementation of the Standard Template Library's list class, namely, the Hewlett-Packard version. Because of the C++ passion for efficiency, all widely used implementations—including the Hewlett-Packard implementation—are somewhat complex. In Section 6.1.6 we will consider some simpler—and less efficient—designs.

As with all the other template classes in the Hewlett-Packard implementation, the class declaration and method definitions for the list class are in the same file. The essential fields are length and node, defined in the following:

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

```
template<class T>
class list {
    protected:
        unsigned length;

        struct list_node
        {
            list_node* next;
            list_node* prev;
            T data; // holds one item
        }; // list_node

        list_node* node;
```

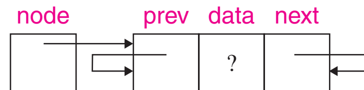
Figure 6.1 shows that the list nodes are strung together like beads in a necklace.

It is somewhat weird that a pointer to a node would have the identifier node, rather than node_ptr, for example, but this is common in virtually all implementations (probably because other implementations are based on Hewlett-Packard's).

The list_node pointed to by node is called the ***header node.*** In the header node, the data field is unused and, initially, the prev and next fields[1] point back to the header node itself. That is, the default constructor includes the following code:

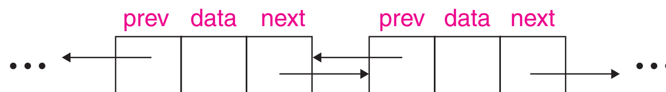
```
(*node).next = node;
(*node).prev = node;
```

So after the default constructor has been called, we have



In a nonempty list, the header node's next field points to the first item in the list and the prev field points to the last item in the list. So the list is stored as a circular, doubly linked list. For example, Figure 6.2 has a list container with two string items.

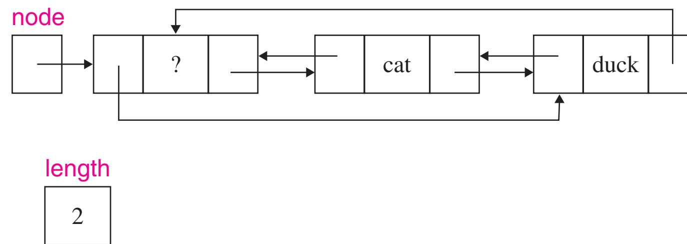
Figure 6.1 | In a list, each item is housed in a node that also includes pointers to the previous and next nodes.



[1]In the Hewlett-Packard implementation, the type of prev and next is given as **void*** because list_node* is correct only for the default allocator. But since we are assuming the default allocator, we substitute list_node*.

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

Figure 6.2 | A list with two items: "cat" and "duck". Each item is stored in a **struct** that also has prev and next fields.



The use of links to connect list nodes was the idea behind the Linked class in Chapter 2, but here each node has a prev pointer as well as a next pointer, and there is a header node.

Before we look at the definitions of some list methods, we need to say a little about the embedded iterator class. That class has two protected members: a field and a constructor:

```
protected:
        list_node* node;
        iterator (list_node* x) : node (x) { }
```

This constructor heading has a ***constructor-initializer section:*** a colon followed by any number of field initializations separated by commas. Each ***field initialization*** consists of the field identifier and, in parentheses, the initial value. The effect here is that the node field is initialized to x. In fact, the same effect could be accomplished without an initialization section by assigning x to node within the constructor definition.[2] Note that this is the node field in the iterator class, not the node field in the list class.

The reason the constructor is **protected** is that ordinary users will know nothing of list_node, so will have no cause to invoke this constructor. There is a default constructor, which is **public**. The definitions of the **public** methods in the iterator class are not unexpected. For example,

---

[2]A constructor-initializer section would be necessary if a constructor for an object field had to be called prior to the execution of the new class's constructor. For example, we might have

```
class D {
public:
        D (int i): v (i) {cout<< v << endl;}
protected:
        very_long_int v;
```

A constructor-initializer section is also needed to intitialize any non-**static** constants that have class scope.

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

```
public:
        iterator( ) { }

        T& operator*( ) const { return(*node).data;}

        iterator& operator++ ( )
        {
                node = (*node).next; return *this;
        }

        iterator operator++ (int)
        {
                iterator tmp = *this; ++*this; return tmp;
        }
```

Lab 15 has more details on the iterator class.

Now we can get to the list methods. We will define eight methods: begin, end, insert, push_front, push_back, erase (with one parameter), pop_front, and pop_back. As you can see from Figure 6.2, the header node is pointed to by the next field of the last list_node. That is, the header node is *one past* the last list_node in the list. So it makes sense that the end method in the list class returns an iterator positioned at the header node. Here is the definition:

```
iterator end ( ) { return node; }
```

Something strange is going on here. The value returned is node, that is, a *pointer* to the header node. But the return type of the end method is iterator! An iterator has a pointer field (also called node), but an iterator is an object, not a pointer. In C++, if an expression of an inappropriate type is encountered, the compiler will, if possible, perform an ***automatic type conversion*** to the appropriate type. In this case, the type list_node* needs to be converted to type iterator. And an automatic type conversion can be performed thanks to the protected constructor we saw a little while ago in the iterator class:

```
iterator (list_node* x) : node(x) { }
```

So what is really returned by the end method is not a copy of the list class's node field but rather an iterator constructed from the list class's node field.

As you can also see from Figure 6.2, the begin method should return an iterator positioned at the same list_node that the header node's next field is pointing to, namely, the list_node that has the first item in the list. The definition of the begin method, also employing automatic type conversion, is

```
iterator begin( ) { return(*node).next; }
```

Now let's tackle the insert method:

```
iterator insert (iterator position, const T& x);
```

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

This method stores an item x in list_node, adjusts some next and prev pointer fields to make that list_node be "in front of" the list_node pointed to by the iterator position, and returns an iterator positioned at the newly inserted node. For example, in Figure 6.3, pets is the list container from Figure 6.2 and the iterator position is positioned at the list_node containing "duck".

Figure 6.4 shows the effect of the message

```
pets.insert (itr, "dog");
```

on the list pets from Figure 6.3.

The key observation for you to make from Figure 6.4 is this:

> When an insertion is made in a list container, no items are relocated.

From Figure 6.4, we can infer the following steps to implement the insert method:

```
iterator insert (iterator position, const T& x);
```

1. allocate space for a list_node, pointed to by tmp.
2. Store item x in tmp's data field.
3. Assign to tmp's next field (technically, the next field of the link_node pointed to by tmp) the value of position.node.
4. Assign to tmp's prev field the value of the prev field of the link_node pointed to by position.node.

Figure 6.3 | The list from Figure 6.2, with the iterator position positioned at the list_node containing "duck".

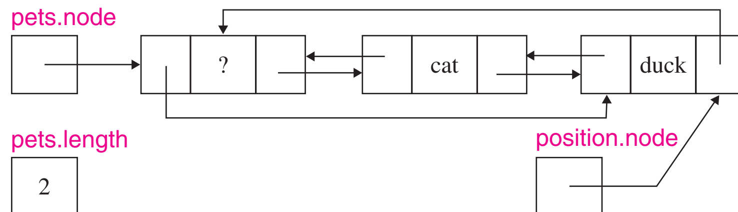
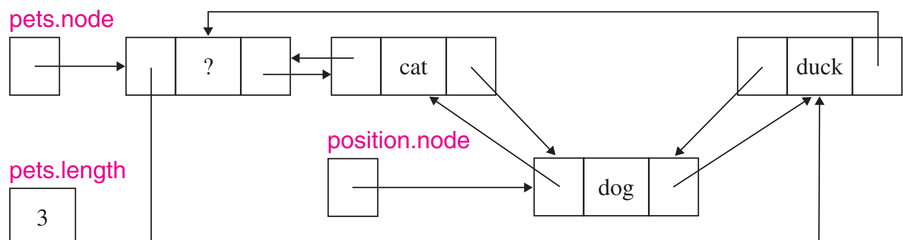


Figure 6.4 | The list from Figure 6.3 after "dog" is inserted.



Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

5. Assign the value of tmp to the next field of the link_node pointed to by the prev field of the list_node pointed to by position.node.
6. Assign the value of tmp to the prev field of the link_node pointed to by position.node. This assignment must be made *after* assignments 4 and 5; otherwise, the node that preceded position's node would be inaccessible.
7. Increment length.
8. Return tmp.

At this point, you could fill in most of the definition of the insert method. For example, step 4 would be written as

```
(*tmp).prev = (*position.node).prev;
```

But your definition would probably implement step 1 with

```
list_node *tmp = new list_node;
```

That would be effective but inefficient. For one thing, every list_node has the same size, but the general-purpose heap manager could not take advantage of this uniformity. Also, depending on the computer system, each call to the **new** operator may generate an interrupt, and these repeated interrupts could drastically slow down a project's execution.

The approach taken in the Hewlett-Packard implementation is to have the list class develop its own memory-management routines: get_node for allocating list nodes, and put_node for deallocation. Here is the resulting definition of insert:

```
iterator insert (iterator position, const T& x)
{
        list_node* tmp = get_node( );
        construct(value_allocator.address((*tmp).data), x);
        (*tmp).next = position.node;
        (*tmp).prev = (*position.node).prev;
        (* ((*position.node).prev)).next = tmp;
        (*position.node).prev = tmp;
        ++length;
        return tmp;
}
```

The details of the get_node method are covered in Lab 15, and the basics of list storage are explored in Section 6.1.5. To get a full appreciation of a list class implementation, you should peruse the actual code in your compiler's implementation of the list class. It will probably be quite similar to the implementation shown here.

The definitions of the push_front and push_back methods are one-liners:

```
void push_front(const T& x) { insert(begin( ), x); }
void push_back(const T& x) { insert(end( ), x); }
```

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

Because of the header node, every list node has a previous list node and a next list node, and this simplifies insertions and removals.

The brevity of these two definitions illustrates the beauty of having a header node: *the* insert *method also handles front insertions and back insertions!* The insert method always inserts the new item (in a list node) between two list nodes. For push_front method, the item is inserted between the header node and what had been the front node. For the push_back method, the item is inserted between what had been the back node and the header node.

We still have three more methods to define: erase, pop_front, and pop_back. But once we have defined the erase method, the definitions of pop_front and pop_back will follow easily (thanks again to having a header node). With the insert method, we attached a new item (in a link_node) to a list. The erase method *detaches* an item from the list. Figure 6.5 shows a three-item list, with an iterator positioned at one of the items.

To erase the item "dog" from pets, there are, essentially, two steps:

1. Change the next field in cat's list_node to point to duck's list_node.
2. Change the prev field in duck's list_node to point to cat's list _node.

The effect of carrying out these two steps is shown in Figure 6.6.

There is still some housekeeping to take care of: the destructor for the string object "dog" must be called, the link_node that had "dog" has to be deallocated, and the length field has to be decremented. Here is the definition:

```
void erase(iterator position)
```

Figure 6.5 | A tidied-up version of the three-item list from Figure 6.4.

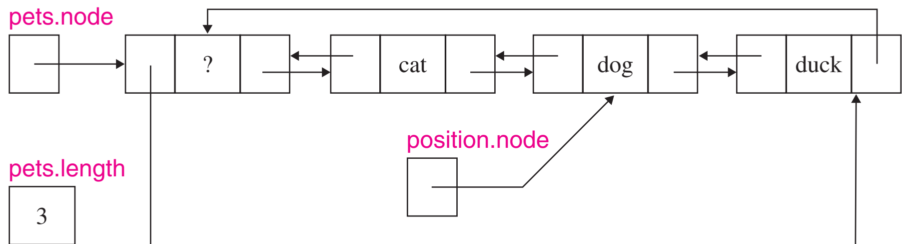
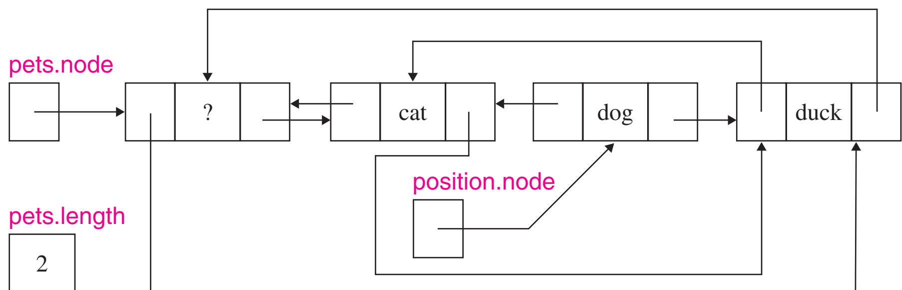


Figure 6.6 | The list from Figure 6.5 after detaching "dog".



Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

```
{
        (*((*position.node).prev)).next = (*position.node).next;
        (*((*position.node).next)).prev = (*position.node).prev;
        destroy(value_allocator.address((*position.node).data));
        put_node(position.node);
        --length;
}
```

Section 6.1.5 will explain how the put_node method deallocates the space for a deleted node.

The pop_front method erases the item that begin( ) is positioned at, and the pop_back method erases the item just before where end( ) is positioned:

```
void pop_front( ) { erase(begin( )); }

void pop_back( )
{
        iterator tmp = end( );
        erase(--tmp);
}
```

We finish up this implementation of the list class by studying the allocation and deallocation of list nodes.

6.1.5 Storage of list Nodes

In Section 6.1.4, we saw that the insert method called the get_node method to return a pointer to the node that will hold the item to be inserted. The definition of the get_node method is discussed in Lab 15, but for that discussion to make sense, we need to study how list nodes are stored.

When the first insertion is made in a list, a chunk of memory—typically, 1K bytes—is allocated. This chunk, called a ***buffer***, is used for subsequent insertions until the buffer is filled—we'll ignore for now what happens when the buffer is filled and how deletions fit into this representation. To determine when the buffer is filled, the list class has a next_avail field, pointing to the node to be used in the next insertion, and a last field, pointing one past the last node in the buffer:

```
list_node* next_avail;
list_node* last;
```

Figure 6.7 illustrates a list object, pets, with four items: "cat", "dog", "duck", and "lion".

The next_avail field is used whenever a new node is allocated, whether the allocation is for insert, push_front, or push_back. In Figure 6.7, for example, a call to push_back would adjust the prev and next fields of next_avail's list node so that list node's prev field would point to the "lion" list node and that list node's next field would point to the header node. And then next_avail itself would be incremented. For example, Figure 6.8 shows the effect of

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

Figure 6.7 | A list of four pets, stored in buffer.

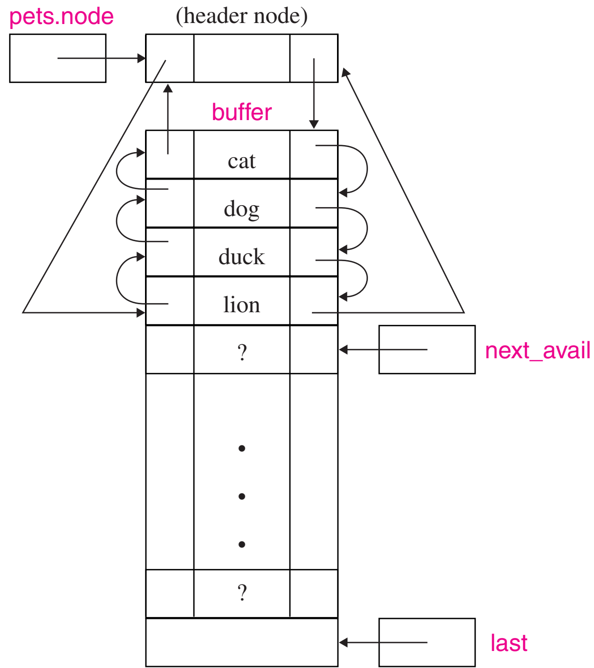
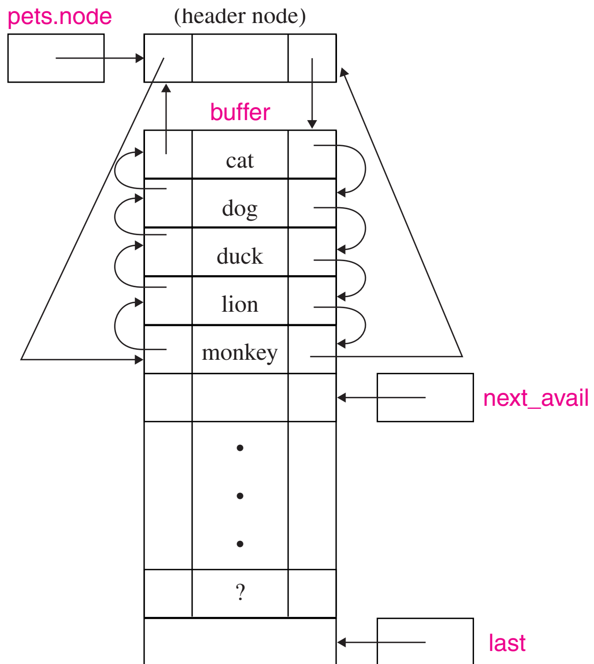


Figure 6.8 | The list from Figure 6.7 after pets.push_back ("monkey") is called.



Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

```
pets.push_back ("monkey");
```

on the configuration shown in Figure 6.7.

There are two more details we need to consider: What happens when the buffer gets used up, and what happens to deleted nodes? When the buffer is full, that is, when next_avail = last, a new buffer of the same size is allocated. To keep track of all of a list's buffers—so they can later be deallocated when the list is destroyed—the list class has a buffer_list field. This field contains a pointer to a singly linked list whose nodes are of type list_node_buffer. Each list_node_buffer has two fields: a pointer to a buffer, and a pointer to the next list_node_buffer:

```
struct list_node_buffer
{
        list_node_buffer* next_buffer;
        list_node* buffer;
};

list_node_buffer* buffer_list;
```

For example, Figure 6.9 shows a list with three buffers allocated.

Lastly, but not leastly, we need to say something about deletions. It would be wasteful if deleted nodes were just left to rot. Instead, there is a list of nodes that had once been in the list but were subsequently erased. These recycled nodes are organized in a singly linked list. The free_list field in the list class holds a pointer to the most recently deleted node:

```
list_node* free_list;
```

This node's prev field is ignored, and its next field points to the node that had been next most recently erased. That (next most recently erased) node's next field points to the third most recently erased node, and so on as the free list winds its way through all the allocated buffers. A deleted node is appended to the free list in the put_node method:

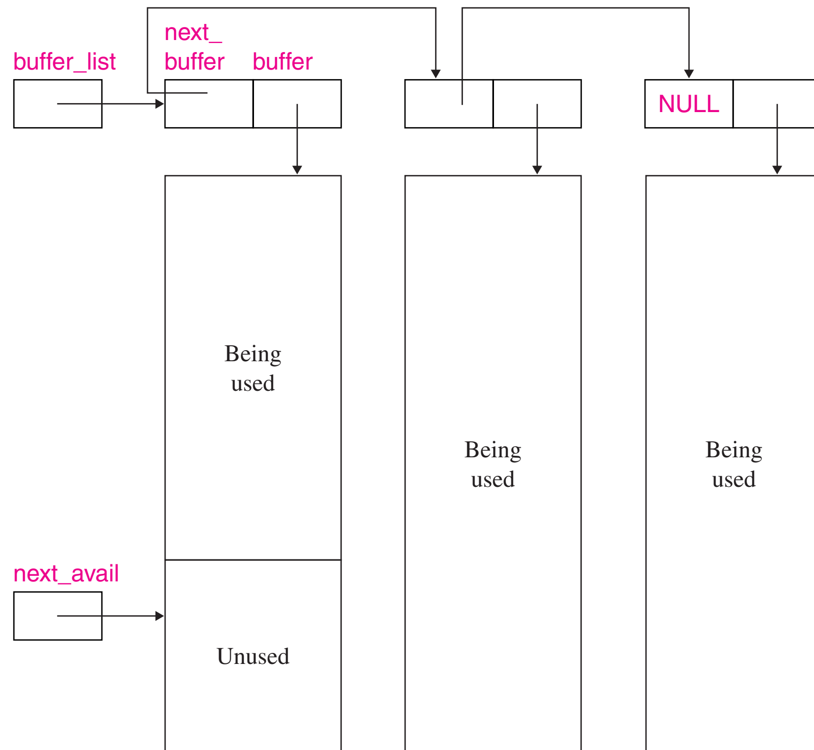
```
void put_node(link_type p)
{
        p->next = free_list;
        free_list = p;
}
```

For example, if "duck" is deleted from the five-pet list in Figure 6.8, then Figure 6.10 shows how the list is affected.

Whenever an insertion—including push_front or push_back—occurs, the free list is checked. If the free list is not empty, its front node is used for the insertion and deleted from the free list. If the free list is empty, next_avail's node is used unless next_avail = last. In that case, a new buffer is allocated and chained to the beginning of the buffer list, and the first node in that new buffer is allocated.

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

Figure 6.9 | Buffer allocation for a list. Three buffers have been allocated, and part of the most recently allocated buffer has space in which no list nodes have yet been stored. The array variable buffer points to the first list node in the array.



The space for the various buffers and lists is not deallocated until the list is destroyed. So if your application creates a large list and then deletes almost all the items, *all* the list's space will still be allocated.

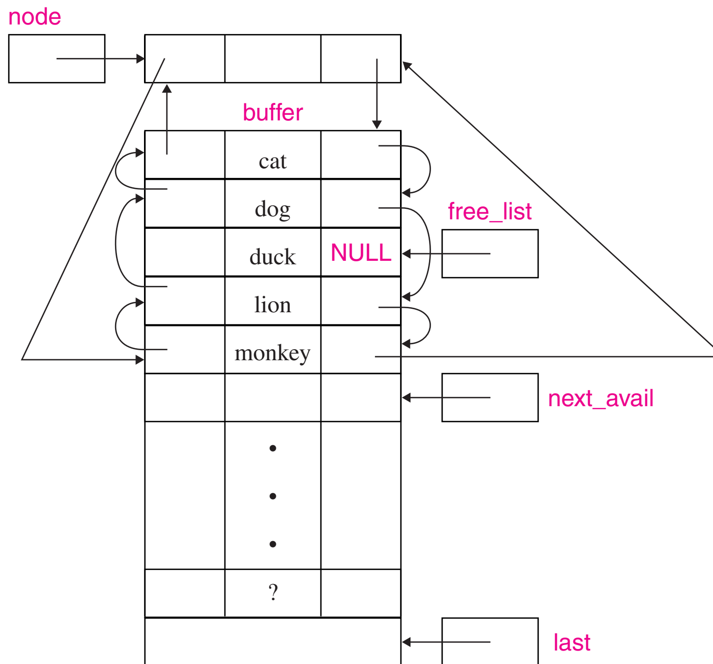
Lab 15 has still more details of the Hewlett-Packard implementation, and Lab 16 has an experiment comparing vectors, deques, and lists. And now that you have seen several kinds of iterators—random-access and bidirectional—you are ready to study the organization of iterators in the Standard Template Library.

LAB **Lab 15: More implementation details for the list class.** *(All Labs Are Optional.)* LAB

LAB **Lab 16: Timing the sequence containers.** *(All Labs Are Optional.)* LAB

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

Figure 6.10 | The list from Figure 6.8 after "duck" is erased. The node with "duck" is now at the head of, in fact the only node in, the free list.



Lab 17: Iterators, part 2.

(All Labs Are Optional.)

As we trumpeted in Section 6.1.4, one of the major, yet subtle features of the Hewlett-Packard implementation is the header node. Because of this node, we did not need a special case for inserting at the front or back, or for removing from the front or back. Each given node always has a node in front of the given node and a node in back of the given node. Specifically, the header node is always located before the first node in a list, and the header node is always located behind the last node in a list.

This advantage of header nodes will become apparent in Section 6.1.6, when we look at alternative implementations of the list class. The implementations are simpler but less efficient than the Hewlett-Packard implementation of the list class. You will get the opportunity, in Project 6.2, to complete one of the implementations outlined in Section 6.1.6.

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

6.1.6 Alternative Implementations of the list Class

We now explore a few other implementations of the list class. For the sake of simplicity, we will rely on the heap manager's implementation of the new and delete operators for the allocation and deallocation of list nodes. How about the singly linked Linked class from Chapter 2? Here are the fields in the Linked class:

```
protected:
    struct Node
    {
        T item;
        Node* next;
    }; // struct Node

    Node* head;
    Node* tail; // this field was added in Lab 8
    long length;
```

Could we expand the Linked class to satisfy all the method interfaces for the list class? The problem comes with the postconditions—specifically, the time estimates—of some of the list methods.

For example, the postcondition for the pop_back method in the list class does not explicitly include a time estimate. By convention, that means that for any implementation of that method, worstTime(n) must be constant. How would our pop_back method work for the Linked container in Figure 6.11?

The pop_back method would have to make NULL the next field of the node *before* the tail node. And for that task, a loop is needed, so worstTime(n) would be linear in n. That would violate the constant-time requirement of the pop_back method interface in the list class. So we must abandon our attempt at a singly linked implementation of the list class.

What if we modify the Linked class to make it doubly linked? Figure 6.12 shows the corresponding three-item list from Figure 6.11.

Here are the field definitions:

```
protected:
    struct Node
    {
        T item;
        Node* next;
        Node* prev;
    }; // struct Node

    Node* head;
    Node* tail;
    long length;
```

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

Figure 6.11 | A Linked container with three items. How would a pop_back method be defined for this class?

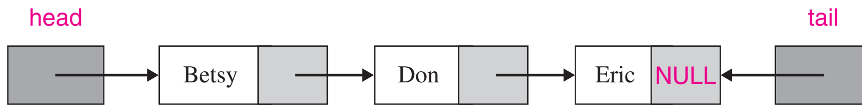


Figure 6.12 | A doubly linked container with head and tail fields.

The default constructor would simply set head and tail to NULL and length to 0. But we now have NULL references to watch out for. For example, the push_front method would have a special case for an empty container:

```
void push_front (const T& x)
{
        Node* temp_head = new Node;
        (*temp_head).item = x;
        (*temp_head).next = head;
        (*temp_head).prev = NULL;
        length++;
        if (head == NULL)
        {
                head = temp_head;
                tail = head;
        } // if
        else
        {
                (*head).prev = temp_head;
                head = temp_head;
        } // else
} // method push_front
```

A similar test would be needed for push_back. For pop_front and pop_back, a special case would be needed for removal of the only item in the container. Also, the insert method would first test

```
if (head == NULL || position.node == head)
```

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

The bottom line for this implementation is that it is encumbered by the special cases for inserting or deleting at the front or back of a list. The Hewlett-Packard implementation avoided special cases by having a dummy node—the header node—that was both in front of the first node in a list *and* in back of the last node.

You will get to flesh out the details of the doubly linked head and tail implementation if you undertake Project 6.2. This version uses the new and delete operators for space management, so it will be somewhat less efficient than the Hewlett-Packard implementation, which has its own memory-management methods (get_node and put_node).

In Section 6.2, we leave behind the nitty-gritty of implementation details and look at an application of the list class: a simple text editor.

APPLICATION 6.2 | LIST APPLICATION: A LINE EDITOR

As an illustration of the list class, let's develop a line editor. A ***line editor*** is a program that manipulates text, line by line. We assume that each line is at most 75 characters long. The first line of the text is thought of as line 0, and one of the lines is designated the ***current line.*** Each editing command begins with a dollar sign, and *only* editing commands begin with a dollar sign. There are eight editing commands. Here are four of the commands; the remaining four are specified in Project 6.1.

1. $Insert

Each subsequent line, up to the next editing command, will be inserted in the text *after* the current line. The last line inserted becomes the current line. If the text is empty when $Insert is called, then the insertions become the only lines in the text. For example, suppose the text is empty and we have the following:

```
$Insert
Water, water every where,
And all the boards did shrink;
Water, water every where,
Nor any drop to drink.
```

Then after the insertions, the text would be as follows, with ">" to indicate the current line:

```
 Water, water every where,
 And all the boards did shrink;
 Water, water every where,
>Nor any drop to drink.
```

For another example, suppose the text is

```
 Now is the
>time for
 citizens to come to
 the
 aid of their country.
```

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

Then the sequence

```
$Insert
all
good
```

will cause the text to become

```
  Now is the
  time for
  all
>good
  citizens to come to
  the
  aid of their country.
```

2. **$Delete k m**

Each line in the text between lines k and m, inclusive, will be deleted. If the current line had been in this range, the new current line will be line $k - 1$. Otherwise, the current line is the same line that it was before this command. For example, suppose the text is

```
  Now is the
  time for
  all
>good
  citizens to come to
  the
  aid of their country.
```

The command

```
$Delete 3 5
```

will cause the text to become

```
  Now is the
  time for
>all
  aid of their country.
```

If k has the value 0 and the current line is one of the lines deleted, the interpretation is that, after the deletion, the current line is before any of the lines in the text. So if the $Insert command follows, the insertions are made *in front of* the first line of the text. For example, suppose the text is

```
  a
  s
>p
  a
  r
  k
```

and the commands are

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

$Delete 0 2
$Insert
q
u

then the text becomes

  q
>u
  a
  r
  k

After the deletion but before the insertion, the current line was before any line in the text. The insertion inserted two lines at the beginning of the text, so the current line is now "u". The following error messages should be printed when appropriate:

***** Error: The first line number > the second.**
***** Error: The first line number < 0.**
***** Error: The second line number > last line number.**
***** Error: The command is not followed by two integers.**

3. **$Line m**

The line whose line number is *m* becomes the current line. For example, if the text is

  Mairzy doats
  an dozy doats
>an liddle lamsy divy.

then the command

$Line 0

will make line 0 the current line:

>Mairzy doats
  and dozy doats
  and liddle lamsy divy.

The command

$Line −1

followed by the $Insert command, is used to insert lines at the beginning of the text. An error message should be printed if *m* is either less than −1 or greater than the last line number in the text. See command 2.

4. **$Done**

This terminates the execution of the text editor. For convenience, we will print out the final text. An error message should be printed for any illegal command, such as "$End", "$insert", or "?Insert".

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

System Test 1 The input is in boldface.

Please enter a line:
$Insert
Please enter a line:
This is line zero.
Please enter a line:
This is line one.
Please enter a line:
This is line two.
Please enter a line:
$Line 1
Please enter a line:
$Insert
Please enter a line:
This is line 1.5.
Please enter a line:
This is line 1.6.
Please enter a line:
This is line 1.7.
Please enter a line:
This is line 1.8.
Please enter a line:
$Delete 1 3
Please enter a line:
$Done
Here is the final text:
 This is line zero.
 This is line 1.7.
>This is line 1.8.
 This is line two.

Please press the Enter key to close this output window.

System Test 2 The input is in boldface.

Please enter a line:
Insert
*** Error: Not one of the given commands
Please enter a line:
$Insert
Please enter a line:
a

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

Please enter a line:
b
Please enter a line:
$line
*** Error: Not one of the given commands
Please enter a line:
$Line 2
*** Error: The line number must be less than the text size.
Please enter a line:
$Done
Here is the final text:
  a
>b

Please press the Enter key to close this output window.

6.2.1 Design of the Editor Class

In order to decide what methods the Editor class should contain, we ask what does an editor have to do? From the editor commands given, some of the responsibilities are apparent:

- To parse the line to see if it is a legal command
- To check the commands for errors
- To manage the text

This is enough to start with. The parse method will interpret a line read in. This method will have the line as its only parameter. What should parse return? For some commands—$Insert, $Delete, and $Line—an error message should be returned if the command cannot be carried out. For some commands—$Done—the entire text should be returned. In Project 6.1, there is a command, $Print, for which either an error message or some text should be returned. How can we distinguish between an error message and text? According to the specifications of the problem, a text line cannot start with a “$”, so by prepending that character to any error message, we can tell which is which. The parse method will return a string, representing an error message if the first character is “$”, and text for the $Done command (for the $Insert, $Delete, and $Line commands, and for inserting a line, a blank line will be returned if there is no error).

The command_check method will check each command for errors, and for each of the four commands there will be a separate method. Here are the method interfaces:

```
// Postcondition: this Editor is empty.
void Editor( );
```

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

```
// Postcondition: if line is a legal command, that command has been carried
//                out and the result of carrying out that command has been
//                returned. If line is to be inserted, that has been attempted
//                and the result has been returned. Otherwise, an illegal-
//                command error message has been returned.
string parse (const string& line);

// Postcondition: line has been checked for errors. If no error was found, the
//                command has been processed and the result returned.
//                Otherwise, an error message has been returned.
string command_check (const string& line);

// Postcondition: if line is not too long, it has been inserted into this Editor and
//                a blank line has been returned. Otherwise, an error
//                message has been returned.
string insert _command (const string& line);

// Postcondition: lines k through m of the text have been deleted, if possible,
//                and a blank line has been returned. Otherwise, an
//                error message has been returned.
string delete_command (int k, int m);

// Postcondition: the line at index m has become the current line in the text, if
//                possible, and a blank line has been returned. Otherwise, an
//                error message has been returned.
string line_command (int m);

// Postcondition: the execution of the editor has been completed and the text
//                has been returned.
string done_command( );
```

Before we can start to define these six methods, we have to decide what fields we will have. One of the fields will hold the text, so we'll call it text. The text will be a sequence, and we will often need to make insertions and/or deletions in the interior of the text, so text should be an object in the list class (surprise!).

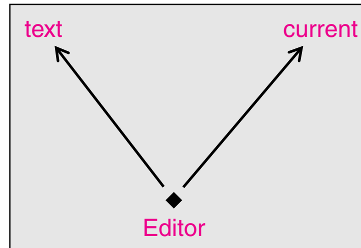
To keep track of the current line, we could either have an integer field, current LineNumber, or an iterator field, current. Some of the commands—$Delete and $Line—work with line numbers, but list insertions and deletions require iterators, so it is not easy to say which would be better. How about both? We'll try that solution, even though both fields will have to be updated for each insertion and deletion. The **bool** field inserting will determine whether the line entered is to be inserted into the

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

Editor or is to be treated as a command. All the fields are protected to allow their use by subclasses:

```
protected:
        list<string> text;
        list<string>::iterator current;
        int currentLineNumber;
        bool inserting;
```

Here is the dependency diagram, with two examples of composition:



Now that we know the method interfaces and fields, we can implement the Editor class by defining those methods.

6.2.2 Implementation of the Editor Class

The default constructor explicitly initializes the fields, except text, which is initialized by *its* default constructor.

```
Editor::Editor( )
{
        current = text.begin( );
        currentLineNumber = −1;
        inserting = false;
} // default constructor
```

The parse method determines if the line represents a command or is to be inserted. If neither, the command-start character (‘$’) is prepended to the error message.

```
string Editor::parse (const string& line)
{
        if (line.substr (0, 1) != COMMAND_START)
                if (inserting)
                                return insert_command (line);
                else
                                return COMMAND_START +
                                        MISSING_COMMAND_ERROR +
                                        COMMAND_START;
        return command_check (line);
} // parse
```

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

The command_check message separates the command from the command arguments and invokes the appropriate command. For the $Delete and $Line commands, we must extract line numbers from the command line. For example, suppose the line contains

$Line 173

The position of the blank is determined with the string method find. The substring "173" starts at the first blank position + 1, and goes until the end of the string. The string method substr returns that substring, which is converted to an array of characters with the string method c_str. Finally, the atoi function converts that array of digit characters into an integer.

```
string Editor::command_check (const string& line)
{
    string command;
    int blank_pos1 = line.find (BLANK),
        blank_pos2;

    if (blank_pos1 >= 0) // line has at least one argument
        command = line.substr (0, blank_pos1);
    else
        command = line;
    if (command == INSERT_COMMAND)
    {
        inserting = true;
        return BLANK;
    } // $Insert
    else
    {
        int k,
            m;

        inserting = false;

        if (command == DELETE_COMMAND)
        {
            // find k and m
            if (blank_pos1 >= 0)
            {
                blank_pos2 = line.find (BLANK, blank_pos1 + 1);
                if (blank_pos2 >= 0)
                {
                    k = atoi (line.substr (blank_pos1 + 1,
                        blank_pos2 - blank_pos1 - 1).c_str( ));
                    m = atoi (line.substr (blank_pos2 + 1).c_str( ));
                    return delete_command (k, m);
```

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

```
                    } // two line numbers given
                    return COMMAND_START +
                            MISSING_NUMBER_ERROR;
                } // at least one line number given
                return COMMAND_START +
                        TWO_NUMBERS_ERROR;
                } // $Delete
            else if (command == LINE_COMMAND)
            {
                // find m
                if (blank_pos1 >= 0)
                {
                    m = atoi (line.substr (blank_pos1 + 1).c_str( ));
                    return line_command (m);
                } // line number given
                return COMMAND_START + MISSING_NUMBER_ERROR;
            } // $Line
            else if (command == DONE_COMMAND)
                return done_command( );
            return COMMAND_START + ILLEGAL_COMMAND_ERROR;
        } // not an insert command
    } // command_check
```

Finally, we get to the real work, managing the list object text. The constant-time insert_command method checks for a line that is too long and, if it is not, inserts line in text *after* the line where current is positioned:

```
string Editor::insert_command (const string& line)
{
    if (line.length( ) > MAX_LINE_LENGTH)
        return COMMAND_START + LINE_TOO_LONG_ERROR;
    current = text.insert (++current, line);
    currentLineNumber++;
    return BLANK;
} // insert
```

The delete_command first checks for errors, such as k < 0. Otherwise, we start an iterator first at the beginning of the text and then increment first k times. We then erase the next m − k items in the text. After that loop, we need to update current and currentLineNumber. Notice that deciding whether to update current would be somewhat difficult if we did not have a currentLineNumber field. Here is the definition of the delete_command method:

```
string Editor::delete_command (int k, int m)
{
```

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

```
        if (k < 0)
            return COMMAND_START + FIRST_TOO_SMALL_ERROR;
        if (m >= (int)text.size( ))
            return COMMAND_START + SECOND_TOO_LARGE_ERROR;
        if (k > m)
            return COMMAND_START +
                 FIRST_GREATER_THAN_SECOND_ERROR;

        list<string>::iterator first = text.begin( );

        for (int i = 0; i < k; i++)
            first++;

        for (int i = k; i <= m; i++)
            text.erase (first++);

        if (currentLineNumber >= k && currentLineNumber <= m)
        {
            currentLineNumber = k - 1;
            current = --first;
        } // if
        else if (currentLineNumber > m)
            currentLineNumber -= m + 1 - k ; // current is unchanged
        return BLANK;
        } // delete_command
```

How long does the delete_command method take? Let n represent the number of lines in text. In the worst case, with k = 0 and m = $n - 1$, each line will be erased, and each call to the erase method takes constant time, so worstTime(n) is linear in n. On average, the value of m will be about $n/2$, so the number of iterations—and averageTime(n)—is still linear in n.

The line_command increments or decrements current, depending on where currentLineNumber is in relation to n:

```
    string Editor::line_command (int m)
    {
        if (m < -1)
            return COMMAND_START + FIRST_TOO_SMALL_ERROR;
        if (m >= (int)text.size( ))
            return COMMAND_START + FIRST_TOO_LARGE_ERROR;
        if (currentLineNumber < m)
        {
            for (int i = currentLineNumber; i < m; i++)
                current++;
            currentLineNumber = m;
        } // if
```

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

```
        else
        {
            for (int i = currentLineNumber; i > m; i--)
                current--;
            currentLineNumber = m;
        } // else
        return BLANK;
    } // line_command
```

Let n represent the number of lines of text. In the worst case, when currentLineNumber $= -1$ and m $= n - 1$, current will iterate through each line of text, so worstTime(n) is linear in n. On average, the distance between currentLineNumber and m will be about $n/2$, and so the average number of iterations—and averageTime(n)—is still linear in n.

For the done_command method, we return the text, including the current line marker:

```
    string Editor::done_command( )
    {
        const string FINAL_MESSAGE = "Here is the final text: \n"; string
                text_string = FINAL_MESSAGE;

        if (currentLineNumber == -1)
            text_string += " >\n";
        for (list<string>::iterator itr = text.begin( ); itr != text.end( ); itr++)
            if (itr == current)
                text_string += " >" + *itr + "\n";
            else
                text_string += " " + *itr + "\n";
        return text_string;
    } // done_command
```

This method iterates through each line of text, so worstTime(n) and averageTime(n) are linear in n.

The main function handles all the input-output for the line-editor application.

The main function defines editor, an Editor object, and then calls editor.parse (line) for each line of input. The result returned is stripped of the leading “$”, if there is one, and printed. Inputting a line presents a problem. The extraction operator, operator>>, could be used to read in a command, but we won’t know, at that point, whether the line also includes line numbers, as part of the $Delete and $Line commands, for example. That determination is made in the Editor class’s methods, which have no input statements. So we will need to read in an entire line, whitespace and all, in the main function. C++ provides the getline function for such occasions. The function interface is

```
    // Postcondition: the characters in inStream, stripped of leading whitespace,
    //                from the current character up to '\n', have been stored in line.
    istream& getline (istream& inStream, string& line);
```

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

The Editor methods can then subdivide the line.

Here is the definition, with a do loop to ensure that the result from the $Done command is printed before the loop terminates:

```
int main( )
{
    const string PROMPT = "Please enter a line: ";
    const string CLOSE_WINDOW_PROMPT =
        "Please press the Enter key to close this output window.";

    Editor editor;

    string result;

    string line;

    do
    {
        cout << PROMPT<< endl;
        getline (cin, line);
        result = editor.parse (line);
        if (result.substr (0, 1) != COMMAND_START)
            cout << result << endl << endl;
        else
            cout << result.substr (1) << endl << endl;
    } // do
    while (line != DONE_COMMAND);

    cout << CLOSE_WINDOW_PROMPT;
    cin.get( );
    return 0;
} // main
```

There is one iteration of the do loop for each item in the text (there are also other iterations of the loop). So worstTime(n) is at least linear in n. To get a better estimate, we would need to know the sequence of commands.

Without changing the Editor class, the program can be modified to accept file input and send the output to a file. Here is the revised main function.

```
int main( )
{
    const string IN_PROMPT = "Please enter the path for the input file: ";

    const string OUT_PROMPT =
        "Please enter the path for the output file: ";

    const string ECHO = "The line was: ";

    const string CLOSE_WINDOW_PROMPT =
        "Please press the Enter key to close this output window.";
```

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

```
        Editor editor;

        string result;

        string inFileName,
               outFileName,
               line;

        fstream inFile,
                outFile;

        cout << IN_PROMPT;
        cin >> inFileName;
        cout << OUT_PROMPT;
        cin >> outFileName;
        inFile.open (inFileName.c_str( ), ios::in);
        outFile.open (outFileName.c_str( ), ios::out);

        do
        {
              getline (inFile, line);
              outFile << ECHO << line << endl;
              result = editor.parse (line);
              if (result.substr (0, 1) != COMMAND_START)
                     outFile << result << endl << endl;
              else
                     outFile << result.substr (1) << endl << endl;
        } // do
        while (line != DONE_COMMAND);
        outFile.close( );

        cout << CLOSE_WINDOW_PROMPT;
        cin.get( );
        return 0;
} // main
```

All the relevant files are available from the Source Code link on the book's website.

SUMMARY

The focus of this chapter is the list class. Lists are sequential containers that lack the random-access ability of vectors and deques. But they take only constant time for insertions and deletions in the interior—versus linear time for vectors and deques. This constant-time aspect holds because the insert and erase methods require as a parameter an iterator that is positioned where the insertion or deletion is to occur.

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

The application, a simple line editor, took advantage of the list class's ability to quickly make multiple insertions and deletions anywhere in the list.

EXERCISES

6.1 a. Suppose we define the following:

```
list <char> letters;
list<char>::iterator itr;
```

Show the sequence of letters in the list after each of the following messages is sent:

```
itr = letters.begin( );
letters.insert (itr, 'f');
letters.insert (itr, 'e');
itr++;
letters.insert (itr, 'r');
itr++;
itr++;
letters.insert (itr, 't'); // Hint: we now have e,r,f,t
letters.insert (itr, 'e');
letters.erase (letters.begin( ));
itr--;
itr--;
letters.insert (itr, 'p');
itr++;
letters.insert (itr, 'e');
itr = letters.end( );
itr--;
letters.insert (itr, 'c');
```

b. Write the code to print out the final contents of letters in part a.

c. Redo part a, with letters being an array of characters instead of a list of characters.

d. Redo part a, with letters being a string instead of a list of characters.

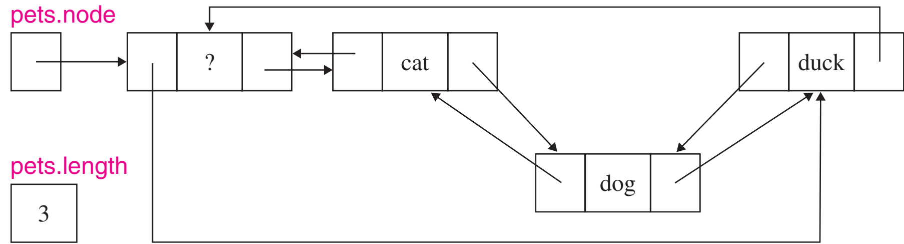
e. Redo part a, with letters being a vector of characters instead of a string.

f. Redo part a, with letters being a deque of characters instead of a string.

6.2 Compare lists with vectors and deques with respect to the Big-O time for access, insertions, and deletions.

6.3 Show the effect of the following messages on the list in Figure 6.4, repeated here:

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.




```
list<string>::iterator position = pets.begin( );
pets.push_front('bunny"):
pets.erase (position);
pets.push_back ('frog");
```

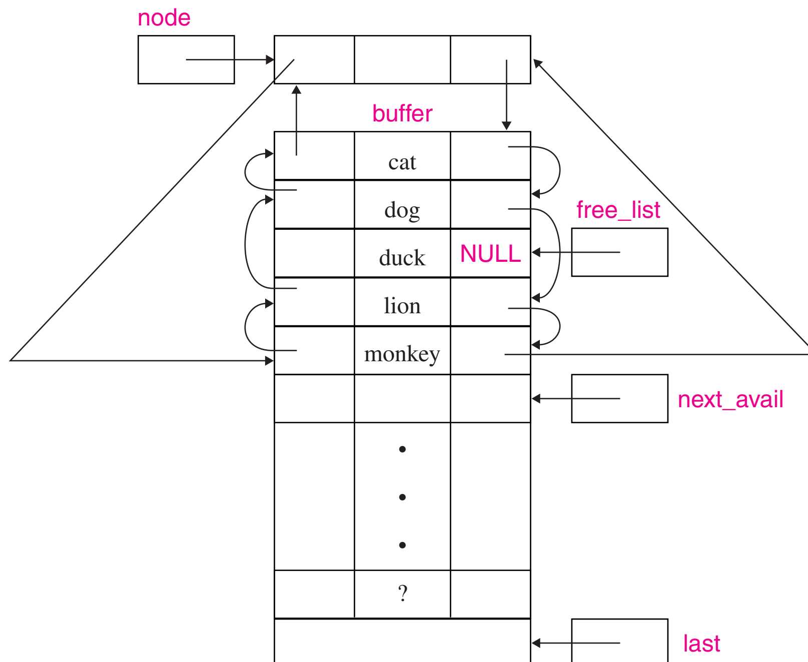
6.4 Which do you think would provide a faster implementation of the very_long_int class: a vector, a deque, or a list? Why?

6.5 Suppose that the list class utilized the heap manager—with the new and delete operators—for memory allocation and deallocation. Define the insert method and the one-parameter erase method.

6.6 In the Editor class's delete_command method, the variables k and m are of type **int**. Why would it be a mistake if the type of k were **unsigned**?

6.7 Suppose myList is a list object whose items are of type **double**. Write the code to print out the items in reverse order.

6.8 Figure 6.10 is repeated here:



Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

If "lion" is deleted from this list, what will that node's next field point to:

a. The node whose item is "cat"?

b. The node whose item is "dog"?

c. The node whose item is "duck"?

d. The header node?

Hint The deleted node becomes the front node in the free list.

6.9 If the list class were designed with head and tail fields, we could avoid NULL values for the prev and next fields in each list node by making the list circular. That is—in a nonempty list—the head node's prev field would point to the tail node, and the tail node's next field would point to the head node. What effect would this design have on the implementation of the begin, end, and push_front methods?

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

PROGRAMMING PROJECT 6.1

Extending the Editor Class

Extend the Editor class to include methods with the following interfaces:

5. $Change
%X%Y%

In the current line, each occurrence of the string given by X will be replaced by the string given by Y. For example, suppose the current line is

bear ruin'd choirs, wear late the sweet birds sang

Then the command

$Change
%ear%are%

will cause the current line to become

bare ruin'd choirs, ware late the sweet birds sang

If we then issue the command

$Change
%wa%whe%

we would get

bare ruin'd choirs, where late the sweet birds sang

Notes

1. If either X or Y contains a percent sign, the end-user should select another delimiter. For example,

   $Change
   #0.16#16%#

2. The string given by Y may be the empty string. For example, if current line is

   aid of their country.

   then the command

   $Change
   %of %%

   will change the current line to

   aid their country.

3. If the delimiter occurs fewer than three times, the error message to be printed is

   ***** Error: Delimiter must occur three times.**

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

6. $Last

The line number of the last line in the text will be printed. For example, if the text is

  I heard a bird sing
> in the dark of December.
  A magical thing
  and a joy to remember.

then

$Last

will cause **3** to be printed. The text and the designation of the current line are unchanged.

7. $Print k m

Each line number and line in the text, from lines *k* through *m*, inclusive, will be printed. For example, if the text is

  Winston Churchill once said that
> democracy is the worst
  form of government
  except for all the others.

then command

$Print 0 2

will cause the following to be printed:

0 Winston Churchill once said that
1 democracy is the worst
2 form of government

The text and the designation of the current line are unchanged. As in command 2, an error message should be printed if (1) *k* is greater than *m*, or if (2) *k* is less than 0 or *m* is greater than the last line number in the text.

System Test 1 Sample input is in boldface.

Please enter a line:
$Insert

Please enter a line:
You can fool

Please enter a line:
some of the people

Please enter a line:
some of the times,

(continued on next page)

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

(continued from previous page)

Please enter a line:
but you cannot foul

Please enter a line:
all of the peeple

Please enter a line:
all of the time.

Please enter a line:
$Line 2

Please enter a line:
$Print 2 1
*** Error: The first line number > the second.

Please enter a line:
$Print 2 2

2 some of the times,

Please enter a line:
$Change %s%%

Please enter a line:
$Print 2 2

2 ome of the time,

Please enter a line:
$Change %o%so
*** Error: Delimiter must occur three times.

Please enter a line:
$Change %o%so%

Please enter a line:
$Print 2 2

2 some sof the time,

Please enter a line:
Change
*** Error: Command must begin with $.

Please enter a line:
$Change %sof%of%

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

Please enter a line:
$Print 2 2

2 some of the time,

Please enter a line:
$Line -1

Please enter a line:
$Insert

Please enter a line:
Lincoln once said that

Please enter a line:
you can fool

Please enter a line:
some of the people

Please enter a line:
all the time and

Please enter a line:
all of the time and

Please enter a line:
$Last

10

Please enter a line:
$Print 0 10

0 Lincoln once said that
1 you can fool
2 some of the people
3 all the time and
4 all of the time and
5 You can fool
6 some of the people
7 some of the time,
8 but you cannot foul
9 all of the peeple
10 all of the time.

Please enter a line:
$Line 5

(continued on next page)

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

(continued from previous page)

Please enter a line:
$Change %Y%y%

Please enter a line:
$Print 5 5

5 you can fool

Please enter a line:
$Line 6

Please enter a line:
$Change %some%all%

Please enter a line:
$Print 6 6

6 all of the people

Please enter a line:
$Line 8

Please enter a line:
$Change %ul%ol%

Please enter a line:
$Print 8 8

8 but you cannot fool

Please enter a line:
$Line 9

Please enter a line:
$Change %ee%eo%

Please enter a line:
$Print 9 9

9 all of the people

Please enter a line:
$Delete 3 3

Please enter a line:
$Print 0 10
*** Error: The second line number is greater than the number of lines in the text.

Please enter a line:
$Last

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

9

Please enter a line:
$Print 0 9

0 Lincoln once said that
1 you can fool
2 some of the people
3 all of the time and
4 you can fool
5 all of the people
6 some of the time,
7 but you cannot fool
8 all of the people
9 all of the time.

Please enter a line:
$Done

Here is the final text:
  Lincoln once said that
  you can fool
  some of the people
  all of the time and
  you can fool
  all of the people
  some of the time,
  but you cannot fool
>all of the people
  all of the time.

Please press the Enter key to close this output window.

System Test 1 Sample input is in boldface.

Please enter a line:
$Insert

Please enter a line:
Life is full of

Please enter a line:
successes and lessons.

Please enter a line:
$Delete 1 1

(continued on next page)

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

(continued from previous page)

Please enter a line:
$Insert

Please enter a line:
wondrous oppurtunities disguised as

Please enter a line:
hopeless situations.

Please enter a line:
$Last

2

Please enter a line:
$Print 0 2

0 Life is full of
1 wondrous oppurtunities disguised as
2 hopeless situations.

Please enter a line:
$Line 1

Please enter a line:
$Change %ur%or%

Please enter a line:
$Print 0 2

0 Life is full of
1 wondrous opportunities disguised as
2 hopeless situations.

Please enter a line:
$Done

```
 Here is the final text:
 Life is full of
>wondrous opportunities disguised as
 hopeless situations.
```

Please press the Enter key to close this output window.

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

PROGRAMMING PROJECT 6.2

An Alternate Design and Implementation of the list Class

Implement the doubly linked, head-and-tail design of the list class described in Section 6.1.6. The only list methods you need to implement for this project are the first thirteen methods given in Section 6.1.1. Set up a driver to validate your implementation. You will also need to implement at least a few iterator operators: *, !=, and ++ (either preincrement or postincrement).

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.

Copyright © McGraw-Hill Companies Inc. Permission required for reproduction or display.