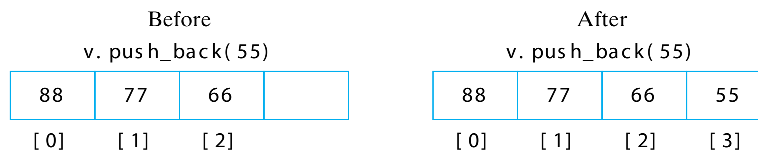
14.4 The STL list<T> Class Template

In our description of the C++ Standard Template Library in Section 10.6 of the text, we saw that it provides a variety of other storage containers besides vector<T> and that one of these containers is named list<T>. Now that we have seen anonymous variables and how C++ pointers provide indirect access to them, we are ready to examine the list<T> class template and its implementation.

A LIMITATION OF vector<T>

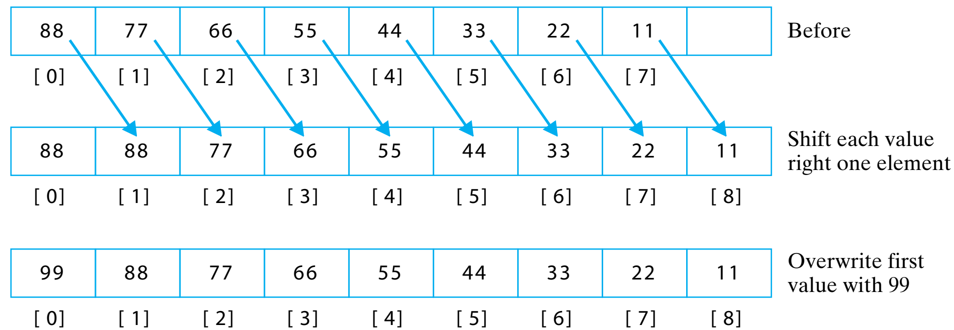
Although vector<T>s are easy to use to store sequences of values, they do have limitations. One limitation is that values can be efficiently added to the sequence only at its *back*. If there are empty elements at the end of its run-time allocated array,[1] the push_back() message permits values to be appended to the *back* of the sequence, without the existing values having to be copied:



Consequently, when a value is appended to a vector using push_back(), any values already in the vector will stay in the same positions.

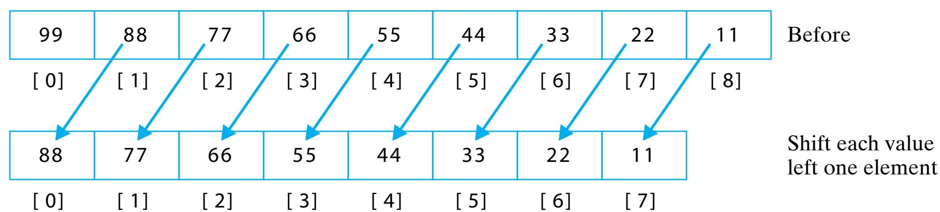
Why is it that vector<T> provides no corresponding push_front() (or pop_front()) functions to manipulate the *front* of the sequence? Because inserting and deleting values at the front of a vector requires extensive copying of values, which takes time. To insert a value at the front, all of the values in the vector must be shifted one position to make room for the new value:

1. To make sure that this is usually the case, each time push_back() is used to append a value to a vector whose run-time allocated array is full, a new array that is twice as large is allocated and the elements of the old array are copied into it. If the vector has no run-time allocated array (i.e., its capacity was zero), then an array is allocated whose capacity is implementation-dependent.



In fact, if a problem requires that values be inserted anywhere except at the back of a sequence, a `vector<T>` is not the best container for storing that sequence, because of the copying required in shifting values to make room for the new value.

The same problem occurs when any element other than the one at the end of the sequence must be removed. In the vector, all of the elements that follow it must be shifted one position to the left to close the gap. The following diagram illustrates this when the first element is removed:



For problems where many such within-the-sequence insertions and deletions are required, STL provides the `list<T>` container. It allows values to be inserted or removed anywhere in a sequence without any of the copying that plagues `vector<T>`.

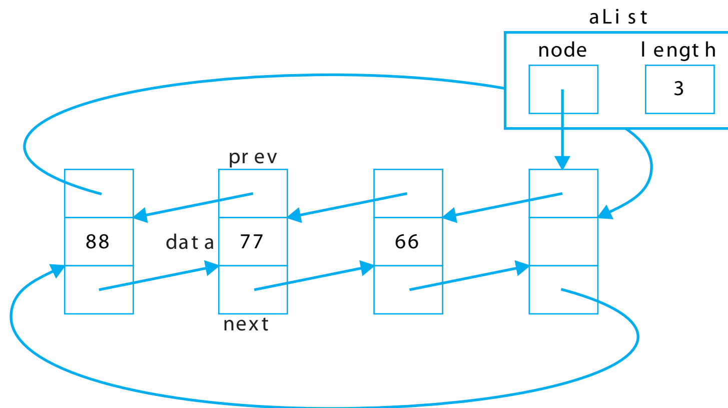
Organization of `list<T>` Objects. To see how `list<T>` stores a sequence of values, suppose that `aList` is defined by

```
list<int> aList;
```

and consider the following sequence of insert operations:

```
aList.push_back(77);
aList.push_back(66);
aList.push_front(88);
```

A simplified picture of the resulting object `aList` is



The values 88, 77, and 66 are stored in a variation of the linked lists studied in the Section 14.3 of the text called a **circular doubly-linked list with a head node.** It is doubly-linked because each node has two pointers, `prev` to its predecessor and `next` to its successor. It is circular because the `next` pointer in the last node is not null, but rather points to the "empty" (rightmost) node, which is the head node; and similarly, the `prev` pointer in the first node points to the head node instead of being null. Note that the instance variable `node` (which we called `first` in `LinkedList` in Section 14.3 of the text) points to this head node rather than to the first node that stores a data value.

The `list<T>` class template declares the type `list_node` as a protected struct,[2] as follows:

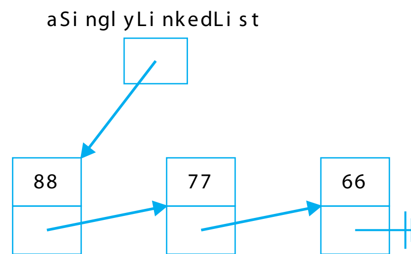
```
template<class T>
class list
{
   // ... previous part of list class  ...

protected:
   struct list_node
   {
      list_node* prev;     // address of the node containing
                           //    the previous value
```

2. A **struct** is exactly the same as a class, except that all of its members are by default public, whereas those of a class are by default private. By declaring a `list_node` as a *struct* within class `list`, the `list` operations can directly access the `list_node` instance variables. By declaring `list_node` *protected* within class `list`, casual users of class `list` are prevented from accessing it or its instance variables, while classes derived from `list` are permitted to do so.

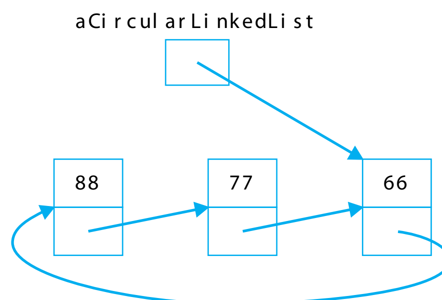
```
      T data;              // the value being stored in
                           //   this node
      list_node* next;     // address of the node containing
                           //   the next value
   };

   // ... remainder of list class ...
};
```

Linked lists that use nodes containing two pointers are called **doubly-linked lists.**

Although the designers of STL chose this organization for their `list<T>` class template, other organizations are possible. One of these is a **singly-linked list,** like those described in Section 14.3 of the text. They consist of a pointer to the first in a sequence of nodes, each containing the value being stored and just one link, a pointer to the next node in the sequence. The final node in the sequence is marked by the null address in its link member:



Another arrangement is a **circular linked list,** which is a singly-linked list, but contains a pointer to the last node. In this arrangement, the final node's link consists of a pointer back to the first node, providing easy access to both the last and first values in the sequence:



As we shall see next, regardless of its organization, the linked structure of a list allows insertion

and deletion operations to be performed that do not require the extensive copying that characterizes its vector<T> counterpart.

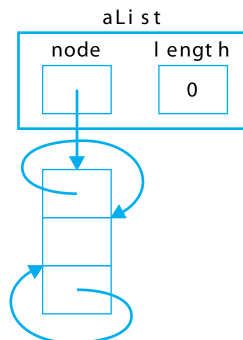
SOME list<T> OPERATIONS

In the remainder of this section, we examine a collection of list<T> operations that illustrate the flexibility provided by pointers. A complete table of the list<T> operations is given in Appendix D. of the text.

The list<T> Default-Value Constructor. Perhaps the most basic list<T> operation is the default-value constructor. When a programmer writes

```
list<int> aList;
```

the default-value constructor builds an empty linked list aList, for which a (simplified) picture is



As shown in the diagram, the default class constructor allocates an empty node, called a **head node,** and stores the address of this node in its node instance variable. In the STL list<T> class template, this head node plays a central role:

- Its next member always points to the node containing the *first* value in the sequence (or to the head node, if the list is empty);
- Its prev member always points to the node containing the *last* value in the sequence (or to the head node, if the list is empty); and
- Its instance variable is unused.

The main advantages of this organization is that there is always at least one node in the list (i.e., the head node) and every node has a predecessor and a successor. These properties simplify several of the list<T> operations.

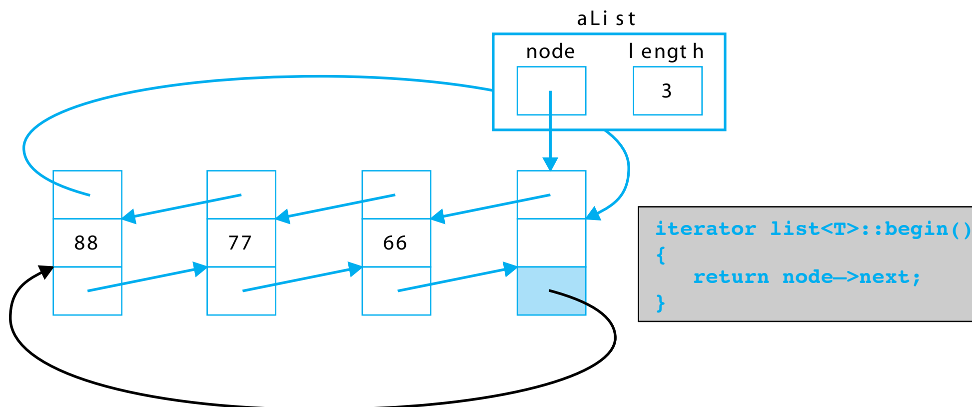
The `size()` and `empty()` Methods. Two of the simplest `list<T>` operations are `size()` and `empty()`. The `size()` method is a simple accessor for the `length` instance variable; it returns the number of values currently stored in the list. It might be defined as follows:

```
template<class T>
inline int list<T>::size() const
{
   return length;
}
```

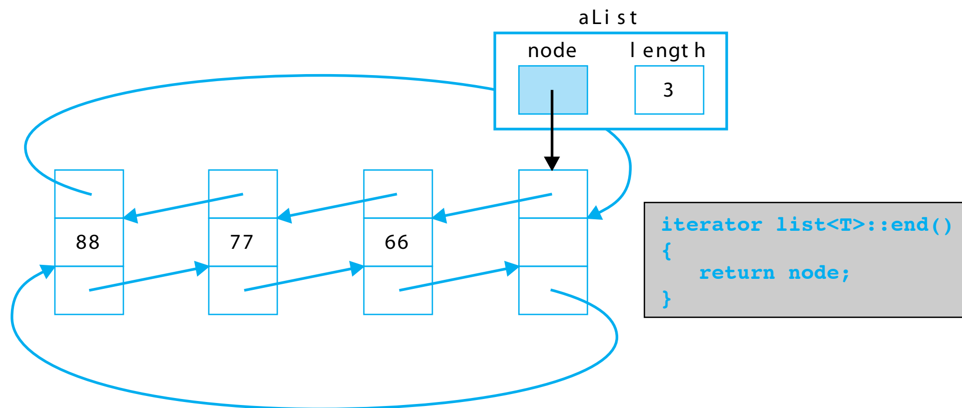
The `empty()` method is nearly as simple, returning `true` if there are no values in the list and `false` otherwise. Its definition might be

```
template<class T>
inline bool list<T>::empty() const
{
   return length == 0;
}
```

The `begin()` and `end()` Iterator Methods. As with `vector<T>`, the `list<T>` class template provides two methods, `begin()` and `end()`, that return iterators to the front and past the end of the list, respectively. In the `list<T>` class template, these methods are implemented using the pointer instance variables of the head node. More precisely, the `begin()` method returns a pointer to the first node, by returning the address stored in the `next` member of the head node:



By contrast, the `end()` function returns a pointer pointing beyond the last node that contains a value by returning the address of the head node:



The `begin()` method thus returns an iterator to the first value in the list, and the `end()` method returns an iterator that points beyond the final value in the list.

Iterators and Pointers. From our discussion of iterators in preceding chapters and our discussion of pointers in this chapter, it should be evident that an iterator is an *abstraction* of a pointer, hiding some of its details and eliminating some of its hazards.

To illustrate, the `list<T>` class template declares a `list<T>::iterator` as an object containing its own `list_node` pointer named `node` as a instance variable. With much of the detail omitted, the class can be thought of as having a structure somewhat like the following:

```
template<class T>
class list
{
 // ... previous list members omitted ...

 public:
   class iterator      // ... some simplification here ...
   {
    protected:
      list_node* node; // ... and here ...

      // ... other iterator members omitted...
   };

   // ... other list members omitted ...
};
```

The `iterator` class overloads `operator*` so that it returns the value of the `instance variable` in the `list_node` pointed to by the iterator's `node` member. Here is a simplified defini-

tion:

```
template<class T>
inline T list<T>::iterator::operator*()
{
   return node->data;
}
```

The `iterator` class also overloads `operator++` to "increment" the iterator to the next node in the list:

```
template<class T>
inline iterator list<T>::iterator::operator++() // prefix version
{
   node = node->next;
   return *this;
}

template<class T>
inline iterator list<T>::iterator::operator++(int i) // postfix
{
   iterator tmp = node;
   node = node->next;
   return tmp;
}
```

and overloads `operator--` similarly to "decrement" the iterator to the previous node in the list.

The `front()` and `back()` Members. Like `vector<T>, list<T>` provides methods to access the first and last values in the sequence. These are implemented by dereferencing the iterators returned by the `begin()` and `end()` operations:

```
template<class T>
inline T& list<T>::front()
{
   return *begin();
}

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

Since the `list<T>::iterator` class overloads `operator*` to return the `instance variable` of the `list_node` whose address is stored in its `node` member, an expression like

```
*begin()
```

can be used to access the first value in the sequence, and an expression like

```
*(--end())
```

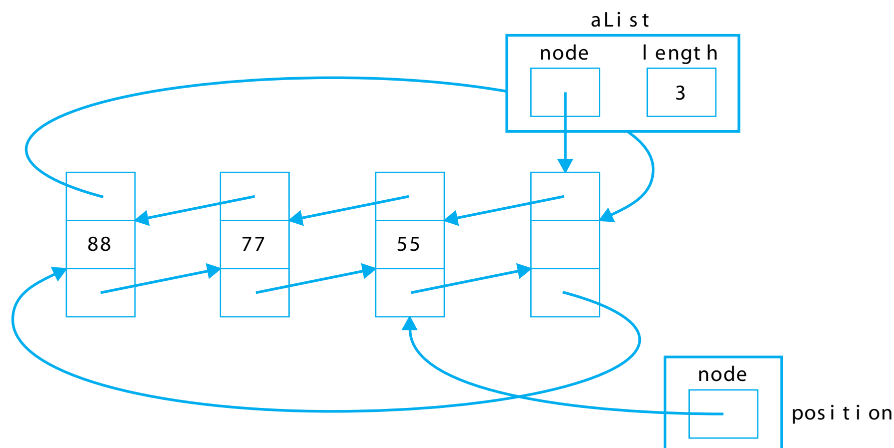
can be used to access the last value in the sequence. Note that `operator*` has higher precedence than `operator--`, so parentheses must be used in this last expression to ensure that the iterator returned by `end()` is decremented before it is dereferenced.

The `insert()`, `push_front()`, and `push_back()` Methods. To add a value to a sequence, the `list<T>` class template provides several operations, including:

- `aList.push_back(newValue);` which appends *newValue* to *aList*;
- `aList.push_front(newValue);` which prepends *newValue* to *aList*; and
- `aList.insert(anIterator, newValue);` which inserts *newValue* into *aList* ahead of the value pointed to by *anIterator*.

Of these three, `insert()` is the most general operation—the `push_back()` and `push_front()` operations are implemented using `insert()`— and we will therefore focus our discussion on it.

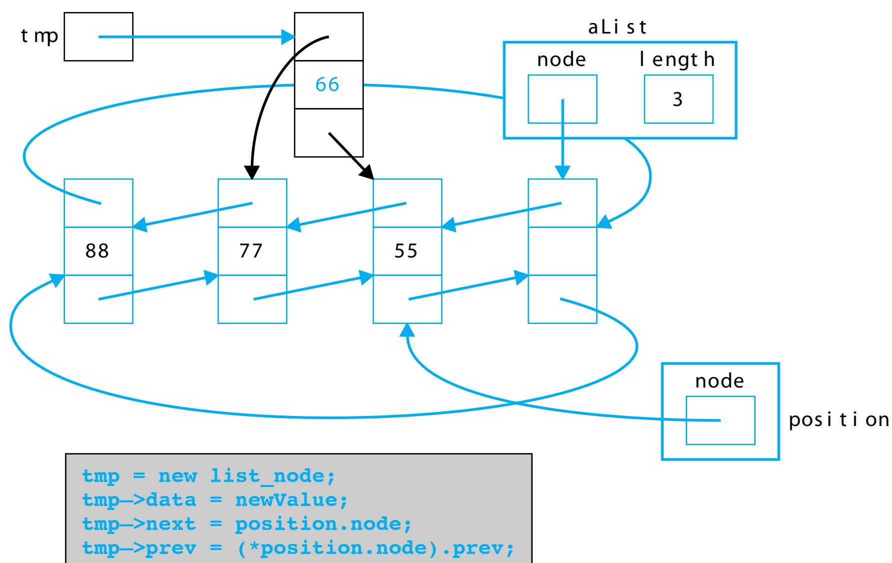
To illustrate its behavior, suppose that `aList` is the following `list<int>`, and `position` is a `list<int>::iterator` that has been positioned at the node containing 55 (perhaps by using the STL `find()` algorithm):



Now suppose that the following statement is executed:

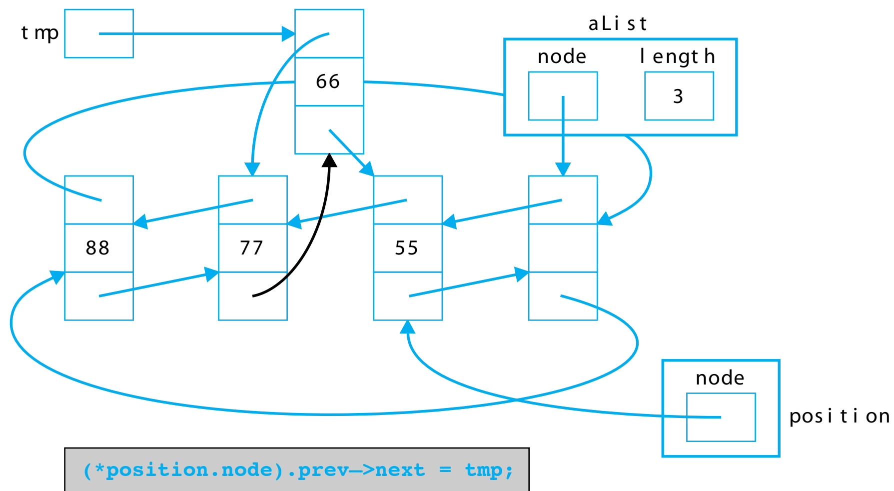
```
aList.insert(position, 66);
```

The `insert()` operation gets a new node,[3] assigns its `instance variable` the value `66`, assigns its `prev` member the address of the node containing 77, and assigns its `next` member the address of the node containing 55:

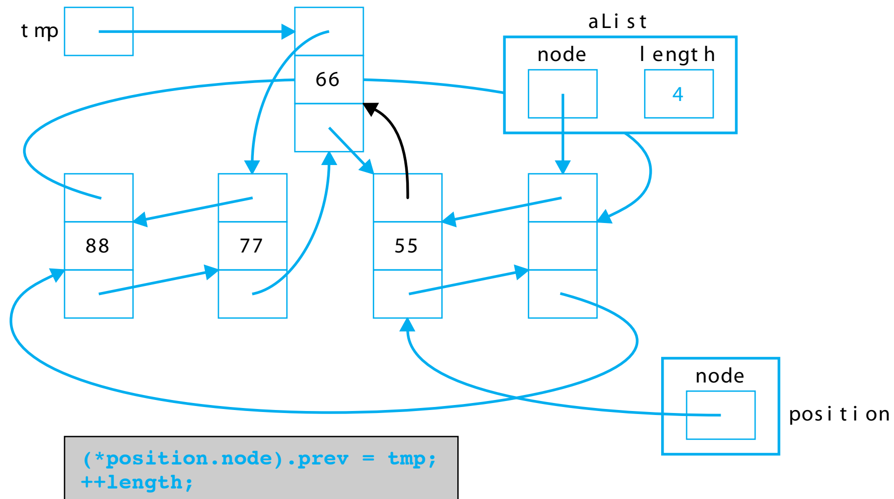


The `next` pointer in the node before `position` is then assigned the address of the new node:

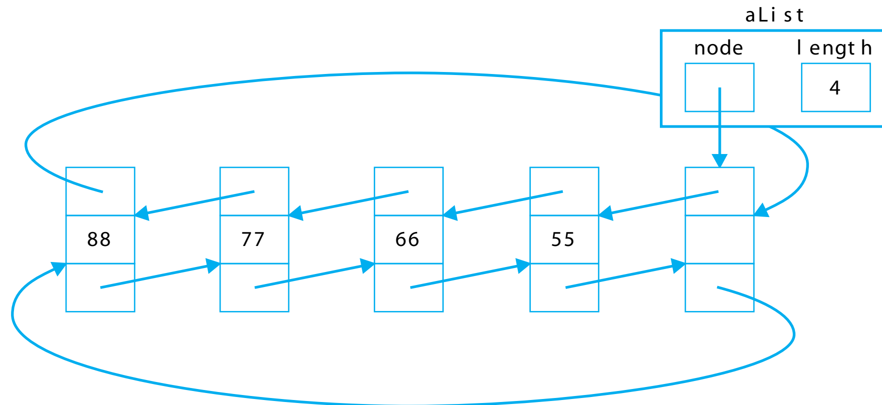
3. We show the new node being allocated using `new`; however, the STL list class actually manages its own collection of `list<T>` nodes. Only when this pool of nodes becomes empty does it use `new` to add more nodes to the pool. The `insert()` method issues a call to `get_node()`, an operation that gets the next available node from this pool of nodes, refilling it with more nodes when it is depleted.



Finally, the `prev` member of the node pointed to by `position` is updated to point to the new node, and the `length` member is incremented:



All we have done is change the values of four pointers, but this has inserted the value 66 into the sequence between the values 77 and 55. Although the nodes containing the sequence values could be anywhere in memory, we can picture the resulting list as follows:



The `push_front()` and `push_back()` operations behave in a similar manner. `push_front()` effectively uses `insert()` and `begin()` to insert its value at the beginning of the sequence,

```
template<class T>
inline void list<T>::push_front()
{
   insert(begin(), newValue);
}
```

while `push_back()` uses `insert()` and `end()` to insert its value at the end of the sequence:

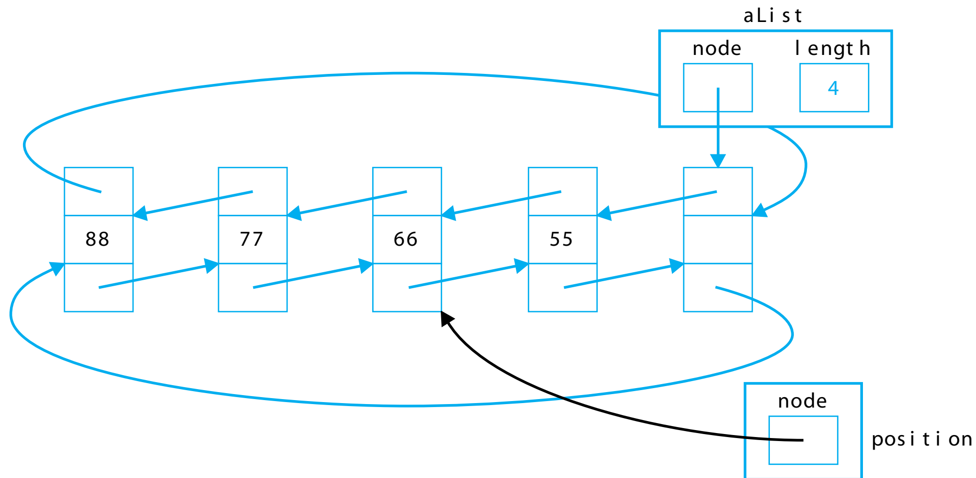
```
template<class T>

inline void list<T>::push_back()
{
   insert(end(), newValue);
}
```

The `pop_back()`, `pop_front()`, `erase()` and `remove()` Methods. To remove a value in a sequence without any copying, `list<T>` provides several different operations:

- *aList*`.pop_back();` removes the last value from *aList*
- *aList*`.pop_front();` removes the first value from *aList*
- *aList*`.erase(`*anIterator*`);` removes the value pointed to by *anIterator* from *aList*
- *aList*`.remove(`*aValue*`);` removes all occurrences of *aValue* from *aList*

The `pop_back()`, `pop_front()`, and `remove()` operations are implemented using the `erase()` function, so we will focus on this operation.

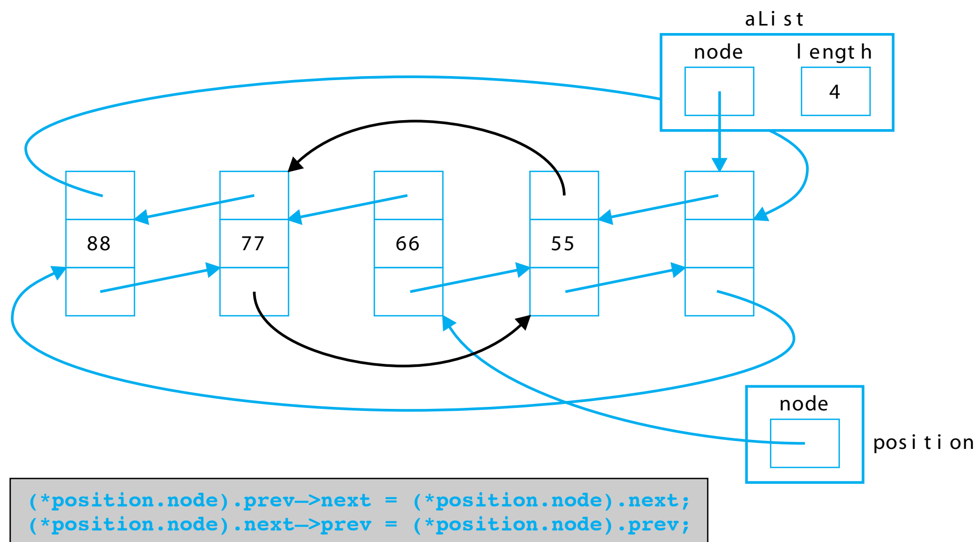
To illustrate its behavior, suppose that `aList` is the `list<int>` we just examined and that `position` is a `list<int>::iterator` pointing at 66, the value we wish to erase:



The call

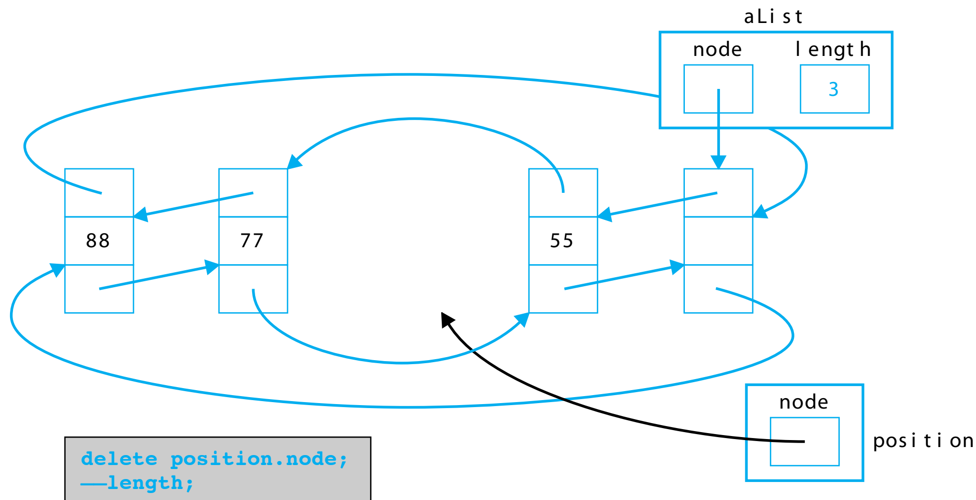
```
aList.erase(position);
```

begins by making the `next` member of the node containing 77 point to the node containing 55 and the `prev` member of the node containing 55 to point to the node containing 77:



These two statements cut the target node out of the sequence, so that all that remains to do is deallocate that node,[4] and decrement the list's `length` member:

4. We show the node being deallocated using `delete`; however `erase()` actually uses a call to a `put_node()` operation that stores the node in the list's pool of nodes (see footnote 10). This approach allows a subsequent `insert()` operation to *recycle* that node, thus avoiding the overhead of deallocating the node now, and reallocating it later. The nodes in the `list_node_buffer` are deallocated by `~list()`, the list destructor.



Note that the `erase()` operation removes the value pointed to by `position` simply by changing two pointers. No copying of values is required, thanks to the flexibility of the linked nodes.

As described earlier, the `pop_front()` and `pop_back()` operations are implemented using this `erase()` function: `pop_front()` erases the node at position `begin()`, and `pop_back()` erases the node at position `--end()`. The **`remove()`** operation that deletes a given value from the sequence can be implemented using `erase()` and a simple `while` loop, as follows:

```
template<class T>
void list<T>::remove(const T& value)
{
   iterator first = begin(), // begin at first node
            last = end(),    // stop at head node
            next = first;    // save current node address

   while (first != last)
   {
      ++next;                // save address of next node
      if (*first == value)   // if value in current node
         erase(first);       //   erase it
      first = next;          // reset first to next node
   }
}
```

These are just a few of the `list<T>` operations. A complete list can be found at the end of this section.

AN APPLICATON: INTERNET ADDRESSES

The TCP (Transmission Control Protocol) and IP (Internet Protocol) communication protocols specify the rules computers use in exchanging messages on the Internet. TCP/IP addresses are used to uniquely identify computers in the Internet; for example, `www.ksc.nasa.gov` is the address of a site at the NASA Kennedy Space Center. Such an address is made up of four fields that represent specific parts of the Internet,

host.subdomain.subdomain.rootdomain

which the computer will translate into a unique TCP/IP numeric address. This address is a 32-bit value, but it is usually represented in a dotted-decimal notation by separating the 32 bits into four 8-bit fields, expressing each field as a decimal integer, and separating the fields with a period; for example, `198.119.202.36` is the TCP/IP numeric address for the above site at the NASA Kennedy Space Center. (A *Part of the Picture* detailing the TCP/IP Communications Architecture written by William Stallings is available on the text's CD and website.)

Problem. A gateway is a device used to interconnect two different computer networks. Suppose that a gateway connects a university to the Internet and that the university's network administrator needs to monitor connections through this gateway. Each time a connection is made (for example, a student using the World Wide Web), the TCP/IP address of the student's computer is stored in a data file. The administrator wants to check periodically who has used the gateway and how many times they have used it.

Solution. The TCP/IP addresses will be read from the file and stored in a linked list of nodes that will store an address and the number of times that address appeared in the data file. As each address is read, we check if it is already in the list. If it is, we increment its count by 1; otherwise, we simply insert it at the end of the list. After all the addresses in the file have been read, the distinct addresses and their counts are displayed.

The following program uses this approach to solve the problem. The addresses are stored in a `list<AddressCounter>` object named `addrCntList`, where `AddressCounter` is a small class containing two instance variables (`address` and `count`), input and output methods, and a `tally()` method to increment the `count` instance variable. Also, `operator==()` is overloaded so that STL's `find()` algorithm can be used to search the list.

Figure 14.2 Internet Addresses.

```
/* internet.cpp reads TCP/IP addresses from a file and produces a
 *  list of distinct addresses and a count of how many times each
 *  appearedin the file.  The addresses and counts are stored in a
```

```
 *  linked list.

 * Input (keyboard): name of file containing addresses
 * Input (file):     addresses
 * Output:           a list of distinct addresses and their counts
 *****************************************************************/

#include <cassert>                  // assert
#include <string>                   // string
#include <iostream>                 // cin, cout, >>, <<
#include <iomanip>                  // setw()
#include <fstream>                  // ifstream, isopen()
#include <list>                     // list<T>
#include <algorithm>                // find
using namespace std;

//--------------- Begin class AddressItem -------------------------
class AddressCounter
{
 public:
   void read(istream & in) { in >> address; count = 0; }

   void print(ostream & out) const
   { out << setw(15) << left << address
         << "\t occurs " << count << " times\n"; }

   void tally() { count++; }

   friend bool operator==(const AddressCounter& addr1,
                          const AddressCounter& addr2);

 private:
   string address;
   int count;
};

inline bool operator==(const AddressCounter& addr1,
                       const AddressCounter& addr2)
{ return addr1.address == addr2.address; }

//----------------- End class AddressCounter --------------------

typedef list<AddressCounter> TCP_IP_List;


int main()
{
   string fileName;                         // file of TCP/IP addresses
   TCP_IP_List addrCntList;                 // list of addresses

   ifstream inStream;                       // open file of addresses
   cout << "Enter name of file containing TCP/IP addresses: ";
```

```
   cin >> fileName;
   inStream.open(fileName.data());
   assert(inStream.is_open());

   AddressCounter item;                    // one address & its count
   for (;;)                                // input loop:
   {
      item.read(inStream);                 //   read an address
      if (inStream.eof()) break;           //   if eof, quit

      TCP_IP_List::iterator it =           //   is item in list?
        find(addrCntList.begin(), addrCntList.end(), item);
      if (it != addrCntList.end())         //   if so:
        (*it).tally();                     //    ++ its count
      else                                 // otherwise
        addrCntList.push_back(item);       // add it to the list
   }                                       // end loop

   cout << "\nAddresses and Counts:\n\n";  // output the list
   for (TCP_IP_List::iterator it = addrCntList.begin();
                              it != addrCntList.end(); it++)
      (*it).print(cout);
}
```

Listing of file *ipAddresses.dat* used in sample run:

```
128.159.4.20
123.111.222.333
100.1.4.31
34.56.78.90
120.120.120.120
128.159.4.20
123.111.222.333
123.111.222.333
77.66.55.44
100.1.4.31
123.111.222.333
128.159.4.20
```

Sample run:

```
Enter name of file containing TCP/IP addresses: ipAddresses.dat

Addresses and Counts:

128.159.4.20      occurs 2 times
123.111.222.333   occurs 3 times
100.1.4.31        occurs 1 times
```

```
34.56.78.90       occurs 0 times
120.120.120.120   occurs 0 times
77.66.55.44       occurs 0 times
```

list<T> OPERATIONS

The following is a list of the operations defined on list<T> objects; n is of type size_type; l, l1, and l2 are of type list<T>; val, val1, and val2 are of type T; ptr1 and ptr2 are pointers to values of type T; it1 and it2 are iterators; and inpIt1, and inpIt2 are input iterators.

Constructors:

list<T> l;	This declaration invokes the default constructor to construct l as an empty list
list<T> l(n);	This declaration initializes l to contain n default values of type **T**
list<T> l(n, val);	This declaration initializes l to contain n copies of val
list<T> l(ptr1, ptr2)	This declaration initializes s to contain the copies of all the T values in the range [ptr1, ptr2)
list<T> l(l1);	This declaration initializes l to contain a copy of l1
l = l1	Assigns a copy of l1 to l
l1 == l2	Returns true if l and l2 contain the same values, and false otherwise
l1 < l2	Returns true if l1 is lexicographically less than l2 l1.size() is less than l2.size() and all the elements of l1 match the first elements of l2; or if val1 and val2 are the first elements of l1 and l2, respectively, that are different, val1 is less than val2 and it returns false otherwise
l.assign(n, val)	Erases l and then inserts n copies of val (default T value if omitted)
l.assign(inpIt1, inpIt2)	Erases l and then inserts copies of the T values in the range [inpIt1, inpIt2)
l.back()	Returns a reference to the last element of l
l.begin()	Returns an iterator positioned at the first element of l
l.empty()	Returns true if l contains no elements, false otherwise
l.end()	Returns an iterator positioned immediately after the last element of l

l.erase(it)	Removes from l the element at the position specifed by it; return type is void
l.erase(it1, it2)	Removes from l the elements in the range [it1, it2); return type is void
l.front()	Returns a reference to the first element of l
l.insert(it, val)	Inserts a copy of val (default T value if omitted) into l at the position specified by it and returns an iterator positioned at this copy
l.insert(it, n, val)	Inserts n copies of val into l at the position specified by it; return type is void
l.insert(it, inpIt1, inpIt2)	Inserts inserts copies of the T values in the range [inpIt1, inpIt2) into l at the position specified by it; return type is void
l.insert(ptr1, ptr2)	Inserts copies of all the T values in the range [ptr1, ptr2) at the position specified by it; return type is void
l.max_size()	Returns the maximum number (of type size_type) of values that l can contain
l.merge(l1)	Merges the elements of l1 into l so that the resulting list is sorted; both l and l1 must have been already sorted (using <); return type is void
l.push_back(val)	Adds a copy of val at the end of l; return type is void
l.push_front(val)	Adds a copy of val at the front of l; return type is void
l.pop_back()	Removes the last element of l; return type is void
l.pop_front()	Removes the first element of l; return type is void
l.rbegin()	Returns a reverse iterator positioned at the last element of l
l.remove(val)	Removes all occurrences of val from l, using == to compare elements; return type is void
l.rend()	Returns a reverse iterator positioned immediately before the first element of l
l.resize(n, val)	Sets the size of l to n; if n.l.size(), copies of val (default T value if omitted) are appended to l; if n,l.size(), the approppriate number of elements is removed from the end of l
l.reverse()	Reverses the order of the elements of l; return type is void
l.size()	Returns the number (of type size_type) of elements l contains
l.sort()	Sorts the elements of l using <; return type is void

`l.splice(it, l1)`	Removes the elements of `l1` and inserts them into `l` at the position specified by `it`; return type is `void`
`l.splice(it, l1, it1)`	Removes the element of `l1` at the position specified by `it1` and inserts it into `l` at the position specified by `it`; return type is `void`
`l.splice(it, l1, it1, it2)`	Removes the elements of `l1` in the range `[it1, it2)` and inserts them into `l` at the position specified by `it`; return type is `void`
`l.swap(l1)`	Swaps the contents of `l` and `l1`; return type is `void`
`l.unique()`	Replaces all repeating sequences of an element of `l` with a single occurence of that element; return type is `void`