Chapter 11: `<list>`

Background

`<list>` **`list`** The header **`<list>`** defines just the template class **`list`**, which is a container that stores its controlled sequence of length **`N`** as a bidirectional linked list of **`N`** nodes, each of which stores a single element. The advantage of a linked list is its flexibility. You can insert and remove elements freely and easily within the list, just by rewriting the forward and backward links in nodes. You can even splice in whole sublists. The list nodes themselves don't move about in memory. As a consequence any iterators you maintain to designate individual nodes remain valid for the life of the node. Similarly, any pointers you maintain to the individual list element itself also remain valid for the life of the node in which the element resides.

The price you pay is sequential access to arbitrary elements in the sequence. To access element number **`i`**, for example, you have to chain from one node to another **`i`** times, beginning with a pointer to the head of the list stored in the container object. You can chain in either direction, but chain you must. So the mean time to locate an arbitrary element increases linearly with the total number of elements in the controlled sequence. Using STL terminology, template class **`list`** supports bidirectional iterators.

Table 11.1, on page 240, shows how template class **`list`** stacks up against the other STL containers. It is the clear winner for all operations that rearrange list elements (insertions, erasures, and replacements). It is the clear loser for all operations that locate arbitrary elements (searches and random access). It also requires a moderately hefty overhead of two pointers per element, the forward and backward links stored in each node.

`splice` **`sort`** **`merge`** Template class **`list`** defines several member functions that take advantage of its peculiar properties. For example, you can splice elements from one list into another, sort a list, or merge one ordered list into another. All these operations simply restitch links between list nodes. No copying occurs. The payoff can be significant for a list of elements that are expensive to copy — because they are large or have nontrivial copy semantics.

exception safety Template class **`list`** has an additional virtue. It alone of the template containers promises to behave predictably in the presence of exceptions thrown by programmer-supplied code. Other containers provide a weaker guarantee. (See Chapter 9: Containers.) For any container, an exception thrown during execution of a member function leaves the container in a consistent state, suitable for destruction; and the container does not lose track of allocated storage. But for many operations, particularly those that affect multiple elements, the exact state of the container is unspecified when the exception is rethrown. **list**, by contrast, guarantees for most member functions that any interrupted member function call leaves the container in its original state when it rethrows the exception.

So in summary, you use template class **list** when you need flexibility in rearranging sequences of elements, and in keeping track of individual elements by storing iterators that remain valid across rearrangements. You also use template class **list** when you need greater determinism in the presence of exceptions. On the other hand, locating arbitrary elements within a **list** object is relatively expensive, even if the list is kept in order, since you have to perform a linear search each time. Consider other containers if more rapid access is important.

Functional Description

```
namespace std {
template<class T, class A>
    class list;

        // TEMPLATE FUNCTIONS
template<class T, class A>
    bool operator==(
        const list<T, A>& lhs,
        const list<T, A>& rhs);
template<class T, class A>
    bool operator!=(
        const list<T, A>& lhs,
        const list<T, A>& rhs);
template<class T, class A>
    bool operator<(
        const list<T, A>& lhs,
        const list<T, A>& rhs);
template<class T, class A>
    bool operator>(
        const list<T, A>& lhs,
        const list<T, A>& rhs);
template<class T, class A>
    bool operator<=(
        const list<T, A>& lhs,
        const list<T, A>& rhs);
template<class T, class A>
    bool operator>=(
        const list<T, A>& lhs,
        const list<T, A>& rhs);
template<class T, class A>
    void swap(
        list<T, A>& lhs,
        list<T, A>& rhs);
    };
```

Include the STL standard header **<list>** to define the container template class **list** and several supporting templates.

▫ list

```
template<class T, class A = allocator<T> >
    class list {
public:
    typedef A allocator_type;
    typedef typename A::pointer pointer;
    typedef typename A::const_pointer
        const_pointer;
    typedef typename A::reference reference;
    typedef typename A::const_reference const_reference;
    typedef typename A::value_type value_type;
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef T2 size_type;
    typedef T3 difference_type;
    typedef reverse_iterator<const_iterator>
        const_reverse_iterator;
    typedef reverse_iterator<iterator>
        reverse_iterator;
    list();
    explicit list(const A& al);
    explicit list(size_type n);
    list(size_type n, const T& v);
    list(size_type n, const T& v, const A& al);
    list(const list& x);
    template<class InIt>
        list(InIt first, InIt last);
    template<class InIt>
        list(InIt first, InIt last, const A& al);
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;
    void resize(size_type n);
    void resize(size_type n, T x);
    size_type size() const;
    size_type max_size() const;
    bool empty() const;
    A get_allocator() const;
    reference front();
    const_reference front() const;
    reference back();
    const_reference back() const;
    void push_front(const T& x);
    void pop_front();
    void push_back(const T& x);
    void pop_back();
    template<class InIt>
        void assign(InIt first, InIt last);
    void assign(size_type n, const T& x);
    iterator insert(iterator it, const T& x);
    void insert(iterator it, size_type n, const T& x);
    template<class InIt>
        void insert(iterator it, InIt first, InIt last);
    iterator erase(iterator it);
    iterator erase(iterator first, iterator last);
    void clear();
    void swap(list& x);
    void splice(iterator it, list& x);
    void splice(iterator it, list& x, iterator first);
    void splice(iterator it, list& x, iterator first,
        iterator last);
    void remove(const T& x);
    templace<class Pred>
        void remove_if(Pred pr);
    void unique();
    template<class Pred>
        void unique(Pred pr);
    void merge(list& x);
    template<class Pred>
        void merge(list& x, Pred pr);
    void sort();
    template<class Pred>
        void sort(Pred pr);
    void reverse();
    };
```

The template class describes an object that controls a varying-length sequence of elements of type **T**. The sequence is stored as a bidirectional linked list of elements, each containing a member of type **T**.

The object allocates and frees storage for the sequence it controls through a stored allocator object of class **A**. Such an allocator object must have the same external interface as an object of template class **allocator**. Note that the stored allocator object is *not* copied when the container object is assigned.

List reallocation occurs when a member function must insert or erase elements of the controlled sequence. In all such cases, only iterators or references that point at erased portions of the controlled sequence become *invalid*.

All additions to the controlled sequence occur as if by calls to **insert**, which is the only member function that calls the constructor **T(const T&)**. If such an expression throws an exception, the container object inserts no new elements and rethrows the exception. Thus, an object of template class **list** is left in a known state when such exceptions occur.

▫ **list::allocator_type**

```
typedef A allocator_type;
```

The type is a synonym for the template parameter **A**.

▫ **list::assign**

```
template<class InIt>
    void assign(InIt first, InIt last);
void assign(size_type n, const T& x);
```

If **InIt** is an integer type, the first member function behaves the same as **assign((size_type)first, (T)last)**. Otherwise, the first member function replaces the sequence controlled by ***this** with the sequence **[first, last)**, which must *not* overlap the initial controlled sequence. The second member function replaces the sequence controlled by ***this** with a repetition of **n** elements of value **x**.

▫ **list::back**

```
reference back();
const_reference back() const;
```

The member function returns a reference to the last element of the controlled sequence, which must be non-empty.

▫ **list::begin**

```
const_iterator begin() const;
iterator begin();
```

The member function returns a bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

▫ **list::clear**

```
void clear();
```

The member function calls `erase( begin(), end())`.

▫ **list::const_iterator**

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T1`.

▫ **list::const_pointer**

```
typedef typename A::const_pointer
    const_pointer;
```

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

▫ **list::const_reference**

```
typedef typename A::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

▫ **list::const_reverse_iterator**

```
typedef reverse_iterator<const_iterator>
    const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse bidirectional iterator for the controlled sequence.

▫ **list::difference_type**

```
typedef T3 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type `T3`.

▫ **list::empty**

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

▫ **`list::end`**

```
const_iterator end() const;
iterator end();
```

The member function returns a bidirectional iterator that points just beyond the end of the sequence.

▫ **`list::erase`**

```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
```

The first member function removes the element of the controlled sequence pointed to by `it`. The second member function removes the elements of the controlled sequence in the range `[first, last)`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

Erasing `N` elements causes `N` destructor calls. No reallocation occurs, so iterators and references become invalid only for the erased elements.

The member functions never throw an exception.

▫ **`list::front`**

```
reference front();
const_reference front() const;
```

The member function returns a reference to the first element of the controlled sequence, which must be non-empty.

▫ **`list::get_allocator`**

```
A get_allocator() const;
```

The member function returns the stored allocator object.

▫ **`list::insert`**

```
iterator insert(iterator it, const T& x);
void insert(iterator it, size_type n, const T& x);
template<class InIt>
    void insert(iterator it, InIt first, InIt last);
```

Each of the member functions inserts, before the element pointed to by `it` in the controlled sequence, a sequence specified by the remaining operands. The first member function inserts a single element with value `x` and returns an iterator that points to the newly inserted element. The second member function inserts a repetition of `n` elements of value `x`.

If `InIt` is an integer type, the last member function behaves the same as `insert(it, (size_type)first, (T)last)`. Otherwise, the last member function inserts the sequence `[first, last)`, which must *not* overlap the initial controlled sequence.

Inserting `N` elements causes `N` constructor calls. No reallocation occurs, so no iterators or references become invalid.

If an exception is thrown during the insertion of one or more elements, the container is left unaltered and the exception is rethrown.

▫ **list::iterator**

```
typedef T0 iterator;
```

The type describes an object that can serve as a bidirectional iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T0`.

▫ **list::list**

```
list();
explicit list(const A& al);
explicit list(size_type n);
list(size_type n, const T& v);
list(size_type n, const T& v,
    const A& al);
list(const list& x);
template<class InIt>
    list(InIt first, InIt last);
template<class InIt>
    list(InIt first, InIt last, const A& al);
```

All constructors store an allocator object and initialize the controlled sequence. The allocator object is the argument `al`, if present. For the copy constructor, it is `x.get_allocator()`. Otherwise, it is `A()`.

The first two constructors specify an empty initial controlled sequence. The third constructor specifies a repetition of `n` elements of value `T()`. The fourth and fifth constructors specify a repetition of `n` elements of value `x`. The sixth constructor specifies a copy of the sequence controlled by `x`. If `InIt` is an integer type, the last two constructors specify a repetition of `(size_type)first` elements of value `(T)last`. Otherwise, the last two constructors specify the sequence `[first, last)`. None of the constructors perform any interim reallocations.

▫ **list::max_size**

```
size_type max_size() const;
```

The member function returns the length of the longest sequence that the object can control.

▫ **list::merge**

```
void merge(list& x);
template<class Pred>
    void merge(list& x, Pred pr);
```

Both member functions remove all elements from the sequence controlled by `x` and insert them in the controlled sequence. Both sequences must be ordered by the same predicate, described below. The resulting sequence is also ordered by that predicate.

For the iterators `Pi` and `Pj` designating elements at positions `i` and `j`, the first member function imposes the order `!(*Pj < *Pi)` whenever `i < j`.

(The elements are sorted in *ascending* order.) The second member function imposes the order `!pr(*Pj, *Pi)` whenever `i < j`.

No pairs of elements in the original controlled sequence are reversed in the resulting controlled sequence. If a pair of elements in the resulting controlled sequence compares equal (`!(*Pi < *Pj) && !(*Pj < *Pi)`), an element from the original controlled sequence appears before an element from the sequence controlled by **x**.

An exception occurs only if **pr** throws an exception. In that case, the controlled sequence is left in unspecified order and the exception is rethrown.

▫ **list::pointer**

```
typedef typename A::pointer pointer;
```

The type describes an object that can serve as a pointer to an element of the controlled sequence.

▫ **list::pop_back**

```
void pop_back();
```

The member function removes the last element of the controlled sequence, which must be non-empty.

The member function never throws an exception.

▫ **list::pop_front**

```
void pop_front();
```

The member function removes the first element of the controlled sequence, which must be non-empty.

The member function never throws an exception.

▫ **list::push_back**

```
void push_back(const T& x);
```

The member function inserts an element with value **x** at the end of the controlled sequence.

If an exception is thrown, the container is left unaltered and the exception is rethrown.

▫ **list::push_front**

```
void push_front(const T& x);
```

The member function inserts an element with value **x** at the beginning of the controlled sequence.

If an exception is thrown, the container is left unaltered and the exception is rethrown.

▫ **list::rbegin**

```
const_reverse_iterator rbegin() const;
reverse_iterator rbegin();
```

The member function returns a reverse bidirectional iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

▫ **list::reference**

```
typedef typename A::reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

▫ **list::remove**

```
void remove(const T& x);
```

The member function removes from the controlled sequence all elements, designated by the iterator `P`, for which `*P == x`.

The member function never throws an exception.

▫ **list::remove_if**

```
templace<class Pred>
    void remove_if(Pred pr);
```

The member function removes from the controlled sequence all elements, designated by the iterator `P`, for which `pr(*P)` is true.

An exception occurs only if `pr` throws an exception. In that case, the controlled sequence is left in an unspecified state and the exception is rethrown.

▫ **list::rend**

```
const_reverse_iterator rend() const;
reverse_iterator rend();
```

The member function returns a reverse bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, it designates the end of the reverse sequence.

▫ **list::resize**

```
void resize(size_type n);
void resize(size_type n, T x);
```

The member functions both ensure that `size()` henceforth returns `n`. If it must make the controlled sequence longer, the first member function appends elements with value `T()`, while the second member function appends elements with value `x`. To make the controlled sequence shorter, both member functions call `erase(begin() + n, end())`.

▫ **list::reverse**

```
void reverse();
```

The member function reverses the order in which elements appear in the controlled sequence.

▫ **list::reverse_iterator**

```
typedef reverse_iterator<iterator>
    reverse_iterator;
```

The type describes an object that can serve as a reverse bidirectional iterator for the controlled sequence.

▫ **`list::size`**

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

▫ **`list::size_type`**

```
typedef T2 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type `T2`.

▫ **`list::sort`**

```
void sort();
template<class Pred>
    void sort(Pred pr);
```

Both member functions order the elements in the controlled sequence by a predicate, described below.

For the iterators `Pi` and `Pj` designating elements at positions `i` and `j`, the first member function imposes the order `!(*Pj < *Pi)` whenever `i < j`. (The elements are sorted in *ascending* order.) The member template function imposes the order `!pr(*Pj, *Pi)` whenever `i < j`. No ordered pairs of elements in the original controlled sequence are reversed in the resulting controlled sequence. (The sort is stable.)

An exception occurs only if `pr` throws an exception. In that case, the controlled sequence is left in unspecified order and the exception is rethrown.

▫ **`list::splice`**

```
void splice(iterator it, list& x);
void splice(iterator it, list& x, iterator first);
void splice(iterator it, list& x, iterator first,
    iterator last);
```

The first member function inserts the sequence controlled by `x` before the element in the controlled sequence pointed to by `it`. It also removes all elements from `x`. (`&x` must not equal `this`.)

The second member function removes the element pointed to by `first` in the sequence controlled by `x` and inserts it before the element in the controlled sequence pointed to by `it`. (If `it == first || it == ++first`, no change occurs.)

The third member function inserts the subrange designated by `[first, last)` from the sequence controlled by `x` before the element in the controlled sequence pointed to by `it`. It also removes the original subrange from the sequence controlled by `x`. (If `&x == this`, the range `[first, last)` must not include the element pointed to by `it`.)

If the third member function inserts **N** elements, and **&x != this**, an object of class **iterator** is incremented **N** times. For all **splice** member functions, If **get_allocator() == str.get_allocator()**, no exception occurs. Otherwise, in this implementation, a copy and a destructor call also occur for each inserted element.

In all cases, only iterators or references that point at spliced elements become *invalid*.

▫ **list::swap**

```
void swap(list& x);
```

The member function swaps the controlled sequences between ***this** and **x**. If **get_allocator() == x.get_allocator()**, it does so in constant time, it throws no exceptions, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

▫ **list::unique**

```
void unique();
template<class Pred>
    void unique(Pred pr);
```

The first member function removes from the controlled sequence every element that compares equal to its preceding element. For the iterators **Pi** and **Pj** designating elements at positions **i** and **j**, the second member function removes every element for which **i + 1 == j && pr(*Pi, *Pj)**.

For a controlled sequence of length **N** (> 0), the predicate **pr(*Pi, *Pj)** is evaluated **N - 1** times.

An exception occurs only if **pr** throws an exception. In that case, the controlled sequence is left in an unspecified state and the exception is rethrown.

▫ **list::value_type**

```
typedef typename A::value_type value_type;
```

The type is a synonym for the template parameter **T**.

▫ **operator!=**

```
template<class T, class A>
    bool operator!=(
        const list <T, A>& lhs,
        const list <T, A>& rhs);
```

The template function returns **!(lhs == rhs)**.

▫ **operator==**

```
template<class T, class A>
    bool operator==(
        const list <T, A>& lhs,
        const list <T, A>& rhs);
```

The template function overloads `operator==` to compare two objects of template class `list`. The function returns `lhs.size() == rhs.size() && equal(lhs. begin(), lhs. end(), rhs.begin())`.

- **`operator<`**

```
template<class T, class A>
    bool operator<(
        const list <T, A>& lhs,
        const list <T, A>& rhs);
```

The template function overloads `operator<` to compare two objects of template class `list`. The function returns `lexicographical_compare(lhs. begin(), lhs. end(), rhs.begin(), rhs.end())`.

- **`operator<=`**

```
template<class T, class A>
    bool operator<=(
        const list <T, A>& lhs,
        const list <T, A>& rhs);
```

The template function returns `!(rhs < lhs)`.

- **`operator>`**

```
template<class T, class A>
    bool operator>(
        const list <T, A>& lhs,
        const list <T, A>& rhs);
```

The template function returns `rhs < lhs`.

- **`operator>=`**

```
template<class T, class A>
    bool operator>=(
        const list <T, A>& lhs,
        const list <T, A>& rhs);
```

The template function returns `!(lhs < rhs)`.

- **`swap`**

```
template<class T, class A>
    void swap(
        list <T, A>& lhs,
        list <T, A>& rhs);
```

The template function executes `lhs.swap(rhs)`.

Using `<list>`

`list` Include the header `<list>` to make use of template class `list`. You can specialize `list` to store elements of type `T` by writing a type definition such as:

```
typedef list<T, allocator<T> > Mycont;
```

Using a default template argument, you can omit the second argument.

Template class `list` supports all the common operations on containers, as we described in Chapter 9: Containers. (See the discussion beginning on page 248.) We summarize here only those properties peculiar to template class **list**.

constructors To construct an object of class **list<T, A>**, you can write any of:

- **list()** to declare an empty list.
- **list(al)** as above, also storing the allocator object **al**.
- **list(n)** to declare a list with **n** elements, each constructed with the default constructor **T()**.
- **list(n, val)** to declare a list with **n** elements, each constructed with the copy constructor **T(val)**.
- **list(n, val, al)** as above, also storing the allocator object **al**.
- **list(first, last)** to declare a list with initial elements copied from the sequence designated by **[first, last)**.
- **list(first, last, al)** as above, also storing the allocator object **al**.

If you have specialized the template class for an allocator of type **allocator<T>**, which is the customary (and default) thing to do, there is nothing to be gained by specifying an explicit allocator argument **al**. Such an argument matters only for some allocators that the program defines explicitly. (See the discussion of allocators in Chapter 4: **<memory>**.)

The following descriptions all assume that **cont** is an object of class **list<T, A>**.

resize **clear** To change the length of the controlled sequence to **n** elements, call **cont.resize(n)**. Excess elements are erased. If the sequence must be extended, elements with the value **T()** are inserted as needed at the end. You can also call **cont.resize(n, val)** to extend the sequence with elements that store the value **val**. To remove all elements, call **cont.clear()**.

front **back** To access the first element of the controlled sequence, call **cont.front()**. To access the last element, call **cont.back()**. If **cont** is not a const object, the expression is an lvalue, so you can alter the value stored in the element by writing an expression such as **cont.front() = T()**. If the sequence is empty, however, these expressions have undefined behavior.

push_back **push_front** **pop_back** **pop_front** To append an object with stored value **x**, call **cont.push_back(x)**; to prepend the object, call **cont.push_front(x)**. To remove the last element, call **cont.pop_back()**; to remove the first, call **cont.pop_front()**. In either case, the sequence must, of course, be non-empty or the call has undefined bahavior.

assign To replace the controlled sequence with the elements from a sequence designated by **[first, last)**, call **cont.assign(first, last)**. The sequence must not be part of the initial controlled sequence. To replace the controlled sequence with **n** copies of the value **x**, call **cont.assign(n, x)**.

insert To insert an element storing the value **x** before the element designated by the iterator **it**, call **cont.insert(it, x)**. The return value is an iterator designating the inserted element. To insert the elements of a sequence designated by `[first, last)` before the element designated by the iterator `it`, call `cont.insert(it, first, last)`. The sequence must not be any part of the initial controlled sequence. To insert `n` copies of the value `x`, call `cont.insert(it, n, x)`.

`erase` To erase the element designated by the iterator `it`, call `cont.erase(it)`. The return value is an iterator designating the element just beyond the erased element. To erase a range of elements designated by `[first, last)`, call `cont.erase(first, last)`.

You can also perform a number of operations on a `list` object that take advantage of its unique representation.

`splice` You can *splice* a sequence of list nodes into a list. The spliced nodes are removed from their existing positions. No elements are copied — a splice simply rewrites links within the nodes as needed.

- To splice the entire contents of the object `cont2` before the element designated by the iterator `it`, call `cont.splice(it, cont2)`. The two list objects must, of course, be distinct.
- To splice the node designated by the iterator `p` in the object `cont2` before the node designated by the iterator `it`, call `cont.splice(it, cont2, p)`. The two list objects need not be distinct. (Splicing a node before itself causes no change
- To splice the sequence of nodes designated by `[first, last)` in the object `cont2` before the element designated by the iterator `it`, call `cont.splice(it, cont2, first, last)`. The two list objects need not be distinct, but `it` must not designate any of the nodes to be spliced.

If a node migrates from one list to another as a result of a splice, it is important that the two lists have allocator objects that compare equal. This requirement is always met by default allocators. Otherwise, the node cannot be safely erased by its new owner.

`remove`
`remove_if` To remove all elements that compare equal (using `operator==`) to the value `v`, call `cont.remove(v)`. To remove each element `x` for which `pr(x)` is true, call `cont.remove_if(pr)`.

`unique` To remove all but the first of each subsequence of elements that compare equal (using `operator==`), call `cont.unique()`. To replace `operator==` with `pr` as the comparison function, call `cont.unique(pr)`. Note that `unique` is generally more effective if you first sort the sequence, so that all groups of equal elements are adjacent.

`sort` To sort the sequence controlled by `cont` call `cont.sort()`. The resulting sequence is ordered by `operator<`. (Elements are left in ascending order.) The sort operation is performed as a succession of splices. To replace `operator<` with `pr` as the ordering function, call `cont.sort(pr)`.

`merge` To merge the ordered sequence controlled by `cont2` into the ordered sequence controlled by `cont`, call `cont.merge(cont2)`. The merge operation is performed as a succession of splices, so `cont2` is left empty (unless it is the same object as `cont`). Both sequences must be ordered by `operator<`

for the merge to work properly. (Sorting them, as above, does the job.) To replace **operator<** with **pr** as the ordering function, call **cont.merge(cont2, pr)**.

reverse Finally, you can reverse the controlled sequence by calling **reverse()**. The operation is performed as a succession of splices.

Implementing <list>

list Figures 11.1 through 11.11 show the file **list**. It defines template class **list**, along with a few template functions that take list operands.

A **list** object stores a pointer and a count to represent the controlled sequence. Besides the allocator objects, described below, a **list** stores two objects:

Head ■ **Head** is a pointer to a dummy "head" node that in turn points forward to the beginning of the controlled sequence and backward to its end.

Size ■ **Size** counts the number of elements in the list.

The dummy head node greatly simplifies many list operations. It eliminates the need for most special handling of the first and last nodes in the list. The price you pay for a head node is unused storage for one list element. This is typically a small price to pay, but for a list of very large elements it can be significant.

That's the easy part, familiar to anyone who has ever managed a linked list. The object also stores three different allocator objects. Here's where the real trickery comes in.

In STL containers, all controlled storage is nominally managed through the allocator object specified when you construct the container object. As we mentioned in the previous chapter, in conjunction with **vector** objects, allocators were originally invented to allocate and free arrays of objects of some element type **T**. They have since been made far more ambitious, and complex. Template class **vector** can get away with the simplest usages, but not so template class **list**, for several subtle reasons.

Node To begin at the beginning, consider how you normally manage a bidirectional linked list. You need to define a class **Node** that stores all the data required of a list node. To store an element of type **T**, you can write:

```
class Node {
    Node *Next, *Prev;
    T Value;
    };
```

The one small trick you must make use of dates back to the earliest days of the C language, from which C++ evolved. The forward link **Next** and the backward link **Prev** are both self-referential pointers — they point at other objects of the same type as the object in which they reside. No sweat. You can declare a pointer to an incomplete type inside a structured type, even if that type is the one you're busy completing.

Figure 11.1: `list` **Part 1**

```
// list standard header
#ifndef LIST_
#define LIST_
#include <functional>
#include <memory>
#include <stdexcept>
namespace std {

          // TEMPLATE CLASS List_nod
template<class Ty, class A>
     class List_nod {
protected:
     typedef typename A::template
          rebind<void>::other::pointer Genptr;
     struct Node;
     friend struct Node;
     struct Node {
          Genptr Next, Prev;
          Ty Value;
          };
     List_nod(A Al)
          : Alnod(Al) {}
     typename A::template rebind<Node>::other Alnod;
     };

          // TEMPLATE CLASS List_ptr
template<class Ty, class A>
     class List_ptr : public List_nod<Ty, A> {
protected:
     typedef typename List_nod<Ty, A>::Node Node;
     typedef typename A::template
          rebind<Node>::other::pointer Nodeptr;
     List_ptr(A Al)
          : List_nod<Ty, A>(Al), Alptr(Al) {}
     typename A::template rebind<Nodeptr>::other Alptr;
     };

          // TEMPLATE CLASS List_val
template<class Ty, class A>
     class List_val : public List_ptr<Ty, A> {
protected:
     List_val(A Al = A())
          : List_ptr<Ty, A>(Al), Alval(Al) {}
     typedef typename A::template
          rebind<Ty>::other Alty;
     Alty Alval;
     };

          // TEMPLATE CLASS list
template<class Ty, class Ax = allocator<Ty> >
     class list : public List_val<Ty, Ax> {
public:
     typedef list<Ty, Ax> Myt;
     typedef List_val<Ty, Ax> Mybase;
     typedef typename Mybase::Alty A;
```

Figure 11.2: `list` **Part 2**

```
protected:
     typedef typename List_nod<Ty, A>::Genptr Genptr;
     typedef typename List_nod<Ty, A>::Node Node;
     typedef typename A::template
          rebind::<Node>::other::pointer Nodeptr;
     struct Acc;
     friend struct Acc;
     struct Acc {
          typedef typename A::template
               rebind::<Nodeptr>::other::reference Nodepref;
          typedef typename A::reference Vref;
          static Nodepref Next(Nodeptr P)
               {return ((Nodepref)(*P).Next); }
          static Nodepref Prev(Nodeptr P)
               {return ((Nodepref)(*P).Prev); }
          static Vref Value(Nodeptr P)
               {return ((Vref)(*P).Value); }
          };
public:
     typedef A allocator_type;
     typedef typename A::size_type size_type;
     typedef typename A::difference_type Dift;
     typedef Dift difference_type;
     typedef typename A::pointer Tptr;
     typedef typename A::const_pointer Ctptr;
     typedef Tptr pointer;
     typedef Ctptr const_pointer;
     typedef typename A::reference Reft;
     typedef Reft reference;
     typedef typename A::const_reference const_reference;
     typedef typename A::value_type value_type;

          // CLASS iterator
     class iterator;
     friend class iterator;
     class iterator : public Bidit<Ty, Dift, Tptr, Reft> {
     public:
          typedef Bidit<Ty, Dift, Tptr, Reft> Mybase;
          typedef typename Mybase::iterator_category
               iterator_category;
          typedef typename Mybase::value_type value_type;
          typedef typename Mybase::difference_type
               difference_type;
          typedef typename Mybase::pointer pointer;
          typedef typename Mybase::reference reference;
          iterator()
               : Ptr(0) {}
          iterator(Nodeptr P)
               : Ptr(P) {}
          reference operator*() const
               {return (Acc::Value(Ptr)); }
          Tptr operator->() const
               {return (&**this); }
          iterator& operator++()
```

Figure 11.3: `list` **Part 3**

```
                {Ptr = Acc::Next(Ptr);
                return (*this); }
            iterator operator++(int)
                {iterator Tmp = *this;
                ++*this;
                return (Tmp); }
            iterator& operator--()
                {Ptr = Acc::Prev(Ptr);
                return (*this); }
            iterator operator--(int)
                {iterator Tmp = *this;
                --*this;
                return (Tmp); }
            bool operator==(const iterator& X) const
                {return (Ptr == X.Ptr); }
            bool operator!=(const iterator& X) const
                {return (!(*this == X)); }
            Nodeptr Mynode() const
                {return (Ptr); }
        protected:
            Nodeptr Ptr;
            };

            // CLASS const_iterator
        class const_iterator;
        friend class const_iterator;
        class const_iterator
            : public Bidit<Ty, Dift, Ctptr, const_reference> {
        public:
            typedef Bidit<Ty, Dift, Ctptr, const_reference>
                Mybase;
            typedef typename Mybase::iterator_category
                iterator_category;
            typedef typename Mybase::value_type value_type;
            typedef typename Mybase::difference_type
                difference_type;
            typedef typename Mybase::pointer pointer;
            typedef typename Mybase::reference reference;
            const_iterator()
                : Ptr(0) {}
            const_iterator(Nodeptr P)
                : Ptr(P) {}
            const_iterator(const typename list<Ty, Ax>::iterator& X)
                : Ptr(X.Mynode()) {}
            const_reference operator*() const
                {return (Acc::Value(Ptr)); }
            Ctptr operator->() const
                {return (&**this); }
            const_iterator& operator++()
                {Ptr = Acc::Next(Ptr);
                return (*this); }
            const_iterator operator++(int)
                {const_iterator Tmp = *this;
                ++*this;
                return (Tmp); }
```

Figure 11.4: `list` **Part 4**

```
          const_iterator& operator--()
               {Ptr = Acc::Prev(Ptr);
               return (*this); }
          const_iterator operator--(int)
               {const_iterator Tmp = *this;
               --*this;
               return (Tmp); }
          bool operator==(const const_iterator& X) const
               {return (Ptr == X.Ptr); }
          bool operator!=(const const_iterator& X) const
               {return (!(*this == X)); }
          Nodeptr Mynode() const
               {return (Ptr); }
     protected:
          Nodeptr Ptr;
          };

     typedef std::reverse_iterator<iterator>
          reverse_iterator;
     typedef std::reverse_iterator<const_iterator>
          const_reverse_iterator;

     list()
          : Mybase(), Head(Buynode()), Size(0)
          {}
     explicit list(const A& Al)
          : Mybase(Al), Head(Buynode()), Size(0)
          {}
     explicit list(size_type N)
          : Mybase(), Head(Buynode()), Size(0)
          {insert(begin(), N, Ty()); }
     list(size_type N, const Ty& V)
          : Mybase(), Head(Buynode()), Size(0)
          {insert(begin(), N, V); }
     list(size_type N, const Ty& V, const A& Al)
          : Mybase(Al), Head(Buynode()), Size(0)
          {insert(begin(), N, V); }
     list(const Myt& X)
          : Mybase(X.Alval),
               Head(Buynode()), Size(0)
          {insert(begin(), X.begin(), X.end()); }
     template<class It>
          list(It F, It L)
          : Mybase(), Head(Buynode()), Size(0)
          {Construct(F, L, Iter_cat(F)); }
     template<class It>
          list(It F, It L, const A& Al)
          : Mybase(Al), Head(Buynode()), Size(0)
          {Construct(F, L, Iter_cat(F)); }
     template<class It>
          void Construct(It F, It L, Int_iterator_tag)
          {insert(begin(), (size_type)F, (Ty)L); }
     template<class It>
          void Construct(It F, It L, input_iterator_tag)
          {insert(begin(), F, L); }
```

Figure 11.5: `list` **Part 5**

```
~list()
     {erase(begin(), end());
     Freenode(Head);
     Head = 0, Size = 0; }
Myt& operator=(const Myt& X)
     {if (this != &X)
          assign(X.begin(), X.end());
     return (*this); }
iterator begin()
     {return (iterator(Head == 0 ? 0
          : Acc::Next(Head))); }
const_iterator begin() const
     {return (const_iterator(Head == 0 ? 0
          : Acc::Next(Head))); }
iterator end()
     {return (iterator(Head)); }
const_iterator end() const
     {return (const_iterator(Head)); }
reverse_iterator rbegin()
     {return (reverse_iterator(end())); }
const_reverse_iterator rbegin() const
     {return (const_reverse_iterator(end())); }
reverse_iterator rend()
     {return (reverse_iterator(begin())); }
const_reverse_iterator rend() const
     {return (const_reverse_iterator(begin())); }
void resize(size_type N)
     {resize(N, Ty()); }
void resize(size_type N, Ty X)
     {if (size() < N)
          insert(end(), N - size(), X);
     else
          while (N < size())
               pop_back(); }
size_type size() const
     {return (Size); }
size_type max_size() const
     {return (Mybase::Alval.max_size()); }
bool empty() const
     {return (size() == 0); }
allocator_type get_allocator() const
     {return (Mybase::Alval); }
reference front()
     {return (*begin()); }
const_reference front() const
     {return (*begin()); }
reference back()
     {return (*(--end())); }
const_reference back() const
     {return (*(--end())); }
void push_front(const Ty& X)
     {Insert(begin(), X); }
void pop_front()
     {erase(begin()); }
```

Figure 11.6:
`list`
Part 6

```
void push_back(const Ty& X)
     {Insert(end(), X); }
void pop_back()
     {erase(--end()); }
template<class It>
     void assign(It F, It L)
     {Assign(F, L, Iter_cat(F)); }
template<class It>
     void Assign(It F, It L, Int_iterator_tag)
     {assign((size_type)F, (Ty)L); }
template<class It>
     void Assign(It F, It L, input_iterator_tag)
     {erase(begin(), end());
     insert(begin(), F, L); }
void assign(size_type N, const Ty& X)
     {Ty Tx = X;
     erase(begin(), end());
     insert(begin(), N, Tx); }
iterator insert(iterator P, const Ty& X)
     {Insert(P, X);
     return (--P); }
void Insert(iterator P, const Ty& X)
     {Nodeptr S = P.Mynode();
     Nodeptr Snew = Buynode(S, Acc::Prev(S));
     Incsize(1);
     try {
     Mybase::Alval.construct(&Acc::Value(Snew), X);
     } catch (...) {
     --Size;
     Freenode(Snew);
     throw;
     }
     Acc::Prev(S) = Snew;
     Acc::Next(Acc::Prev(Snew)) = Snew; }
void insert(iterator P, size_type M, const Ty& X)
     {size_type N = M;
     try {
     for (; 0 < M; --M)
          Insert(P, X);
     } catch (...) {
     for (; M < N; ++M)
          {iterator Pm = P;
          erase(--Pm); }
     throw;
     }}
template<class It>
     void insert(iterator P, It F, It L)
     {Insert(P, F, L, Iter_cat(F)); }
template<class It>
     void Insert(iterator P, It F, It L,
          Int_iterator_tag)
     {insert(P, (size_type)F, (Ty)L); }
template<class It>
     void Insert(iterator P, It F, It L,
          input_iterator_tag)
```

Figure 11.7: `list` **Part 7**

```
          {size_type N = 0;
          try {
          for (; F != L; ++F, ++N)
               Insert(P, *F);
          } catch (...) {
          for (; 0 < N; --N)
               {iterator Pm = P;
               erase(--Pm); }
          throw;
          }}
     template<class It>
          void Insert(iterator P, It F, It L,
               forward_iterator_tag)
          {It Fs = F;
          try {
          for (; F != L; ++F)
               Insert(P, *F);
          } catch (...) {
          for (; Fs != F; ++Fs)
               {iterator Pm = P;
               erase(--Pm); }
          throw;
          }}
     iterator erase(iterator P)
          {Nodeptr S = (P++).Mynode();
          Acc::Next(Acc::Prev(S)) = Acc::Next(S);
          Acc::Prev(Acc::Next(S)) = Acc::Prev(S);
          Mybase::Alval.destroy(&Acc::Value(S));
          Freenode(S);
          --Size;
          return (P); }
     iterator erase(iterator F, iterator L)
          {while (F != L)
               erase(F++);
          return (F); }
     void clear()
          {erase(begin(), end()); }
     void swap(Myt& X)
          {if (Mybase::Alval == X.Alval)
               {std::swap(Head, X.Head);
               std::swap(Size, X.Size); }
          else
               {iterator P = begin();
               splice(P, X);
               X.splice(X.begin(), *this, P, end()); }}
     void splice(iterator P, Myt& X)
          {if (this != &X && !X.empty())
               {Splice(P, X, X.begin(), X.end(), X.Size); }}
     void splice(iterator P, Myt& X, iterator F)
          {iterator L = F;
          if (F != X.end() && P != F && P != ++L)
               {Splice(P, X, F, L, 1); }}
     void splice(iterator P, Myt& X, iterator F, iterator L)
          {if (F != L && P != L)
               {size_type N = 0;
```

Figure 11.8: list Part 8

```
                for (iterator Fs = F; Fs != L; ++Fs, ++N)
                     if (Fs == P)
                          return;                // else granny knot
                Splice(P, X, F, L, N); }}
     void remove(const Ty& V)
          {iterator L = end();
          for (iterator F = begin(); F != L; )
               if (*F == V)
                    erase(F++);
               else
                    ++F; }
     template<class Pr1>
          void remove_if(Pr1 Pr)
          {iterator L = end();
          for (iterator F = begin(); F != L; )
               if (Pr(*F))
                    erase(F++);
               else
                    ++F; }
     void unique()
          {iterator F = begin(), L = end();
          if (F != L)
               for (iterator M = F; ++M != L; M = F)
                    if (*F == *M)
                         erase(M);
                    else
                         F = M; }
     template<class Pr2>
          void unique(Pr2 Pr)
          {iterator F = begin(), L = end();
          if (F != L)
               for (iterator M = F; ++M != L; M = F)
                    if (Pr(*F, *M))
                         erase(M);
                    else
                         F = M; }
     void merge(Myt& X)
          {if (&X != this)
               {iterator F1 = begin(), L1 = end();
               iterator F2 = X.begin(), L2 = X.end();
               while (F1 != L1 && F2 != L2)
                    if (*F2 < *F1)
                         {iterator Mid2 = F2;
                         Splice(F1, X, F2, ++Mid2, 1);
                         F2 = Mid2; }
                    else
                         ++F1;
               if (F2 != L2)
                    Splice(L1, X, F2, L2, X.Size); }}
     template<class Pr3>
          void merge(Myt& X, Pr3 Pr)
          {if (&X != this)
               {iterator F1 = begin(), L1 = end();
               iterator F2 = X.begin(), L2 = X.end();
```

Figure 11.9:
`list`
Part 9

```
            while (F1 != L1 && F2 != L2)
                 if (Pr(*F2, *F1))
                      {iterator Mid2 = F2;
                      Splice(F1, X, F2, ++Mid2, 1);
                      F2 = Mid2; }
                 else
                      ++F1;
            if (F2 != L2)
                 Splice(L1, X, F2, L2, X.Size); }}
  void sort()
       {if (2 <= size())
            {const size_t MAXN = 25;
            Myt X(Mybase::Alval), Arr[MAXN + 1];
            size_t N = 0;
            while (!empty())
                 {X.splice(X.begin(), *this, begin());
                 size_t I;
                 for (I = 0; I < N && !Arr[I].empty(); ++I)
                      {Arr[I].merge(X);
                      Arr[I].swap(X); }
                 if (I == MAXN)
                      Arr[I - 1].merge(X);
                 else
                      {Arr[I].swap(X);
                      if (I == N)
                           ++N; }}
            for (size_t I = 1; I < N; ++I)
                 Arr[I].merge(Arr[I - 1]);
            swap(Arr[N - 1]); }}
  template<class Pr3>
       void sort(Pr3 Pr)
       {if (2 <= size())
            {const size_t MAXN = 25;
            Myt X(Mybase::Alval), Arr[MAXN + 1];
            size_t N = 0;
            while (!empty())
                 {X.splice(X.begin(), *this, begin());
                 size_t I;
                 for (I = 0; I < N && !Arr[I].empty(); ++I)
                      {Arr[I].merge(X, Pr);
                      Arr[I].swap(X); }
                 if (I == MAXN)
                      Arr[I - 1].merge(X, Pr);
                 else
                      {Arr[I].swap(X);
                      if (I == N)
                           ++N; }}
            for (size_t I = 1; I < N; ++I)
                 Arr[i].merge(Arr[I - 1], Pr);
            swap(Arr[N - 1]); }}
  void reverse()
       {if (2 <= size())
            {iterator L = end();
```

Figure 11.10:
`list`
Part 10

```
                for (iterator F = ++begin(); F != L; )
                     {iterator M = F;
                     Splice(begin(), *this, M, ++F, 1); }}}
protected:
     Nodeptr Buynode(Nodeptr Narg = 0, Nodeptr Parg = 0)
          {Nodeptr S = Alnod.allocate(1, (void *)0);
          Alptr.construct(&Acc::Next(S),
               Narg != 0 ? Narg : S);
          Alptr.construct(&Acc::Prev(S),
               Parg != 0 ? Parg : S);
          return (S); }
     void Freenode(Nodeptr S)
          {Alptr.destroy(&Acc::Next(S));
          Alptr.destroy(&Acc::Prev(S));
          Alnod.deallocate(S, 1); }
     void Splice(iterator P, Myt& X, iterator F, iterator L,
          size_type N)
          {if (Mybase::Alval == X.Alval)
               {if (this != &X)
                    {Incsize(N);
                    X.Size -= N; }
               Acc::Next(Acc::Prev(F.Mynode())) =
                    L.Mynode();
               Acc::Next(Acc::Prev(L.Mynode())) =
                    P.Mynode();
               Acc::Next(Acc::Prev(P.Mynode())) =
                    F.Mynode();
               Nodeptr S = Acc::Prev(P.Mynode());
               Acc::Prev(P.Mynode()) =
                    Acc::Prev(L.Mynode());
               Acc::Prev(L.Mynode()) =
                    Acc::Prev(F.Mynode());
               Acc::Prev(F.Mynode()) = S; }
          else
               {insert(P, F, L);
               X.erase(F, L); }}
     void Incsize(size_type N)
          {if (max_size() - size() < N)
               throw length_error("list<T> too long");
          Size += N; }
     Nodeptr Head;
     size_type Size;
     };

          // list TEMPLATE OPERATORS
template<class Ty, class A> inline
     void swap(list<Ty, A>& X, list<Ty, A>& Y)
     {X.swap(Y); }

template<class Ty, class A> inline
     bool operator==(const list<Ty, A>& X,
          const list<Ty, A>& Y)
     {return (X.size() == Y.size()
          && equal(X.begin(), X.end(), Y.begin())); }
```

Figure 11.11: `list` **Part 11**

```
template<class Ty, class A> inline
     bool operator!=(const list<Ty, A>& X,
          const list<Ty, A>& Y)
     {return (!(X == Y)); }
template<class Ty, class A> inline
     bool operator<(const list<Ty, A>& X,
          const list<Ty, A>& Y)
     {return (lexicographical_compare(X.begin(), X.end(),
          Y.begin(), Y.end())); }
template<class Ty, class A> inline
     bool operator>(const list<Ty, A>& X,
          const list<Ty, A>& Y)
     {return (Y < X); }
template<class Ty, class A> inline
     bool operator<=(const list<Ty, A>& X,
          const list<Ty, A>& Y)
     {return (!(Y < X)); }
template<class Ty, class A> inline
     bool operator>=(const list<Ty, A>& X,
          const list<Ty, A>& Y)
     {return (!(X < Y)); }
} /* namespace std */
#endif /* LIST_ */                                       □
```

But allocators cause problems. The first problem is that an object of type **`list<T, allocator<T> >`** is constructed with an allocator object that doesn't do the whole job. We're not interested in allocating objects of type **`T`**, which is all that an **`allocator<T>`** object knows how to allocate. (But it does know how to *construct* and *destroy* such an object, so it is still needed.) Instead, we want to allocate objects of type **`Node`**. That means we need an allocator object of type **`allocator<Node>`**. And we want to associate it, in some obvious way, with the **`allocator<T>`** object supplied to the **`list`** object when it is constructed. The allocator might be allocating objects from a private storage pool, for example, which we certainly want to use as intended.

`rebind` Two bits of trickery, supplied by all allocator types, give you the power you need. The first is member template class **`rebind`** — the bizarre formula **`A::rebind<Node>::other`** is a way of naming the type **`allocator<Node>`** when all you have is the synonym **`A`** for the type **`allocator<T>`**. Once you can name the kind of allocator object you want, you still have to construct one from the original allocator object. So all allocator types supply a template constructor. For the default template class **`allocator`**, this constructor looks like:

```
template<class U>
    allocator(const allocator<U>&);
```

You can thus construct an **`allocator<Node>`** object from an **`allocator<T>`** object. For a more complex allocator than template class **`allocator`**, the constructor must be smart enough to copy over any pointers to private storage, or what have you.

smart pointers Allocators cause yet another problem. They reserve the right to store the objects they allocate in funny places. More precisely, an allocator type **A** defines the type **A::pointer**, which you are obliged to use to describe any "pointer" to an allocated object. We use quotes here because the type need not be a pointer in the old-fashioned sense inherited from C. (See the discussion of smart pointers on page 99.) If **p** has type **A::pointer**, it promises that ***p** is an lvalue that designates the allocated object. (You can access the value or assign to it via ***p**.) But not much more.

This weaker promise causes a real problem with the declaration of class **Node**. You want to write:

```
class Node {
    A::rebind<Node>::pointer Next, Prev;
    T Value;
    };
```

but you can't. An allocator template can be specialized only for a complete type. Type **Node** is not complete until the closing brace of its definition. You need to describe the pointers it stores before you can complete its definition. What can you do?

void pointers When such dependency loops occur, the usual copout in C is to introduce generic, or "void," pointers, as in:

```
class Node {
    void *Next, *Prev;
    T Value;
    };
```

A void pointer is obliged to store any kind of object pointer you can declare in C. You lose a bit of type checking this way, and you have to write occasional type casts when you use the pointers, but it does solve the problem.

When it comes to pointers supplied by allocators, however, the C++ Standard is less than clear. It *suggests* that an **A::pointer** can be an arbitrary template class type, subject to the restrictions we sketched above. But it imposes no requirement that such a template class type define the equivalent of a void pointer. An implementation has to fill in the blanks.

This implementation assumes that any **A::pointer** is interconvertible with any **A::rebind<void>::pointer**. Put another way, the type **A::<void>::pointer** supplies the generic pointer type for the family of types A<T>::pointer, all of which have the same representation. Whoever writes the allocator template must supply an explicit specialization for type **void** anyway. It shouldn't be all that hard to ensure that the explicit specialization supplies a sufficiently flexible pointer type in the bargain.

null pointers This implementation also assumes that an integer zero still serves as a null pointer, no matter how exotic the pointers defined by the allocator. Specifically, you can assign zero to a generic pointer object; the resulting value will not compare equal to a generic pointer to any allocated object. And you can compare a generic pointer object to zero, using **operator==**; the result is true only if the generic pointer object stores a null pointer. (We quietly made this assumption in the previous chapter, with respect to the member object **`First`**.)

Smart pointers introduce one last wrinkle. This implementation assumes that they may have a nontrivial constructor and destructor, unlike the scalar pointers inherited from C. To play the game strictly by the rules, you need an allocator object to perform these tasks for you (though it's hard to imagine what special magic might be required). So a **`list`** object makes use of three allocator objects:

`Alnod`
- **`Alnod`** for the node type, to allocate and free nodes

`Alptr`
- **`Alptr`** for the node pointer type, to construct and destroy links stored in nodes

`Alty`
- **`Alty`** for the element type, to construct and destroy elements stored in nodes

In principle, it is necessary to store only one of these allocator objects. Either of the others can be generated on the fly as needed, as in:

```
A::rebind<Node>::other(Alty).destroy(p);
```

which generates a temporary allocator akin to **`Alnod`** long enough to destroy the node pointer **`p`**. Perhaps a compiler will know enough to optimize away the actual generation of a temporary, at least for default allocator objects. On the other hand, we know that the storage for a typical allocator object can be optimized away. (See the discussion of zero-size allocator objects on page 266.) So this implementation takes the safer bet that storing three allocator objects in a **`list`** object involves no real overheads in space or time.

After that long preamble, we can now study the code with a bit more wisdom. The file **`list`** reveals that a specializaton **`list<T, A>`** derives from a succession of three base classes:

`List_nod`
- **`List_nod<T, A>`** defines the generic pointer type **`Genptr`** and the node type **`Node`**. It also stores the allocator object **`Alnod`**.

`List_ptr`
- **`List_ptr<T, A>`** defines the node pointer type **`Nodeptr`**. It also stores the allocator object **`Alptr`**.

`List_val`
- **`List_val<T, A>`** stores the allocator object **`Alnod`**.

A smart enough compiler knows to allocate no storage within a **`list<T, A>`** object for any of these base objects.

Still more complexity is encapsulated in the member struct **`Acc`**. It supplies handy functions for accessing the objects stored in a list node. Thus, the expression **`Acc::Next(Ptr)`** lets you access the forward pointer **`Next`** in the node designated by **`Ptr`**. To make the expression an lvalue, the function **`Acc::Next`** must return a reference to the stored object. Opinions differ considerably on how much latitude an allocator has in defining reference types. Some feel that **`Alloc<T>::reference`** must always be a synonym for **`T&`**. But just in case someone supplies an allocator that succeeds in being more clever, these functions make uniform use of the reference types defined by the allocators.

`list` defines nontrivial member classes **`iterator`** and **`const_iterator`**. They are even simpler than template class **`Ptrit`**, which **`vector`** uses to define its iterators — **`list`** supports just bidirectional iterators, not random-access iterators as does **`vector`**. The iterators for both containers store only a single pointer.

`Buynode` **`Freenode`** **`Incsize`** A handful of protected member functions perform a number of common operations. The call **`Buynode(next, prev)`** allocates a node and initializes the two pointer member objects appropriately. The call **`Freenode(p)`** frees a node. Both assume that some other agency will construct and destroy the stored element value as needed. The call **`Incsize(n)`** increments the stored length of the controlled sequence. It checks for the unlikely event that the list has grown too large.

`Splice` The call **`Splice(p, cont2, first, last, n)`** splices the sequence of **`n`** nodes designated by **`[first, last)`**, in the list object **`cont2`**, just before **`p`**. It assumes that the caller has checked for any overlap that could cause problems. But it does check for an attempt to splice between two containers with incompatible allocators, in which case it copies (and erases) the nodes instead. The order in which links are altered here is very delicate, if various special cases are to work properly.

The member functions that **`list`** shares with other template containers introduce little new. We described most of the machinery in conjunction with template class **`vector`**. What you will find here is a greater effort to recover gracefully when programmer-supplied code throws an exception. Recall that **`list`** alone among the containers promises to roll back any operation interrupted by a thrown exception.

`splice` The three versions of **`splice`** defer the actual work to **`Splice`** and **`Incsize`**, after suitable checking. An attempt to splice a sequence of nodes to a point somewhere inside the sequence is a particular concern. The C++ Standard simply says this operation is undefined. Moreover, it requires a splice of a subrange from the same container to occur in constant time, independent of the number of elements in the subrange. This requirement leaves no room for the kind of checking required to avoid generating a knot in the list.

This implementation deviates from the C++ Standard by checking anyway. If the splice would generate a knot, the controlled sequence is left unaltered.

`sort` Both forms of **`sort`** work the same way. They perform a succession of merges to an array of temporary **`list`** objects. Each element of the array stores a list that can grow twice as long as the one that precedes it, before it is merged into the next larger list. The last element is a special case — it stores an arbitrarily long list. But the array size is (arbitrarily) set at 25 elements, plus the overflow list at the end. So **`sort`** can sort up to 32 million elements before it has to deviate from the simple doubling algorithm. The final step is to merge the temporary lists back into the (now empty) container.

Testing `<list>`

`tlist.c` Figures 11.12 through 11.14 shows the file **`tlist.c`**. It is one of three test programs that look very much alike. See the file **`tvector.c`**, beginning on page 285, and the file **`tdeque.c`**, beginning on page 350. To ease comparison of these three test programs, we have simply commented out any tests inappropriate for a given container, without removing the unused code.

The test program performs a simple test that each of the member functions and types is present and behaves as intended, for one specialization of template class **`list`**. If all goes well, the program prints:

```
SUCCESS testing <list>
```

and takes a normal exit.

Figure 11.12: `tlist.c` Part 1

```
// test <list>
#include <assert.h>
#include <iostream>
#include <functional>
#include <list>
using namespace std;

     // TEST <list>
int main()
     {typedef allocator<char> Myal;

     // TEST list
     typedef list<char, Myal> Mycont;
     char ch, carr[] = "abc";
     Mycont::allocator_type *p_alloc = (Myal *)0;
     Mycont::pointer p_ptr = (char *)0;
     Mycont::const_pointer p_cptr = (const char *)0;
     Mycont::reference p_ref = ch;
     Mycont::const_reference p_cref = (const char&)ch;
     Mycont::size_type *p_size = (size_t *)0;
     Mycont::difference_type *p_diff = (ptrdiff_t *)0;
     Mycont::value_type *p_val = (char *)0;

     Mycont v0;
     Myal al = v0.get_allocator();
     Mycont v0a(al);
     assert(v0.empty() && v0.size() == 0);
     assert(v0a.size() == 0 && v0a.get_allocator() == al);
     Mycont v1(5), v1a(6, 'x'), v1b(7, 'y', al);
     assert(v1.size() == 5 && v1.back() == '\0');
     assert(v1a.size() == 6 && v1a.back() == 'x');
     assert(v1b.size() == 7 && v1b.back() == 'y');
     Mycont v2(v1a);
     assert(v2.size() == 6 && v2.front() == 'x');
     Mycont v3(v1a.begin(), v1a.end());
     assert(v3.size() == 6 && v3.front() == 'x');
```

Figure 11.13: `tlist.c` **Part 2**

```
     const Mycont v4(v1a.begin(), v1a.end(), al);
     assert(v4.size() == 6 && v4.front() == 'x');
     v0 = v4;
     assert(v0.size() == 6 && v0.front() == 'x');
//   assert(v0[0] == 'x' && v0.at(5) == 'x');

//   v0.reserve(12);
//   assert(12 <= v0.capacity());
     v0.resize(8);
     assert(v0.size() == 8 && v0.back() == '\0');
     v0.resize(10, 'z');
     assert(v0.size() == 10 && v0.back() == 'z');
     assert(v0.size() <= v0.max_size());

     Mycont::iterator p_it(v0.begin());
     Mycont::const_iterator p_cit(v4.begin());
     Mycont::reverse_iterator p_rit(v0.rbegin());
     Mycont::const_reverse_iterator p_crit(v4.rbegin());
     assert(*p_it == 'x' && *--(p_it = v0.end()) == 'z');
     assert(*p_cit == 'x' && *--(p_cit = v4.end()) == 'x');
     assert(*p_rit == 'z'
          && *--(p_rit = v0.rend()) == 'x');
     assert(*p_crit == 'x'
          && *--(p_crit = v4.rend()) == 'x');

     assert(v0.front() == 'x' && v4.front() == 'x');
     v0.push_back('a');
     assert(v0.back() == 'a');
     v0.pop_back();
     assert(v0.back() == 'z' && v4.back() == 'x');

     v0.push_front('b');
     assert(v0.front() == 'b');
     v0.pop_front();
     assert(v0.front() == 'x');

     v0.assign(v4.begin(), v4.end());
     assert(v0.size() == v4.size()
          && v0.front() == v4.front());
     v0.assign(4, 'w');
     assert(v0.size() == 4 && v0.front() == 'w');
     assert(*v0.insert(v0.begin(), 'a') == 'a');
     assert(v0.front() == 'a'
          && *++v0.begin() == 'w');
     v0.insert(v0.begin(), 2, 'b');
     assert(v0.front() == 'b'
          && *++v0.begin() == 'b'
          && *++ ++v0.begin() == 'a');
     v0.insert(v0.end(), v4.begin(), v4.end());
     assert(v0.back() == v4.back());
     v0.insert(v0.end(), carr, carr + 3);
     assert(v0.back() == 'c');
     v0.erase(v0.begin());
     assert(v0.front() == 'b' && *++v0.begin() == 'a');
     v0.erase(v0.begin(), ++v0.begin());
```

Figure 11.14: `tlist.c` **Part 3**

```
    assert(v0.front() == 'a');

    v0.clear();
    assert(v0.empty());
    v0.swap(v1);
    assert(!v0.empty() && v1.empty());
    swap(v0, v1);
    assert(v0.empty() && !v1.empty());
    assert(v1 == v1 && v0 < v1);
    assert(v0 != v1 && v1 > v0);
    assert(v0 <= v1 && v1 >= v0);

    v0.insert(v0.begin(), carr, carr + 3);
    v1.splice(v1.begin(), v0);
    assert(v0.empty() && v1.front() == 'a');
    v0.splice(v0.end(), v1, v1.begin());
    assert(v0.size() == 1 && v0.front() == 'a');
    v0.splice(v0.begin(), v1, v1.begin(), v1.end());
    assert(v0.front() == 'b' && v1.empty());
    v0.remove('b');
    assert(v0.front() == 'c');
    v0.remove_if(binder2nd<not_equal_to<char> >(
         not_equal_to<char>(), 'c'));
    assert(v0.front() == 'c' && v0.size() == 1);

    v0.merge(v1, greater<char>());
    assert(v0.front() == 'c' && v0.size() == 1);
    v0.insert(v0.begin(), carr, carr + 3);
    v0.unique();
    assert(v0.back() == 'c'&& v0.size() == 3);
    v0.unique(not_equal_to<char>());
    assert(v0.front() == 'a' && v0.size() == 1);
    v1.insert(v1.begin(), carr, carr + 3);
    v0.merge(v1);
    assert(v0.back() == 'c' && v0.size() == 4);
    v0.sort(greater<char>());
    assert(v0.back() == 'a' && v0.size() == 4);
    v0.sort();
    assert(v0.back() == 'c' && v0.size() == 4);
    v0.reverse();
    assert(v0.back() == 'a' && v0.size() == 4);

    cout << "SUCCESS testing <list>" << endl;
    return (0); }
```
□

Exercises

Exercise 11.1 This implementation of template class **`list`** never copies elements between nodes. Elements are constructed and destroyed, but never assigned. Under what circumstances is this behavior most desirable?

Exercise 11.2 Why must an allocator be specialized only for a complete type?

Exercise 11.3 Rewrite template class `list` to eliminate the use of a head node. Under what circumstances is this rewrite a better design?

Exercise 11.4 Write the template class `forward_list`, which stores only a single forward pointer in each node. What operations become more difficult (have less desirable time complexity) compared to template class `list`? What are the relative sizes of nodes for the two containers?

Exercise 11.5 Alter the definition of list iterators so that it is easy to determine if two iterators designate elements in different lists.

Exercise 11.6 One way to implement a *hash table* is as a vector of lists. A *hash function* maps a key value to an index into the vector. All elements of a given list share the same *hash value* even if their keys differ. (With a good hash function, a typical hash table has lists that are uniformly short, so lookup time for a given key is essentially constant.) Write the template class `hash_list` that implements a simple hash table.

Exercise 11.7 Alter the implementation of `hash_list` from the previous list to use a single list and a vector of list iterators. What are the advantages and disadvantages of the two versions?

Exercise 11.8 [**Harder**] How can you implement a bidirectional linked list storing only one pointer object per node?

Exercise 11.9 [**Very hard**] How can you eliminate the need for generic pointers in defining a list node?