
7.5. Lists

A list (an instance of the container class `list<>`) manages its elements as a doubly linked list (Figure 7.5). As usual, the C++ standard library does not specify the kind of the implementation, but it follows from the list's name, constraints, and specifications.

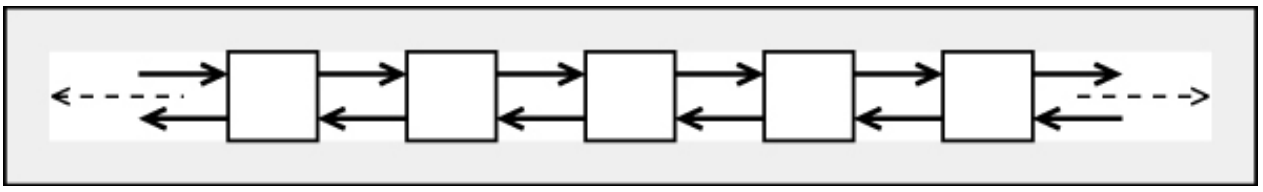

Figure 7.5. Structure of a List

To use a list, you must include the header file `<list>`:

```
#include <list>
```

There, the type is defined as a class template inside namespace `std`:

```
namespace std {
    template <typename T,
              typename Allocator = allocator<T> >
    class  list;
}
```

The elements of a list may have any type `T`. The optional second template parameter defines the memory model (see Chapter 19). The default memory model is the model `allocator`, which is provided by the C++ standard library.

7.5.1. Abilities of Lists

The internal structure of a list is totally different from that of an `array`, a vector, or a deque. The list object itself provides two pointers, the so-called *anchors*, which refer to the first and last elements. Each element has pointers to the previous and next elements (or back to the anchor). To insert a new element, you just manipulate the corresponding pointers (see Figure 7.6).

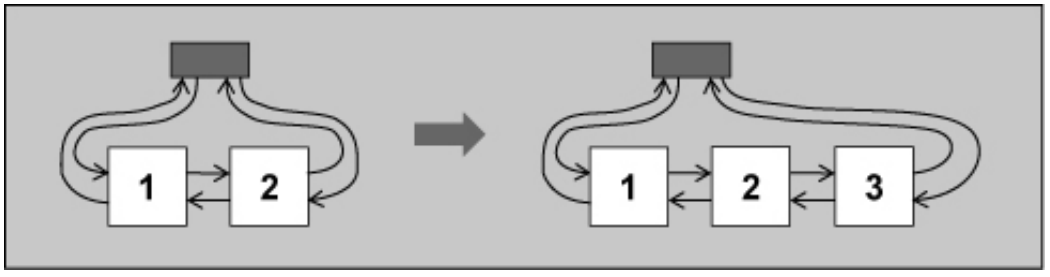

Figure 7.6. Internal Structure of a List when Appending a Value

Thus, a list differs in several major ways from `array`s, vectors, and deques:

- A list does not provide random access. For example, to access the fifth element, you must navigate the first four elements, following the chain of links. Thus, accessing an arbitrary element using a list is slow. However, you can navigate through the list from both ends. So accessing both the first and the last elements is fast.
- Inserting and removing elements is fast at each position (provided you are there), and not only at one or both ends. You can always insert and delete an element in constant time, because no other elements have to be moved. Internally, only some pointer values are manipulated.
- Inserting and deleting elements does not invalidate pointers, references, and iterators to other elements.
- A list supports exception handling in such a way that almost every operation succeeds or is a no-op. Thus, you can't get into an intermediate state in which only half of the operation is complete.

The member functions provided for lists reflect these differences from `array`s, vectors, and deques as follows:

- Lists provide `front()`, `push_front()`, and `pop_front()`, as well as `back()`, `push_back()`, and `pop_back()`.
- Lists provide neither a subscript operator nor `at()`, because no random access is provided.
- Lists don't provide operations for capacity or reallocation, because neither is needed. Each element has its own memory that stays

valid until the element is deleted.

- Lists provide many special member functions for moving and removing elements. These member functions are faster versions of general algorithms that have the same names. They are faster because they only redirect pointers rather than copy and move the values.

7.5.2. List Operations

Create, Copy, and Destroy

The ability to create, copy, and destroy lists is the same as it is for every sequence container. See Table 7.19 for the list operations that do this. See also Section 7.1.2, page 254, for some remarks about possible initialization sources.

Table 7.19. Constructors and Destructor of Lists

Operation	Effect
`list<Elem> c`	Default constructor; creates an empty list without any elements
`list<Elem> c(c2)`	Copy constructor; creates a new list as a copy of *c2* (all elements are copied)
`list<Elem> c = c2`	Copy constructor; creates a new list as a copy of *c2* (all elements are copied)
`list<Elem> c(rv)`	Move constructor; creates a new list, taking the contents of the rvalue *rv* (since C++11)
`list<Elem> c = rv`	Move constructor; creates a new list, taking the contents of the rvalue *rv* (since C++11)
`list<Elem> c(n)`	Creates a list with *n* elements created by the default constructor
`list<Elem> c(n,elem)`	Creates a list initialized with *n* copies of element *elem*
`list<Elem> c(beg,end)`	Creates a list initialized with the elements of the range *[beg,end)*
`list<Elem> c(initlist)`	Creates a list initialized with the elements of initializer list *initlist* (since C++11)
`list<Elem> c = initlist`	Creates a list initialized with the elements of initializer list *initlist* (since C++11)
`c.~list()`	Destroys all elements and frees the memory

Nonmodifying Operations

Lists provide the usual operations for size and comparisons. See Table 7.20 for a list and Section 7.1.2, page 254, for details.

Table 7.20. Nonmodifying Operations of Lists

Operation	Effect
`c.empty()`	Returns whether the container is empty (equivalent to `size()==0` but might be faster)
`c.size()`	Returns the current number of elements
`c.max_size()`	Returns the maximum number of elements possible
`c1 == c2`	Returns whether *c1* is equal to *c2* (calls == for the elements)
`c1 != c2`	Returns whether *c1* is not equal to *c2* (equivalent to `!(c1==c2)`)
`c1 < c2`	Returns whether *c1* is less than *c2*
`c1 > c2`	Returns whether *c1* is greater than *c2* (equivalent to `c2<c1`)
`c1 <= c2`	Returns whether *c1* is less than or equal to *c2* (equivalent to `!(c2<c1)`)
`c1 >= c2`	Returns whether *c1* is greater than or equal to *c2* (equivalent to `!(c1<c2)`)

Assignments

Lists also provide the usual assignment operations for sequence containers (Table 7.21). As usual, the insert operations match the constructors to provide different sources for initialization (see Section 7.1.2, page 254, for details).

Table 7.21. Assignment Operations of Lists

Operation	Effect
`c = c2`	Assigns all elements of *c2* to `c`
`c = rv`	Move assigns all elements of the rvalue *rv* to `c` (since C++11)
`c = initlist`	Assigns all elements of the initializer list *initlist* to `c` (since C++11)
`c.assign(n,elem)`	Assigns *n* copies of element *elem*
`c.assign(beg,end)`	Assigns the elements of the range [*beg*,*end*)
`c.assign(initlist)`	Assigns all the elements of the initializer list *initlist*
`c1.swap(c2)`	Swaps the data of *c1* and *c2*
`swap(c1,c2)`	Swaps the data of *c1* and *c2*

Element Access

To access all elements of a list, you must use range-based `for` loops (see Section 3.1.4, page 17), specific operations, or iterators. Because it does not have random access, a list provides only `front()` and `back()` for accessing elements directly (Table 7.22).

Table 7.22. Direct Element Access of Lists

Operation	Effect
`c.front()`	Returns the first element (*no* check whether a first element exists)
`c.back()`	Returns the last element (*no* check whether a last element exists)

As usual, these operations do *not* check whether the container is empty. If the container is empty, calling these operations results in undefined behavior. Thus, the caller must ensure that the container contains at least one element. For example:

Click here to view code image

```
std::list<Elem> coll;              // empty!

std::cout << coll.front();         // RUNTIME ERROR      undefined behavior

if (!coll.empty()) {
    std::cout << coll.back();      // OK
}
```

Note that this code is OK only in single-threaded environments. In multithreaded contexts, you need synchronization mechanisms to ensure that `coll` is not modified between the check for its size and the access to the element (see Section 18.4.3, page 984, for details).

Iterator Functions

To access all elements of a list, you must use iterators. Lists provide the usual iterator functions (Table 7.23). However, because a list has no random access, these iterators are only bidirectional. Thus, you can't call algorithms that require random-access iterators. All algorithms that manipulate the order of elements a lot, especially sorting algorithms, are in this category. However, for sorting the elements, lists provide the special member function `sort()` (see Section 8.8.1, page 422).

Table 7.23. Iterator Operations of Lists

Operation	Effect
`c.begin()`	Returns a bidirectional iterator for the first element
`c.end()`	Returns a bidirectional iterator for the position after the last element
`c.cbegin()`	Returns a constant bidirectional iterator for the first element (since C++11)
`c.cend()`	Returns a constant bidirectional iterator for the position after the last element (since C++11)
`c.rbegin()`	Returns a reverse iterator for the first element of a reverse iteration
`c.rend()`	Returns a reverse iterator for the position after the last element of a reverse iteration
`c.crbegin()`	Returns a constant reverse iterator for the first element of a reverse iteration (since C++11)
`c.crend()`	Returns a constant reverse iterator for the position after the last element of a reverse iteration (since C++11)

Inserting and Removing Elements

Table 7.24 shows the operations provided for lists to insert and to remove elements. Lists provide all functions of deques, supplemented by

special implementations of the `remove()` and `remove_if()` algorithms.

Table 7.24. Insert and Remove Operations of Lists

Operation	Effect
`c.push_back(`*elem*`)`	Appends a copy of *elem* at the end
`c.pop_back()`	Removes the last element (does not return it)
`c.push_front(`*elem*`)`	Inserts a copy of *elem* at the beginning
`c.pop_front()`	Removes the first element (does not return it)
`c.insert(`*pos*`,`*elem*`)`	Inserts a copy of *elem* before iterator position *pos* and returns the position of the new element
`c.insert(`*pos*`,`*n*`,`*elem*`)`	Inserts *n* copies of *elem* before iterator position *pos* and returns the position of the first new element (or *pos* if there is no new element)
`c.insert(`*pos*`,`*beg*`,`*end*`)`	Inserts a copy of all elements of the range [*beg*,*end*) before

As usual when using the STL, you must ensure that the arguments are valid. Iterators must refer to valid positions, and the beginning of a range must have a position that is not behind the end.

Inserting and removing is faster if, when working with multiple elements, you use a single call for all elements rather than multiple calls.

For removing elements, lists provide special implementations of the `remove()` algorithms (see Section 11.7.1, page 575). These member functions are faster than the `remove()` algorithms because they manipulate only internal pointers rather than the elements. So, unlike with vectors or deques, you should call `remove()` as a member function and not as an algorithm (see Section 7.3.2, page 276, for details). To remove all elements that have a certain value, you can do the following (see Section 6.7.3, page 223, for further details):

```
std::list<Elem> coll;
...
// remove all elements with value val
coll.remove(val);
```

However, to remove only the first occurrence of a value, you must use an algorithm such as that mentioned for vectors in Section 7.3.2, page 277.

You can use `remove_if()` to define the criterion for the removal of the elements by a function or a function object. `remove_if()` removes each element for which calling the passed operation yields `true`. An example of the use of `remove_if()` is a statement to remove all elements that have an even value:

```
// remove all even elements
coll.remove_if ([] (int i) {
                    return i % 2 == 0;
                });
```

Here, a lambda is used to find out which elements to remove. Because the lambda returns whether a passed element is even, the statement as a whole removes all even elements. See Section 11.7.1, page 575, for additional examples of `remove()` and `remove_if()` .

The following operations do not invalidate iterators and references to other elements: `insert()` , `emplace()` , `emplace ... ()` , `push_front()` , `push_back()` , `pop_front()` , `pop_back()` , and `erase()` .

Splice Functions and Functions to Change the Order of Elements

Linked lists have the advantage that you can remove and insert elements at any position in constant time. If you move elements from one container to another, this advantage doubles in that you need only redirect some internal pointers (Figure 7.7).

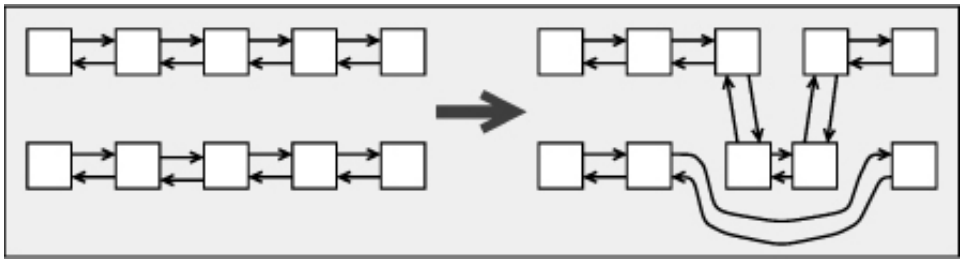

Figure 7.7. Splice Operations to Change the Order of List Elements

To support this ability, lists provide not only `remove()` but also additional modifying member functions to change the order of and relink elements and ranges. You can call these operations to move elements inside a single list or between two lists, provided that the lists have the same type. Table 7.25 lists these functions. They are covered in detail in Section 8.8, page 420, with examples on page 298.

Table 7.25. Special Modifying Operations for Lists

Operation	Effect
`c.unique()`	Removes duplicates of consecutive elements with the same value
`c.unique(`*op*`)`	Removes duplicates of consecutive elements, for which *op*`()` yields `true`
`c.splice(`*pos*`,`*c2*`)`	Moves all elements of *c2* to `c` in front of the iterator position *pos*
`c.splice(`*pos*`,`*c2*`,`*c2pos*`)`	Moves the element at *c2pos* in *c2* in front of *pos* of list `c` (`c` and *c2* may be identical)
`c.splice(`*pos*`,`*c2*	Moves all elements of the range [*c2beg*,*c2end*) in *c2* in

7.5.3. Exception Handling

Lists have the best support of exception safety of the standard containers in the STL. Almost all list operations will either succeed or have no effect. The only operations that don't give this guarantee in the face of exceptions are assignment operations and the member function `sort()` (they give the usual "basic guarantee" that they will not leak resources or violate container invariants in the face of

exceptions). `merge()`, `remove()`, `remove_if()`, and `unique()` give guarantees under the condition that comparing the elements (using operator `==` or the predicate) doesn't throw. Thus, to use a term from database programming, you could say that lists are *transaction safe*, provided that you don't call assignment operations or `sort()` and that you ensure that comparing elements doesn't throw. Table 7.26 lists all operations that give special guarantees in the face of exceptions. See Section 6.12.2, page 248, for a general discussion of exception handling in the STL.

Table 7.26. List Operations with Special Guarantees in Face of Exceptions

Operation	Guarantee
`push_back()`	Either succeeds or has no effect
`push_front()`	Either succeeds or has no effect

7.5.4. Examples of Using Lists

The following example in particular shows the use of the special member functions for lists:

Click here to view code image

```
// cont/list1.cpp

#include <list>
#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;

void printLists (const list<int>& l1, const list<int>& l2)
{
    cout << "list1: ";
    copy (l1.cbegin(), l1.cend(), ostream_iterator<int>(cout," "));
    cout << endl << "list2: ";
    copy (l2.cbegin(), l2.cend(), ostream_iterator<int>(cout," "));
    cout << endl << endl;
}

int main()
{
    // create two empty lists
    list<int> list1, list2;

    // fill both lists with elements
    for (int i=0; i<6; ++i) {
        list1.push_back(i);
        list2.push_front(i);
    }
    printLists(list1, list2);

    // insert all elements of list1 before the first element with value 3 of list2
    // - find() returns an iterator to the first element with value 3
    list2.splice(find(list2.begin(),list2.end(),     // destination position
                      3),
                 list1);                             // source list
    printLists(list1, list2);

    // move first element of list2 to the end
```

```
        list2.splice(list2.end(),       // destination position
                     list2,             // source list
                     list2.begin());    // source position
        printLists(list1, list2);

        // sort second list, assign to list1 and remove duplicates
        list2.sort();
        list1 = list2;
        list2.unique();
        printLists(list1, list2);

        // merge both sorted lists into the first list
        list1.merge(list2);
        printLists(list1, list2);
    }
```

The program has the following output:

```
list1: 0 1 2 3 4 5
list2: 5 4 3 2 1 0

list1:
list2: 5 4 0 1 2 3 4 5 3 2 1 0

list1:
list2: 4 0 1 2 3 4 5 3 2 1 0 5

list1: 0 0 1 1 2 2 3 3 4 4 5 5
list2: 0 1 2 3 4 5

list1: 0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
list2:
```

See Section 7.6.4, page 312, for a corresponding example using a forward list.