**Lists**

A list manages its elements as a doubly linked list.

Header file **#include <list>**

Type is defined as a class template inside namespace std:

namespace std {

template <typename T,

typename Allocator = allocator<T> >

class list;

}

The optional second template parameter defines the memory model.

# Abilities of Lists

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.

A list differs in several major ways from arrays, vectors, and deques:

1. A list does not provide random access.
2. Inserting and removing elements is fast at each position.
3. Inserting and deleting elements does not invalidate pointers, references, and iterators to other elements.
4. 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.

Member functions provided for lists reflect these differences from arrays, vectors, and deques as follows:

1. Lists provide front(), push\_front(), and pop\_front(), as well as back(), push\_back(), and pop\_back().
2. Lists provide neither a subscript operator nor at(), because no random access is provided.
3. Lists don’t provide operations for capacity or reallocation, because neither is needed.
4. 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.

# List Operations

## Create, Copy and Destroy

The ability to create, copy, and destroy lists is the same as it is for every sequence container.

|  |  |
| --- | --- |
| 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.

|  |  |
| --- | --- |
| 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. As usual, the insert operations match the constructors to provide different sources for initialization.

|  |  |
| --- | --- |
| 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 17), specific operations, or iterators. Because it does not have random access, a list provides only front() and back() for accessing elements directly.

|  |  |
| --- | --- |
| 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) |

These operations do not check whether the container is empty. If the container is empty, calling these operations results in undefined behavior.

## Iterator Functions

To access all elements of a list, you must use iterators. However, because a list has no random access, these iterators are only bidirectional. Thus, you can’t call algorithms that require random-access iterators.

|  |  |
| --- | --- |
| 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

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, member functions are faster than the remove() algorithms because they manipulate only internal pointers rather than the elements.

|  |  |
| --- | --- |
| 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 iterator position pos and returns the position of the first new element (or pos if there is no new element) |
| c.insert(pos,initlist) | Inserts a copy of all elements of the initializer list initlist before iterator position pos and returns the position of the first new element (or pos if there is no new element; since C++11) |
| c.emplace(pos,args...) | Inserts a copy of an element initialized with args before iterator position pos and returns the position of the new element (since C++11) |
| c.emplace\_back(args...) | Appends a copy of an element initialized with args at the end (returns nothing; since C++11) |
| c.emplace\_front(args...) | Inserts a copy of an element initialized with args at the beginning (returns nothing; since C++11) |
| c.erase(pos) | Removes the element at iterator position pos and returns the position of the next element |
| c.erase(beg,end) | Removes all elements of the range [beg,end) and returns the position of the next element |
| c.remove(val) | Removes all elements with value val |
| c.remove\_if(op) | Removes all elements for which op(elem) yields true |
| c.resize(num) | Changes the number of elements to num (if size() grows new elements are created by their default constructor) |
| c.resize(num,elem) | Changes the number of elements to num (if size() grows new elements are copies of elem) |
| c.clear() | Removes all elements (empties the container) |

## 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.

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.

|  |  |
| --- | --- |
| 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,c2beg,c2end) | Moves all elements of the range [c2beg,c2end) in c2 in front of pos of list c (c and c2 may be identical) |
| c.sort() | Sorts all elements with operator < |
| c.sort(op) | Sorts all elements with op() |
| c.merge(c2) | Assuming that both containers contain the elements sorted, moves all elements of c2 into c so that all elements are merged and still sorted |
| c.merge(c2,op) | Assuming that both containers contain the elements sorted due to the sorting criterion op(), moves all elements of c2 into c so that all elements are merged and still sorted according to op() |
| c.reverse() | Reverses the order of all elements |

# Exception Handling

Almost all list operations will either succeed or have no effect.

The only operations that don’t give this guarantee 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.

|  |  |
| --- | --- |
| Operation | Guarantee |
| push\_back() | Either succeeds or has no effect |
| push\_front() | Either succeeds or has no effect |
| insert() | Either succeeds or has no effect |
| pop\_back() | Doesn’t throw |
| pop\_front() | Doesn’t throw |
| erase() | Doesn’t throw |
| clear() | Doesn’t throw |
| resize() | Either succeeds or has no effect |
| remove() | Doesn’t throw if comparing the elements doesn’t throw |
| remove\_if() | Doesn’t throw if the predicate doesn’t throw |
| unique() | Doesn’t throw if comparing the elements doesn’t throw |
| splice() | Doesn’t throw |
| merge() | Either succeeds or has no effect if comparing the elements doesn’t throw |
| reverse() | Doesn’t throw |
| swap() | Doesn’t throw |

# END