list<T, Alloc>

Containers

Type

Category: containers

Component type: type

Description

A `list` is a doubly linked list. That is, it is a Sequence that supports both forward and backward traversal, and (amortized) constant time insertion and removal of elements at the beginning or the end, or in the middle. `Lists` have the important property that insertion and splicing do not invalidate iterators to list elements, and that even removal invalidates only the iterators that point to the elements that are removed. The ordering of iterators may be changed (that is, `list<T>::iterator` might have a different predecessor or successor after a list operation than it did before), but the iterators themselves will not be invalidated or made to point to different elements unless that invalidation or mutation is explicit. [1]

Note that singly linked lists, which only support forward traversal, are also sometimes useful. If you do not need backward traversal, then `slist` may be more efficient than `list`.

Definition

Defined in the standard header list, and in the nonstandard backward-compatibility header list.h.

Example

```
list<int> L;
L.push_back(0);
L.push_front(1);
L.insert(++L.begin(), 2);
copy(L.begin(), L.end(), ostream_iterator<int>(cout, " "));
// The values that are printed are 1 2 0
```

Template parameters

Parameter	Description	Default
`T`	The `list`'s value type: the type of object that is stored in the list.	
`Alloc`	The `list`'s allocator, used for all internal memory management.	`alloc`

Model of

Reversible Container, Front Insertion Sequence, Back Insertion Sequence.

Type requirements

None, except for those imposed by the requirements of Reversible Container, Front Insertion Sequence, and Back Insertion Sequence.

Public base classes

None.

Members

Member	Where defined	Description
`value_type`	Container	The type of object, `T`, stored in the list.
`pointer`	Container	Pointer to `T`.
`reference`	Container	Reference to `T`
`const_reference`	Container	Const reference to `T`
`size_type`	Container	An unsigned integral type.
`difference_type`	Container	A signed integral type.
`iterator`	Container	Iterator used to iterate through a `list`.
`const_iterator`	Container	Const iterator used to iterate through a `list`.
`reverse_iterator`	Reversible Container	Iterator used to iterate backwards through a `list`.
`const_reverse_iterator`	Reversible Container	Const iterator used to iterate backwards through a `list`.
`iterator begin()`	Container	Returns an `iterator` pointing to the beginning of the `list`.
`iterator end()`	Container	Returns an `iterator` pointing to the end of the `list`.
`const_iterator begin() const`	Container	Returns a `const_iterator` pointing to the beginning of the `list`.
`const_iterator end() const`	Container	Returns a `const_iterator` pointing to the end of the `list`.
`reverse_iterator rbegin()`	Reversible Container	Returns a `reverse_iterator` pointing to the beginning of the reversed list.
`reverse_iterator rend()`	Reversible Container	Returns a `reverse_iterator` pointing to the end of the reversed list.
`const_reverse_iterator rbegin() const`	Reversible Container	Returns a `const_reverse_iterator` pointing to the beginning of the reversed list.
`const_reverse_iterator rend() const`	Reversible Container	Returns a `const_reverse_iterator` pointing to the end of the reversed list.

`size_type size() const`	Container	Returns the size of the `list`. Note: you should not assume that this function is constant time. It is permitted to be *O(N)*, where *N* is the number of elements in the `list`. If you wish to test whether a `list` is empty, you should write `L.empty()` rather than `L.size() == 0`.
`size_type max_size() const`	Container	Returns the largest possible size of the `list`.
`bool empty() const`	Container	`true` if the `list`'s size is `0`.
`list()`	Container	Creates an empty list.
`list(size_type n)`	Sequence	Creates a list with `n` elements, each of which is a copy of `T()`.
`list(size_type n, const T& t)`	Sequence	Creates a list with `n` copies of `t`.
`list(const list&)`	Container	The copy constructor.
`template <class InputIterator> list(InputIterator f, InputIterator l)` [2]	Sequence	Creates a list with a copy of a range.
`~list()`	Container	The destructor.
`list& operator=(const list&)`	Container	The assignment operator
`reference front()`	Front Insertion Sequence	Returns the first element.
`const_reference front() const`	Front Insertion Sequence	Returns the first element.
`reference back()`	Sequence	Returns the last element.
`const_reference back() const`	Back Insertion Sequence	Returns the last element.
`void push_front(const T&)`	Front Insertion Sequence	Inserts a new element at the beginning.
`void push_back(const T&)`	Back Insertion Sequence	Inserts a new element at the end.
`void pop_front()`	Front Insertion Sequence	Removes the first element.
`void pop_back()`	Back Insertion Sequence	Removes the last element.
`void swap(list&)`	Container	Swaps the contents of two lists.
`iterator insert(iterator pos, const T& x)`	Sequence	Inserts `x` before `pos`.
`template <class InputIterator> void insert(iterator pos,`	Sequence	Inserts the range `[f, l)` before `pos`.

`InputIterator f, InputIterator l)` [2]		
`void insert(iterator pos, size_type n, const T& x)`	Sequence	Inserts `n` copies of `x` before `pos`.
`iterator erase(iterator pos)`	Sequence	Erases the element at position `pos`.
`iterator erase(iterator first, iterator last)`	Sequence	Erases the range `[first, last)`
`void clear()`	Sequence	Erases all of the elements.
`void resize(n, t = T())`	Sequence	Inserts or erases elements at the end such that the size becomes `n`.
`void splice(iterator pos, list& L)`	`list`	See below.
`void splice(iterator pos, list& L, iterator i)`	`list`	See below.
`void splice(iterator pos, list& L, iterator f, iterator l)`	`list`	See below.
`void remove(const T& value)`	`list`	See below.
`void unique()`	`list`	See below.
`void merge(list& L)`	`list`	See below.
`void sort()`	`list`	See below.
`bool operator==(const list&, const list&)`	Forward Container	Tests two lists for equality. This is a global function, not a member function.
`bool operator<(const list&, const list&)`	Forward Container	Lexicographical comparison. This is a global function, not a member function.

New members

These members are not defined in the Reversible Container, Front Insertion Sequence, and Back Insertion Sequence requirements, but are specific to `list`.

Function	Description
`void splice(iterator position, list<T, Alloc>& x);`	`position` must be a valid iterator in `*this`, and `x` must be a list that is distinct from `*this`. (That is, it is required that `&x != this`.) All of the elements of `x` are inserted before `position` and removed from `x`. All iterators remain valid, including iterators that point to elements of `x`. [3] This function is constant time.
`void splice(iterator position, list<T, Alloc>& x, iterator i);`	`position` must be a valid iterator in `*this`, and `i` must be a dereferenceable iterator in `x`. `Splice` moves the element pointed to by `i` from `x` to `*this`, inserting it before `position`. All iterators remain valid, including iterators that point to elements of `x`. [3] If `position == i` or `position == ++i`, this function is a null operation. This function is constant time.
`void splice(iterator position,`	`position` must be a valid iterator in `*this`, and `[first, last)`

`list<T, Alloc>& x, iterator f, iterator l);`	must be a valid range in `x`. `position` may not be an iterator in the range `[first, last)`. `Splice` moves the elements in `[first, last)` from `x` to `*this`, inserting them before `position`. All iterators remain valid, including iterators that point to elements of `x`. [3] This function is constant time.
`void remove(const T& val);`	Removes all elements that compare equal to `val`. The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid. This function is linear time: it performs exactly `size()` comparisons for equality.
`template<class Predicate> void remove_if(Predicate p);` [4]	Removes all elements `*i` such that `p(*i)` is true. The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid. This function is linear time: it performs exactly `size()` applications of `p`.
`void unique();`	Removes all but the first element in every consecutive group of equal elements. The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid. This function is linear time: it performs exactly `size() - 1` comparisons for equality.
`template<class BinaryPredicate> void unique(BinaryPredicate p);` [4]	Removes all but the first element in every consecutive group of equivalent elements, where two elements `*i` and `*j` are considered equivalent if `p(*i, *j)` is true. The relative order of elements that are not removed is unchanged, and iterators to elements that are not removed remain valid. This function is linear time: it performs exactly `size() - 1` comparisons for equality.
`void merge(list<T, Alloc>& x);`	Both `*this` and `x` must be sorted according to `operator<`, and they must be distinct. (That is, it is required that `&x != this`.) This function removes all of `x`'s elements and inserts them in order into `*this`. The merge is stable; that is, if an element from `*this` is equivalent to one from `x`, then the element from `*this` will precede the one from `x`. All iterators to elements in `*this` and `x` remain valid. This function is linear time: it performs at most `size() + x.size() - 1` comparisons.
`template<class BinaryPredicate> void merge(list<T, Alloc>& x, BinaryPredicate Comp);` [4]	`Comp` must be a comparison function that induces a strict weak ordering (as defined in the LessThan Comparable requirements) on objects of type `T`, and both `*this` and `x` must be sorted according to that ordering. The lists `x` and `*this` must be distinct. (That is, it is required that `&x != this`.) This function removes all of `x`'s elements and inserts them in order into `*this`. The merge is stable; that is, if an element from `*this` is equivalent to one from `x`, then the element from `*this` will precede the one from `x`. All iterators to elements in `*this` and `x` remain valid. This function is linear time: it performs at most `size() + x.size() - 1` applications of `Comp`.
`void reverse();`	Reverses the order of elements in the list. All iterators remain

	valid and continue to point to the same elements. [5] This function is linear time.
`void sort();`	Sorts `*this` according to `operator<`. The sort is stable, that is, the relative order of equivalent elements is preserved. All iterators remain valid and continue to point to the same elements. [6] The number of comparisons is approximately `N log N`, where `N` is the `list`'s size.

`template<class BinaryPredicate>`
`void sort(BinaryPredicate comp);`
[4]
	`Comp` must be a comparison function that induces a strict weak ordering (as defined in the LessThan Comparable requirements on objects of type `T`. This function sorts the list `*this` according to `Comp`. The sort is stable, that is, the relative order of equivalent elements is preserved. All iterators remain valid and continue to point to the same elements. [6] The number of comparisons is approximately `N log N`, where `N` is the `list`'s size.

Notes

[1] A comparison with `vector` is instructive. Suppose that `i` is a valid `vector<T>::iterator`. If an element is inserted or removed in a position that precedes `i`, then this operation will either result in `i` pointing to a different element than it did before, or else it will invalidate `i` entirely. (A `vector<T>::iterator` will be invalidated, for example, if an insertion requires a reallocation.) However, suppose that `i` and `j` are both iterators into a vector, and there exists some integer `n` such that `i == j + n`. In that case, even if elements are inserted into the vector and `i` and `j` point to different elements, the relation between the two iterators will still hold. A `list` is exactly the opposite: iterators will not be invalidated, and will not be made to point to different elements, but, for `list` iterators, the predecessor/successor relationship is not invariant.

[2] This member function relies on *member template* functions, which at present (early 1998) are not supported by all compilers. If your compiler supports member templates, you can call this function with any type of input iterator. If your compiler does not yet support member templates, though, then the arguments must either be of type `const value_type*` or of type `list::const_iterator`.

[3] A similar property holds for all versions of `insert()` and `erase()`. `List<T, Alloc>::insert()` never invalidates any iterators, and `list<T, Alloc>::erase()` only invalidates iterators pointing to the elements that are actually being erased.

[4] This member function relies on *member template* functions, which at present (early 1998) are not supported by all compilers. You can only use this member function if your compiler supports member templates.

[5] If `L` is a list, note that `L.reverse()` and `reverse(L.begin(), L.end())` are both correct ways of reversing the list. They differ in that `L.reverse()` will preserve the value that each iterator into `L` points to but will not preserve the iterators' predecessor/successor relationships, while `reverse(L.begin(), L.end())` will not preserve the value that each iterator points to but will preserve the iterators' predecessor/successor relationships. Note also that the algorithm `reverse(L.begin(), L.end())` will use `T`'s assignment operator, while the member function `L.reverse()` will not.

[6] The `sort` algorithm works only for random access iterators. In principle, however, it would be possible to write a sort algorithm that also accepted bidirectional iterators. Even if there were such a version of `sort`, it would still be useful for `list` to have a `sort` member function. That is, `sort` is provided

as a member function not only for the sake of efficiency, but also because of the property that it preserves the values that list iterators point to.

See also

Bidirectional Iterator, Reversible Container, Sequence, `slist` `vector`.
