**List**

template < class T, class Alloc = allocator<T> > class list;

defines the list container class

# List class

1. sequence containers that allow constant time insert and erase operations anywhere within the sequence, and iteration in both directions.
2. implemented as doubly-linked lists

Doubly linked lists can store each of the elements they contain in different and unrelated storage locations.

The ordering is kept internally by the association to each element of a link to the element preceding it and a link to the element following it.

1. very similar to forward\_list

The main difference being that forward\_list objects are single-linked lists, and thus they can only be iterated forwards, in exchange for being somewhat smaller and more efficient.

1. Compared to other base standard sequence containers (array, vector and deque),

lists perform generally better in inserting, extracting and moving elements in any position within the container for which an iterator has already been obtained, and therefore also in algorithms that make intensive use of these, like sorting algorithms.

1. drawback of lists and forward\_lists

* compared to other sequence containers they lack direct access to the elements by their position;

For example, to access the sixth element in a list, one has to iterate from a known position (like the beginning or the end) to that position, which takes linear time in the distance between these.

* They also consume some extra memory to keep the linking information associated to each element

(which may be an important factor for large lists of small-sized elements).

# Container properties

## Sequence

Elements in sequence containers are ordered in a strict linear sequence.

Individual elements are accessed by their position in this sequence.

## Doubly-linked list

Each element keeps information on how to locate the next and the previous elements,

allowing constant time insert and erase operations before or after a specific element (even of entire ranges), but no direct random access.

## Allocator-aware

The container uses an allocator object to dynamically handle its storage needs.

# Member functions

[**(constructor)**](http://www.cplusplus.com/reference/list/list/list/)

Construct list (public member function )

[**(destructor)**](http://www.cplusplus.com/reference/list/list/~list/)

List destructor (public member function )

[**operator=**](http://www.cplusplus.com/reference/list/list/operator=/)

Assign content (public member function )

## Iterators

[**begin**](http://www.cplusplus.com/reference/list/list/begin/)

Return iterator to beginning (public member function )

[**end**](http://www.cplusplus.com/reference/list/list/end/)

Return iterator to end (public member function )

[**rbegin**](http://www.cplusplus.com/reference/list/list/rbegin/)

Return reverse iterator to reverse beginning (public member function )

[**rend**](http://www.cplusplus.com/reference/list/list/rend/)

Return reverse iterator to reverse end (public member function )

[**cbegin**](http://www.cplusplus.com/reference/list/list/cbegin/)

Return const\_iterator to beginning (public member function )

[**cend**](http://www.cplusplus.com/reference/list/list/cend/)

Return const\_iterator to end (public member function )

[**crbegin**](http://www.cplusplus.com/reference/list/list/crbegin/)

Return const\_reverse\_iterator to reverse beginning (public member function )

[**crend**](http://www.cplusplus.com/reference/list/list/crend/)

Return const\_reverse\_iterator to reverse end (public member function )

## Capacity

[**empty**](http://www.cplusplus.com/reference/list/list/empty/)

Test whether container is empty (public member function )

[**size**](http://www.cplusplus.com/reference/list/list/size/)

Return size (public member function )

[**max\_size**](http://www.cplusplus.com/reference/list/list/max_size/)

Return maximum size (public member function )

## Element access

[**front**](http://www.cplusplus.com/reference/list/list/front/)

Access first element (public member function )

[**back**](http://www.cplusplus.com/reference/list/list/back/)

Access last element (public member function )

## Modifiers

[**assign**](http://www.cplusplus.com/reference/list/list/assign/)

Assign new content to container (public member function )

[**emplace\_front**](http://www.cplusplus.com/reference/list/list/emplace_front/)

Construct and insert element at beginning (public member function )

[**push\_front**](http://www.cplusplus.com/reference/list/list/push_front/)

Insert element at beginning (public member function )

[**pop\_front**](http://www.cplusplus.com/reference/list/list/pop_front/)

Delete first element (public member function )

[**emplace\_back**](http://www.cplusplus.com/reference/list/list/emplace_back/)

Construct and insert element at the end (public member function )

[**push\_back**](http://www.cplusplus.com/reference/list/list/push_back/)

Add element at the end (public member function )

[**pop\_back**](http://www.cplusplus.com/reference/list/list/pop_back/)

Delete last element (public member function )

[**emplace**](http://www.cplusplus.com/reference/list/list/emplace/)

Construct and insert element (public member function )

[**insert**](http://www.cplusplus.com/reference/list/list/insert/)

Insert elements (public member function )

[**erase**](http://www.cplusplus.com/reference/list/list/erase/)

Erase elements (public member function )

[**swap**](http://www.cplusplus.com/reference/list/list/swap/)

Swap content (public member function )

[**resize**](http://www.cplusplus.com/reference/list/list/resize/)

Change size (public member function )

[**clear**](http://www.cplusplus.com/reference/list/list/clear/)

Clear content (public member function )

## Operations

[**splice**](http://www.cplusplus.com/reference/list/list/splice/)

Transfer elements from list to list (public member function )

[**remove**](http://www.cplusplus.com/reference/list/list/remove/)

Remove elements with specific value (public member function )

[**remove\_if**](http://www.cplusplus.com/reference/list/list/remove_if/)

Remove elements fulfilling condition (public member function template )

[**unique**](http://www.cplusplus.com/reference/list/list/unique/)

Remove duplicate values (public member function )

[**merge**](http://www.cplusplus.com/reference/list/list/merge/)

Merge sorted lists (public member function )

[**sort**](http://www.cplusplus.com/reference/list/list/sort/)

Sort elements in container (public member function )

[**reverse**](http://www.cplusplus.com/reference/list/list/reverse/)

Reverse the order of elements (public member function )

## Observers

[**get\_allocator**](http://www.cplusplus.com/reference/list/list/get_allocator/)

Get allocator (public member function )

# Non-member function overloads

[**relational operators (list)**](http://www.cplusplus.com/reference/list/list/operators/)

Relational operators for list (function )

[**swap (list)**](http://www.cplusplus.com/reference/list/list/swap-free/)

Exchanges the contents of two lists (function template )

# Member functions

## constructor

Construct list (public member function )

|  |  |
| --- | --- |
| default (1) | explicit list (const allocator\_type& alloc = allocator\_type()); // (until C++14) |
|  | list(); // (since C++14) |
|  | explicit list (const allocator\_type& alloc); // (since C++14) |
|  |  |
| fill (2) | list (size\_type n, const value\_type& val, const allocator\_type& alloc = allocator\_type()); // (since C++11) |
|  | explicit list (size\_type n, const value\_type& val = value\_type(),  const allocator\_type& alloc = allocator\_type()); // (until C++11) |
|  | explicit list (size\_type n); // (since C++11), (until C++14) |
|  | explicit list (size\_type n, const allocator\_type& alloc = allocator\_type()); // (since C++14) |
|  |  |
| range (3) | template <class InputIterator>  list (InputIterator first, InputIterator last, const allocator\_type& alloc = allocator\_type()); |
|  |  |
| copy (4) | list (const list& x); |
|  | list (const list& x, const allocator\_type& alloc); // (since C++11) |
|  |  |
| move (5) | list (list&& x); // (since C++11) |
|  | list (list&& x, const allocator\_type& alloc); // (since C++11) |
|  |  |
| initializer list (6) | list (initializer\_list<value\_type> il, const allocator\_type& alloc = allocator\_type()); // (since C++11) |
|  |  |

* Construct list
* Constructs a list container object
* initializing its contents depending on the constructor version used:

1. empty container constructor (default constructor)

Constructs an empty container, with no elements.

1. fill constructor

Constructs a container with n elements. Each element is a copy of val (if provided).

1. range constructor

Constructs a container with as many elements as the range [first,last),

with each element emplace-constructed from its corresponding element in that range, in the same order.

1. copy constructor (and copying with allocator)

Constructs a container with a copy of each of the elements in x, in the same order.

1. move constructor (and moving with allocator)

Constructs a container that acquires the elements of x.

If alloc is specified and is different from x's allocator, the elements are moved.

Otherwise, no elements are constructed (their ownership is directly transferred).

x is left in an unspecified but valid state.

1. initializer list constructor

Constructs a container with a copy of each of the elements in il, in the same order.

container keeps an internal copy of alloc, which is used to

* + - allocate and deallocate storage for its elements
    - construct and destroy them

If no alloc argument is passed to the constructor,

a default-constructed allocator is used, except in the following cases:

* + - The copy constructor, list (const list& x);
    - The move constructor, list (list&& x);

**Complexity**:

1. Constant
2. Linear in count
3. Linear in distance between first and last
4. Linear in size of x
5. Constant if alloc == other.get\_allocator(), otherwise constant
6. Linear in size of list il

**Return**: (nothing)

**Data Races**:

* All copied elements are accessed.
* The move constructors (5) modify x.

**Exception**:

Strong guarantee: no effects in case an exception is thrown.

undefined behavior:

* If allocator\_traits::construct is not supported with the appropriate arguments for the element constructions
* if the range specified by [first,last) is not valid

## destructor

List destructor (public member function )

|  |
| --- |
| ~list(); |

* List destructor
* Destroys the container object.
* This destroys all container elements, and deallocates all the storage capacity
* allocated by the list container using its allocator. // (until C++11)
* This calls allocator\_traits::destroy on each of the contained elements, and
* deallocates all the storage capacity allocated by the list container using its allocator. // (since C++11)

**Complexity**:

Linear in list::size (destructors)

**Return**:

(nothing)

**Data Races**:

The container and all its elements are modified.

**Exception**:

No-throw guarantee: never throws exceptions.

## operator=

Assign content (public member function )

|  |  |
| --- | --- |
| copy (1) | list& operator= (const list& x); |
|  |  |
| move (2) | list& operator= (list&& x); // (since C++11), (until C++17) |
|  | list& operator= (list&& x) noexcept; // (since C++17) |
|  |  |
| initializer list (3) | list& operator= (initializer\_list<value\_type> il);  // (since C++11) |

* Assign content
* Assigns new contents to the container, replacing its current contents, and modifying its size accordingly
* copy assignment (1) copies all the elements from x into the container (with x preserving its contents)
* move assignment (2) moves the elements of x into the container (x is left in an unspecified but valid state)
* initializer list assignment (3) Replaces the contents with those identified by initializer list il

**Complexity**:

1. Linear in the size of \*this and x
2. Linear in the size of \*this unless the allocators do not compare equal and do not propagate, in which case linear in the size of \*this and x
3. Linear in the size of \*this and il

**Return**:

\*this

**Data Races**:

* All copied elements are accessed
* The move assignment (2) modifies x
* The container and all its elements are modified

**Exception**:

Basic guarantee: if an exception is thrown, the container is in a valid state

causes undefined behavior:

* If allocator\_traits::construct is not supported with the appropriate arguments for the element constructions
* if value\_type is not copy assignable (or move assignable)

// constructing lists

#include <iostream>

#include <list>

using namespace std;

int main ()

{

// constructors used in the same order as described above:

list<int> first; // empty list of ints

list<int> second (4,100); // four ints with value 100

list<int> third (second.begin(),second.end()); // iterating through second

list<int> fourth (third); // a copy of third

// the iterator constructor can also be used to construct from arrays:

int myints[] = {16,2,77,29};

list<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );

cout << "The contents of fifth are: ";

for (list<int>::iterator it = fifth.begin(); it != fifth.end(); it++)

cout << \*it << ' ';

cout << "\n\n";

// operator=

cout << "The contents of first are: ";

for (list<int>::iterator it = first.begin(); it != first.end(); it++)

cout << \*it << ' ';

cout << "\n";

cout << "The contents of second are: ";

for (list<int>::iterator it = second.begin(); it != second.end(); it++)

cout << \*it << ' ';

cout << "\n";

first = second;

second = std::list<int>();

cout << "The contents of first are: ";

for (list<int>::iterator it = first.begin(); it != first.end(); it++)

cout << \*it << ' ';

cout << "\n";

cout << "The contents of second are: ";

for (list<int>::iterator it = second.begin(); it != second.end(); it++)

cout << \*it << ' ';

cout << "\n";

return 0;

}

Output:

The contents of fifth are: 16 2 77 29

The contents of first are:

The contents of second are: 100 100 100 100

The contents of first are: 100 100 100 100

The contents of second are:

# Iterators:

## begin

Return iterator to beginning (public member function )

|  |
| --- |
| iterator begin(); // (until C++11) |
| const\_iterator begin() const; // (until C++11) |
|  |
| iterator begin() noexcept; // (since C++11) |
| const\_iterator begin() const noexcept; // (since C++11) |

* Return iterator to beginning
* Returns an iterator pointing to the first element in the list container.
* Notice that, unlike member list::front, which returns a reference to the first element,
* this function returns a bidirectional iterator pointing to it.
* If the container is empty, the returned iterator value ( equal to end() ) shall not be dereferenced.

**Complexity**: Constant

**Return**:

* An iterator to the beginning of the sequence container.
* If the list object is const-qualified, the function returns a const\_iterator. Otherwise, it returns an iterator.

**Data Races**:

* The container is accessed (neither the const nor the non-const versions modify the container).
* No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements.
* Concurrently accessing or modifying different elements is safe.

**Exception**:

* No-throw guarantee: this member function never throws exceptions.
* The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## end

Return iterator to end (public member function )

|  |
| --- |
| iterator end(); // (until C++11) |
| const\_iterator end() const; // (until C++11) |
|  |
| iterator end() noexcept; // (since C++11) |
| const\_iterator end() const noexcept; // (since C++11) |

* Return iterator to end
* Returns an iterator referring to the past-the-end element in the list container.
* past-the-end element : theoretical element that would follow the last element in the list container.
* It does not point to any element, and thus shall not be dereferenced.
* Because the ranges used by functions of the standard library do not include the element pointed by their closing iterator,
* this function is often used in combination with list::begin to specify a range including all the elements in the container.
* If the container is empty, this function returns the same as list::begin

**Complexity**: Constant

**Return**:

* An iterator to the element past the end of the sequence.
* If the list object is const-qualified, the function returns a const\_iterator. Otherwise, it returns an iterator

**Data Races**:

* The container is accessed (neither the const nor the non-const versions modify the container).
* No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements.
* Concurrently accessing or modifying different elements is safe.

**Exception**:

* No-throw guarantee: this member function never throws exceptions.
* The copy construction or assignment of the returned iterator is also guaranteed to never throw

#include <iostream>

#include <list>

using namespace std;

int main ()

{

int myints[] = {75,23,65,42,13};

list<int> mylist (myints,myints+5);

cout << "mylist contains:";

for (list<int>::iterator it=mylist.begin(); it != mylist.end(); ++it)

cout << ' ' << \*it;

cout << '\n';

return 0;

}

Output:

mylist contains: 75 23 65 42 13

# Capacity

## empty

Test whether container is empty (public member function )

|  |
| --- |
| bool empty() const; // (until C++11) |
| bool empty() const noexcept; // (since C++11) |

* Test whether container is empty i.e. whether begin() == end()
* This function does not modify the container in any way.
* To clear the content of a list container, see list::clear.

**Complexity**: Constant

**Return**:

* true : if the container size is 0
* false : otherwise

**Data Races**:

* The container is accessed.
* No contained elements are accessed: concurrently accessing or modifying them is safe.

**Exception**:

No-throw guarantee: this member function never throws exceptions

## size

Return size (public member function )

|  |
| --- |
| size\_type size() const; // (until C++11) |
| size\_type size() const noexcept; // (since C++11) |

* Return size
* Returns the number of elements in the container, i.e. std::distance(begin(), end())

**Complexity**:

* Constant or linear // (until C++11)
* Constant // (since C++11)

**Return**: number of elements in the container

**Data Races**:

* The container is accessed
* No contained elements are accessed: concurrently accessing or modifying them is safe

**Exception**:

* No-throw guarantee: this member function never throws exceptions

## max\_size

Return maximum size (public member function )

|  |
| --- |
| size\_type max\_size() const; // (until C++11) |
| size\_type max\_size() const noexcept; // (since C++11) |

* Return maximum size
* Returns the maximum number of elements that the list container can hold i.e. std::distance(begin(), end()) for the largest container.
* This is the maximum potential size the container can reach due to known system or library implementation limitations,

but the container is by no means guaranteed to be able to reach that size:

it can still fail to allocate storage at any point before that size is reached.

**Complexity**: Constant

**Return**:

* maximum number of elements the object can hold as content

**Data Races**:

* The container is accessed.
* No contained elements are accessed: concurrently accessing or modifying them is safe

**Exception**:

* No-throw guarantee: this member function never throws exceptions

#include <iostream>

#include <list>

using namespace std;

int main ()

{

list<int> myints;

cout << "myints.empty(): " << myints.empty() << '\n';

cout << "myints.size(): " << myints.size() << '\n';

cout << "myints.max\_size(): " << myints.max\_size() << "\n\n";

for (int i=0; i<10; i++) myints.push\_back(i);

cout << "myints.empty(): " << myints.empty() << '\n';

cout << "myints.size(): " << myints.size() << '\n';

cout << "myints.max\_size(): " << myints.max\_size() << "\n\n";

myints.insert (myints.begin(),10,100);

cout << "myints.empty(): " << myints.empty() << '\n';

cout << "myints.size(): " << myints.size() << '\n';

cout << "myints.max\_size(): " << myints.max\_size() << "\n\n";

return 0;

}

Output:

myints.empty(): 1

myints.size(): 0

myints.max\_size(): 768614336404564650

myints.empty(): 0

myints.size(): 10

myints.max\_size(): 768614336404564650

myints.empty(): 0

myints.size(): 20

myints.max\_size(): 768614336404564650

# Element access:

## front

Access first element (public member function )

|  |
| --- |
| reference front(); |
| const\_reference front() const; |

* Access first element
* Returns a reference to the first element in the list container

**c.front() is equivalent to \*c.begin()**

* this function returns a direct reference'
* unlike member list::begin, which returns an iterator to this same element
* Calling this function on an empty container causes undefined behavior

**Complexity**: Constant

**Return**:

* A reference to the first element in the list container
* If the list object is const-qualified, the function returns a const\_reference. Otherwise, it returns a reference

**Data Races**:

* The container is accessed (neither the const nor the non-const versions modify the container).
* The first element is potentially accessed or modified by the caller.
* Concurrently accessing or modifying other elements is safe.

**Exception**:

* No-throw guarantee: If the container is not empty
* undefined behavior: Otherwise

## back

Access last element (public member function )

|  |
| --- |
| reference back(); |
| const\_reference back() const; |

* Access last element
* Returns a reference to the last element in the list container.

c.back() is equivalent to { auto tmp = c.end(); --tmp; return \*tmp; }

* this function returns a direct reference,
* unlike member list::end, which returns an iterator just past this element
* Calling this function on an empty container causes undefined behavior.

**Complexity**: Constant

**Return**:

* A reference to the last element in the list.
* If the list object is const-qualified, the function returns a const\_reference. Otherwise, it returns a reference.

**Data Races**:

* The container is accessed (neither the const nor the non-const versions modify the container).
* The last element is potentially accessed or modified by the caller.
* Concurrently accessing or modifying other elements is safe.

**Exception**:

* No-throw guarantee: If the container is not empty
* undefined behavior: Otherwise

// list::front

#include <iostream>

#include <list>

using namespace std;

int main ()

{

list<int> mylist;

mylist.push\_back(77);

mylist.push\_back(22);

cout << "mylist.front() " << mylist.front() << '\n';

cout << "mylist.back() " << mylist.back() << "\n\n";

mylist.push\_front(84);

mylist.push\_back(33);

cout << "mylist.front() " << mylist.front() << '\n';

cout << "mylist.back() " << mylist.back() << "\n\n";

// now front equals 77, and back 22

mylist.front() -= mylist.back();

cout << "mylist.front() " << mylist.front() << '\n';

cout << "mylist.back() " << mylist.back() << "\n\n";

return 0;

}

Output:

mylist.front() 77

mylist.back() 22

mylist.front() 84

mylist.back() 33

mylist.front() 51

mylist.back() 33

# Modifiers:

## assign

Assign new content to container (public member function )

|  |  |
| --- | --- |
| range (1) | template <class InputIterator> void assign (InputIterator first, InputIterator last); |
|  |  |
| fill (2) | void assign (size\_type n, const value\_type& val); |
|  |  |
| initializer list (3) | void assign (initializer\_list<value\_type> il); |

1. the new contents are elements constructed from each of the elements in the range [first, last), in the same order.
2. the new contents are n elements, each initialized to a copy of val.
3. the new contents are copies of the values passed as initializer list, in the same order.

* If there are changes in storage, the internal allocator is used (through its traits).
* It is also used to destroy all existing elements, and to construct the new ones.
* Any elements held in the container before the call are either assigned to or destroyed.

**Complexity**:

1. Linear in distance between first and last
2. Linear in size n
3. Linear in ilist.size()

**Return**: none

**Data Races**:

* All copied elements are accessed.
* The container is modified.
* All contained elements are modified.

**Exception**:

* Basic guarantee: if an exception is thrown, the container is in a valid state.
* undefined behavior:
  + If allocator\_traits::construct is not supported with the appropriate arguments for the element constructions, or
  + if the range specified by [first,last) is not valid

// list::assign

#include <iostream>

#include <list>

using namespace std;

int main ()

{

list<int> first;

list<int> second;

first.assign (7,100); // 7 ints with value 100

second.assign (first.begin(),first.end()); // a copy of first

cout << "Size of first: " << int (first.size()) << '\n';

cout << "Size of second: " << int (second.size()) << "\n\n";

int myints[]={1776,7,4};

first.assign (myints,myints+3); // assigning from array

cout << "Size of first: " << int (first.size()) << '\n';

cout << "Size of second: " << int (second.size()) << '\n';

return 0;

}

Output:

Size of first: 7

Size of second: 7

Size of first: 3

Size of second: 7

## emplace\_front

Construct and insert element at beginning (public member function )

|  |
| --- |
| template< class... Args >  void emplace\_front( Args&&... args ); // (since C++11), (until C++17) |
|  |
| template< class... Args >  reference emplace\_front( Args&&... args ); // (since C++17) |

* Construct and insert element at beginning
* Inserts a new element at the beginning of the list, right before its current first element.
* This new element is constructed in place using args as the arguments for its construction.
* This effectively increases the container size by one.
* The element is constructed in-place by calling allocator\_traits::construct with args forwarded.
* A similar member function exists, push\_front, which either copies or moves an existing object into the container.
* No iterators or references are invalidated.

**Complexity**: Constant

**Return**:

* (none) // (until C++17)
* A reference to the inserted element. // (since C++17)

**Data Races**:

* The container is modified.
* No contained elements are accessed: concurrently accessing or modifying them is safe.

**Exception**:

* Strong guarantee: if an exception is thrown, there are no changes in the container.
* undefined behavior: If allocator\_traits::construct is not supported with the appropriate arguments

## push\_front

Insert element at beginning (public member function )

|  |
| --- |
| void push\_front (const value\_type& val); |
| void push\_front (value\_type&& val); // (since C++11) |

* Insert element at beginning
* Inserts a new element at the beginning of the list, right before its current first element.
* The content of val is copied (or moved) to the inserted element.
* This effectively increases the container size by one.
* No iterators or references are invalidated.

**Complexity**: Constant

**Return**: none

**Data Races**:

* The container is modified.
* No existing elements are accessed (although see iterator validity above).

**Exception**:

* Strong guarantee: if an exception is thrown, there are no changes in the container.
* undefined behavior: If allocator\_traits::construct is not supported with val as argument

## pop\_front

Delete first element (public member function )

|  |
| --- |
| void pop\_front(); |

* Delete first element
* Removes and destroys the first element of the container, effectively reducing its size by one.
* If there are no elements in the container, the behavior is undefined.
* References and iterators to the erased element are invalidated.

**Complexity**: Constant

**Return**: none

**Data Races**:

* The container is modified.
* The first element is modified.
* Concurrently accessing or modifying other elements is safe.

**Exception**:

* No-throw guarantee: If the container is not empty, the function never throws exceptions
* Otherwise, it causes undefined behavior.

#include <iostream>

#include <list>

using namespace std;

int main ()

{

list< pair<int,char> > mylist;

mylist.emplace\_front(10,'a');

mylist.emplace\_front(20,'b');

cout << "mylist contains:";

for (auto& x: mylist)

cout << " (" << x.first << "," << x.second << ")";

cout << endl;

mylist.emplace\_front(30,'c');

cout << "mylist contains:";

for (auto& x: mylist)

cout << " (" << x.first << "," << x.second << ")";

cout << endl << endl;

// push\_front

list<int> mylist2 (2,100); // two ints with a value of 100

mylist2.push\_front (200);

cout << "mylist2 contains:";

for (list<int>::iterator it=mylist2.begin(); it!=mylist2.end(); ++it)

cout << ' ' << \*it;

cout << endl;

mylist2.push\_front (300);

cout << "mylist2 contains:";

for (list<int>::iterator it=mylist2.begin(); it!=mylist2.end(); ++it)

cout << ' ' << \*it;

cout << endl << endl;

// pop\_front

cout << "Popping out the elements in mylist2:";

while (!mylist2.empty())

{

cout << ' ' << mylist2.front();

mylist2.pop\_front();

}

cout << "\nFinal size of mylist2 is " << mylist2.size() << '\n';

return 0;

}

Output:

mylist contains: (20,b) (10,a)

mylist contains: (30,c) (20,b) (10,a)

mylist2 contains: 200 100 100

mylist2 contains: 300 200 100 100

Popping out the elements in mylist2: 300 200 100 100

Final size of mylist2 is 0

## emplace\_back

Construct and insert element at the end (public member function )

|  |
| --- |
| template< class... Args >  void emplace\_back( Args&&... args ); // (since C++11), (until C++17) |
|  |
| template< class... Args >  reference emplace\_back( Args&&... args ); // (since C++17) |

* Construct and insert element at the end
* Inserts a new element at the end of the list, right after its current last element.
* This new element is constructed in place using args as the arguments for its construction.
* This effectively increases the container size by one.
* The element is constructed in-place by calling allocator\_traits::construct with args forwarded.
* A similar member function exists, push\_back, which either copies or moves an existing object into the container.
* No iterators or references are invalidated.

**Complexity**: Constant

**Return**:

* (none) // (until C++17)
* A reference to the inserted element. // (since C++17)

**Data Races**:

* The container is modified.
* No contained elements are accessed: concurrently accessing or modifying them is safe.

**Exception**:

* Strong guarantee: if an exception is thrown, there are no changes in the container.
* undefined behavior: If allocator\_traits::construct is not supported with the appropriate arguments

## push\_back

Add element at the end (public member function )

|  |
| --- |
| void push\_back (const value\_type& val); |
| void push\_back (value\_type&& val); // (since C++11) |

* Add element at the end
* Adds a new element at the end of the list container, after its current last element.
* The content of val is copied (or moved) to the new element.
* This effectively increases the container size by one.
* No iterators or references are invalidated.

**Complexity**: Constant

**Return**: none

**Data Races**:

* The container is modified.
* No existing contained elements are accessed:
* concurrently accessing or modifying them is safe.

**Exception**:

* Strong guarantee: if an exception is thrown, there are no changes in the container.
* undefined behavior: If allocator\_traits::construct is not supported with val as argument

## pop\_back

Delete last element (public member function )

|  |
| --- |
| void pop\_back(); |

* Removes and destroys the last element of the container.
* Calling pop\_back on an empty container is undefined.
* References and iterators to the erased element are invalidated.

**Complexity**: Constant

**Return**: none

**Data Races**:

* The container is modified.
* The last element is modified.
* Concurrently accessing or modifying other elements is safe.

**Exception**:

* No-throw guarantee: If the container is not empty, the function never throws exceptions
* Otherwise, it causes undefined behavior.

#include <iostream>

#include <list>

using namespace std;

int main (void)

{

list< pair<int,char> > mylist;

mylist.emplace\_back(10,'a');

mylist.emplace\_back(20,'b');

cout << "mylist contains:";

for (auto& x: mylist)

cout << " (" << x.first << "," << x.second << ")";

cout << endl;

mylist.emplace\_back(30,'c');

cout << "mylist contains:";

for (auto& x: mylist)

cout << " (" << x.first << "," << x.second << ")";

cout << endl << endl;

// push\_back

list<int> mylist2 (2,100); // two ints with a value of 100

mylist2.push\_back (200);

cout << "mylist2 contains:";

for (list<int>::iterator it=mylist2.begin(); it!=mylist2.end(); ++it)

cout << ' ' << \*it;

cout << endl;

mylist2.push\_back (300);

cout << "mylist2 contains:";

for (list<int>::iterator it=mylist2.begin(); it!=mylist2.end(); ++it)

cout << ' ' << \*it;

cout << endl << endl;

// pop\_back

cout << "Popping out the elements in mylist2:";

while (!mylist2.empty())

{

cout << ' ' << mylist2.back();

mylist2.pop\_back();

}

cout << "\nFinal size of mylist2 is " << mylist2.size() << '\n';

return 0;

}

Output:

mylist contains: (10,a) (20,b)

mylist contains: (10,a) (20,b) (30,c)

mylist2 contains: 100 100 200

mylist2 contains: 100 100 200 300

Popping out the elements in mylist2: 300 200 100 100

Final size of mylist2 is 0

## emplace

Construct and insert element (public member function )

|  |
| --- |
| template< class... Args >  iterator emplace( const\_iterator pos, Args&&... args ); // (since C++11) |

* Construct and insert element
* The container is extended by inserting a new element at position.
* This new element is constructed in place using args as the arguments for its construction.
* This effectively increases the container size by one.
* Unlike other standard sequence containers, list and forward\_list objects are specifically designed to be
* efficient inserting and removing elements in any position, even in the middle of the sequence.
* The element is constructed in-place by calling allocator\_traits::construct with args forwarded.
* A similar member function exists, insert, which either copies or moves existing objects into the container.
* No iterators or references are invalidated.

**Complexity**: Constant

**Return**:

* An iterator that points to the newly emplaced element.
* Member type iterator is a bidirectional iterator type that points to an element.

**Data Races**:

* The container is modified.
* No contained elements are accessed: concurrently accessing or modifying them is safe,

although iterating ranges that include position is not.

**Exception**:

* Strong guarantee: if an exception is thrown, there are no changes in the container.
* undefined behavior:
  + If allocator\_traits::construct is not supported with the appropriate arguments, or
  + if position is not valid

## insert

Insert elements (public member function )

|  |  |
| --- | --- |
| single element (1) | iterator insert (iterator position, const value\_type& val);  // (until C++11)  iterator insert (const\_iterator position, const value\_type& val);  // (since C++11) |
|  |  |
| fill (2) | void insert (iterator position, size\_type n, const value\_type& val);  // (until C++11)  iterator insert (const\_iterator position, size\_type n, const value\_type& val); // (since C++11) |
|  |  |
| range (3) | template <class InputIterator>  void insert (iterator position, InputIterator first, InputIterator last); // (until C++11)  iterator insert (const\_iterator position, InputIterator first, InputIterator last); // (since C++11) |
|  |  |
| move (4) | iterator insert (const\_iterator position, value\_type&& val); |
|  |  |
| initializer list (5) | iterator insert (const\_iterator position,  initializer\_list<value\_type> il); |

Inserts elements at the specified location in the container.

1. inserts value before position
2. inserts count copies of the value before position
3. inserts elements from range [first, last) before position.

This overload has the same effect as overload (3) if InputIt is an integral type. (until C++11)

This overload only participates in overload resolution if InputIt qualifies as InputIterator, to avoid ambiguity with the overload (3). (since C++11)

The behavior is undefined if first and last are iterators into \*this.

1. inserts value before position
2. inserts elements from initializer list ilist before position.

* No iterators or references are invalidated.

**Complexity**:

1. Constant
2. Linear in count
3. Linear in std::distance(first, last)
4. Constant
5. Linear in ilist.size()

**Return**:

1. Iterator pointing to the inserted value
2. Iterator pointing to the first element inserted, or pos if count==0
3. Iterator pointing to the first element inserted, or pos if first==last
4. Iterator pointing to the inserted value
5. Iterator pointing to the first element inserted, or pos if ilist is empty

**Data races**:

* The container is modified.
* No contained elements are accessed.
* Concurrently accessing or modifying different elements is safe, although iterating ranges that include position is not.

**Exception safety**:

* Strong guarantee: if an exception is thrown, there are no changes in the container
* undefined behavior:
  + If allocator\_traits::construct is not supported with the appropriate arguments for the element constructions, or
  + if an invalid position or range is specified

## erase

Erase elements (public member function )

|  |
| --- |
| iterator erase (iterator position); //(until C++11) |
| iterator erase (iterator first, iterator last); //(until C++11) |
|  |
| iterator erase (const\_iterator position); //(since C++11) |
| iterator erase (const\_iterator first, const\_iterator last); //(since C++11) |

* Erase elements
  1. Removes the element at position
  2. Removes the elements in the range [first; last)
* This effectively reduces the container size by the number of elements removed, which are destroyed.
* References and iterators to the erased elements are invalidated.
* Other references and iterators are not affected.
* The iterator pos must be valid and dereferenceable.
* Thus the end() iterator (which is valid, but is not dereferencable) cannot be used as a value for pos.
* The iterator first does not need to be dereferenceable if first==last: erasing an empty range is a no-op.

**Complexity**:

1. Constant
2. Linear in the distance between first and last.

**Return**:

* An iterator pointing to the element that followed the last element erased by the function call.
* This is the container end if the operation erased the last element in the sequence.

**Data races**:

* The container is modified.
* The elements removed are modified.
* Concurrently accessing or modifying other elements is safe,

although iterating ranges that include the removed elements is not.

**Exception safety**:

* No-throw guarantee: If position (or the range) is valid, the function never throws exceptions
* Otherwise, it causes undefined behavior.

// emplace, insert and erase from list

#include <iostream>

#include <list>

using namespace std;

int main ()

{

list< pair<int,char> > mylist1;

mylist1.emplace ( mylist1.begin(), 100, 'x' );

mylist1.emplace ( mylist1.begin(), 200, 'y' );

cout << "mylist1 contains:";

for (auto& x: mylist1)

cout << " (" << x.first << "," << x.second << ")";

cout << '\n';

list<int> mylist;

list<int>::iterator it1,it2;

list<int>::iterator it;

// set some values:

for (int i=1; i<10; ++i) mylist.push\_back(i\*10);

it = mylist.begin(); // 10 20 30 40 50 60 70 80 90

++it; // it points now to number 2

mylist.insert (it,100); // 10 100 30 40 50 60 70 80 90

// "it" still points to number 2

mylist.insert (it,2,200); // 10 100 200 200 20 30 40 50 60 70 80 90

cout << "mylist contains:";

for (it1=mylist.begin(); it1!=mylist.end(); ++it1)

cout << ' ' << \*it1;

cout << '\n';

it1 = it2 = mylist.begin();

advance (it2,6);

++it1;

it1 = mylist.erase (it1);

it2 = mylist.erase (it2);

++it1;

--it2;

mylist.erase (it1,it2);

cout << "mylist contains:";

for (it1=mylist.begin(); it1!=mylist.end(); ++it1)

cout << ' ' << \*it1;

cout << '\n';

return 0;

}

Output:

mylist1 contains: (200,y) (100,x)

mylist contains: 10 100 200 200 20 30 40 50 60 70 80 90

mylist contains: 10 200 30 50 60 70 80 90

## swap

Swap content (public member function )

|  |
| --- |
| void swap( list& x ); // (until C++17) |
| void swap( list& x ) noexcept; // (since C++17) |

* Swap content
* Exchanges the content of the container by the content of x, which is another list of the same type. Sizes may differ.
* Does not invoke any move, copy, or swap operations on individual elements.
* All iterators, references and pointers remain valid for the swapped objects.
* Notice that a non-member function exists with the same name, swap, overloading that algorithm with an optimization that behaves like this member function.
* Whether the container allocators are also swapped is not defined,
* unless in the case the appropriate allocator trait indicates explicitly that they shall propagate.
* All iterators and references remain valid.
* It is unspecified whether an iterator holding the past-the-end value in this container will refer to this or the other container after the operation.

**Complexity**: Constant

**Return**: none

**Data races**:

* Both the container and x are modified.
* No contained elements are accessed by the call (although see iterator validity above).

**Exception safety**:

* No-throw guarantee:
  + If the allocators in both containers compare equal, or
  + if their allocator traits indicate that the allocators shall propagate
* Otherwise, it causes undefined behavior.

// swap lists

#include <iostream>

#include <list>

using namespace std;

int main ()

{

list<int> first (3,100); // three ints with a value of 100

list<int> second (5,200); // five ints with a value of 200

list<int>::iterator itf1=first.begin();

list<int>::iterator itf2=first.end();

list<int>::iterator its1=second.begin();

list<int>::iterator its2=second.end();

cout << \*itf1 << " " << \*itf2 << " " << \*its1 << " " << \*its2 << '\n';

cout << "first contains:";

for (list<int>::iterator it=first.begin(); it!=first.end(); it++)

cout << ' ' << \*it;

cout << '\n';

cout << "second contains:";

for (list<int>::iterator it=second.begin(); it!=second.end(); it++)

cout << ' ' << \*it;

cout << '\n';

first.swap(second);

cout << \*itf1 << " " << \*itf2 << " " << \*its1 << " " << \*its2 << '\n';

cout << "first contains:";

for (list<int>::iterator it=first.begin(); it!=first.end(); it++)

cout << ' ' << \*it;

cout << '\n';

cout << "second contains:";

for (list<int>::iterator it=second.begin(); it!=second.end(); it++)

cout << ' ' << \*it;

cout << '\n';

return 0;

}

Output:

100 3 200 5

first contains: 100 100 100

second contains: 200 200 200 200 200

100 5 200 3

first contains: 200 200 200 200 200

second contains: 100 100 100

## resize

Change size (public member function )

|  |
| --- |
| void resize (size\_type n, value\_type val = value\_type()); //(until C++11) |
| void resize (size\_type n); //(since C++11) |
| void resize (size\_type n, const value\_type& val); //(since C++11) |

* Change size
* Resizes the container so that it contains n elements.
* If **n < current container size**, the content is reduced to its first n elements, removing those beyond (and destroying them).
* If **n > current container size**,
  1. additional default-inserted elements are appended//(since C++11)
  2. additional copies of value are appended
* If val is specified, the new elements are initialized as copies of val, otherwise, they are value-initialized.
* Notice that this function changes the actual content of the container by inserting or erasing elements from it.

**Complexity**:

* If the container grows, linear in the number number of elements inserted (constructor).
* If the container shrinks, linear in the number of elements erased (destructions), plus up to linear in the size (iterator advance).

**Return**: none

**Data races**:

* The container is modified.
* Removed elements are modified. Concurrently accessing or modifying other elements is safe.

**Exception safety**:

* No-throw guarantee: If the operation decreases the size of the container, the function never throws exceptions
* Basic guarantee: Otherwise, if an exception is thrown, the container is left with a valid state: Constructing elements or allocating storage may throw.

## clear

Clear content (public member function )

|  |
| --- |
| void clear(); //(until C++11) |
| void clear() noexcept; //(since C++11) |

* Clear content
* Removes all elements from the list container (which are destroyed), and leaving the container with a size of 0.

**Complexity**: Linear in list::size (destructions)

**Return**: none

**Data races**:

1. The container is modified.
2. All contained elements are modified.

**Exception safety**:

* No-throw guarantee: this member function never throws exceptions

// resizing list

#include <iostream>

#include <list>

using namespace std;

int main ()

{

list<int> mylist;

// set some initial content:

for (int i=1; i<10; ++i) mylist.push\_back(i);

cout << "mylist contains:";

for (list<int>::iterator it=mylist.begin(); it!=mylist.end(); ++it)

cout << ' ' << \*it;

cout << '\n';

mylist.resize(5);

cout << "mylist contains:";

for (list<int>::iterator it=mylist.begin(); it!=mylist.end(); ++it)

cout << ' ' << \*it;

cout << '\n';

mylist.resize(8,100);

cout << "mylist contains:";

for (list<int>::iterator it=mylist.begin(); it!=mylist.end(); ++it)

cout << ' ' << \*it;

cout << '\n';

mylist.resize(12);

cout << "mylist contains:";

for (list<int>::iterator it=mylist.begin(); it!=mylist.end(); ++it)

cout << ' ' << \*it;

cout << '\n';

mylist.clear();

cout << "mylist contains:";

for (list<int>::iterator it=mylist.begin(); it!=mylist.end(); ++it)

cout << ' ' << \*it;

cout << '\n';

return 0;

}

Output:

mylist contains: 1 2 3 4 5 6 7 8 9

mylist contains: 1 2 3 4 5

mylist contains: 1 2 3 4 5 100 100 100

mylist contains: 1 2 3 4 5 100 100 100 0 0 0 0

mylist contains:

# Operations:

## splice

Transfer elements from list to list (public member function )

|  |  |
| --- | --- |
| entire list (1) | void splice (iterator position, list& x); //(until C++11) |
|  |  |
| single element (2) | void splice (iterator position, list& x, iterator i);  //(until C++11) |
|  |  |
| element range (3) | void splice (iterator position, list& x, iterator first, iterator last); //(until C++11) |

|  |  |
| --- | --- |
| entire list (1) | void splice (const\_iterator position, list& x); //(since C++11)  void splice (const\_iterator position, list&& x); //(since C++11) |
|  |  |
| single element (2) | void splice (const\_iterator position, list& x, const\_iterator i);  //(since C++11)  void splice (const\_iterator position, list&& x, const\_iterator i);  //(since C++11) |
|  |  |
| element range (3) | void splice (const\_iterator position, list& x, const\_iterator first, const\_iterator last); //(since C++11)  void splice (const\_iterator position, list&& x, const\_iterator first, const\_iterator last); //(since C++11) |

* Transfer elements from list to list
* Transfers elements from x into the container, inserting them at position.
* This effectively inserts those elements into the container and removes them from x,
* altering the sizes of both containers.
* The operation does not involve the construction or destruction of any element.
* They are transferred, no matter whether x is an lvalue or an rvalue, or whether the value\_type supports move-construction or not.
* The first version (1) transfers all the elements of x into the container. The behavior is undefined if this == &x.
* The second version (2) transfers only the element pointed by i from x into the container.
* The third version (3) transfers the range [first,last) from x into the container. The behavior is undefined if position is an iterator in the range [first,last).
* No elements are copied or moved, only the internal pointers of the list nodes are re-pointed.
* The behavior is undefined if: get\_allocator() != other.get\_allocator().
* No iterators or references become invalidated, the iterators to moved elements remain valid, but now refer into \*this, not into other.

**Complexity**:

1. Constant
2. Constant
3. Constant if this == &other, otherwise linear in std::distance(first, last).

**Return**: none

**Data races**:

* Both the container and x are modified.
* Concurrently accessing or modifying their elements is safe, although iterating x or ranges that include position is not.

**Exception safety**:

* undefined behavior
  + If the allocators in both containers do not compare equal, if any of the iterators or ranges specified is not valid, or
  + if x is \*this in (1), or
  + if position is in the range [first,last) in (3)
* Otherwise, No-throw guarantee: the function never throws exceptions

// splicing lists

#include <iostream>

#include <list>

using namespace std;

int main ()

{

list<int> mylist1, mylist2;

list<int>::iterator it;

// set some initial values:

for (int i=1; i<=4; ++i)

mylist1.push\_back(i); // mylist1: 1 2 3 4

for (int i=1; i<=3; ++i)

mylist2.push\_back(i\*10); // mylist2: 10 20 30

it = mylist1.begin();

++it; // points to 2

mylist1.splice (it, mylist2); // mylist1: 1 10 20 30 2 3 4

// mylist2 (empty)

// "it" still points to 2 (the 5th element)

mylist2.splice (mylist2.begin(),mylist1, it);

// mylist1: 1 10 20 30 3 4

// mylist2: 2

// "it" is now invalid.

cout << "mylist1 contains:";

for (it=mylist1.begin(); it!=mylist1.end(); ++it)

cout << ' ' << \*it;

cout << '\n';

cout << "mylist2 contains:";

for (it=mylist2.begin(); it!=mylist2.end(); ++it)

cout << ' ' << \*it;

cout << '\n';

it = mylist1.begin();

advance(it,3); // "it" points now to 30

mylist1.splice ( mylist1.begin(), mylist1, it, mylist1.end());

// mylist1: 30 3 4 1 10 20

cout << "mylist1 contains:";

for (it=mylist1.begin(); it!=mylist1.end(); ++it)

cout << ' ' << \*it;

cout << '\n';

cout << "mylist2 contains:";

for (it=mylist2.begin(); it!=mylist2.end(); ++it)

cout << ' ' << \*it;

cout << '\n';

return 0;

}

Output:

mylist1 contains: 1 10 20 30 3 4

mylist2 contains: 2

mylist1 contains: 30 3 4 1 10 20

mylist2 contains: 2

## remove

Remove elements with specific value (public member function )

|  |
| --- |
| void remove (const value\_type& val); |

* Remove elements with specific value
* Removes from the container all the elements that compare equal to val.
* This calls the destructor of these objects and reduces the container size by the number of elements removed.
* Unlike member function list::erase, which erases elements by their position (using an iterator),
* this function (list::remove) removes elements by their value.
* A similar function, list::remove\_if, exists, which allows for
* a condition other than an equality comparison to determine whether an element is removed.

**Complexity**: Linear in container size (comparisons)

**Return**: none

**Data races**:

1. The container is modified.
2. The elements removed are modified.
3. Concurrently accessing or modifying other elements is safe, although iterating through the container is not.

**Exception safety**:

* No-throw guarantee: If the equality comparison between elements is guaranteed to not throw, the function never throws exceptions
* Otherwise, if an exception is thrown, the container is left in a valid state (basic guarantee).

## remove\_if

Remove elements fulfilling condition (public member function template )

|  |
| --- |
| template <class Predicate>  void remove\_if (Predicate pred); |

* Remove elements fulfilling condition
* Removes from the container all the elements for which Predicate pred returns true.
* This calls the destructor of these objects and reduces the container size by the number of elements removed.
* The function calls pred(\*i) for each element (where i is an iterator to that element).
* Any of the elements in the list for which this returns true, are removed from the container

**Complexity**: Linear in list size (applications of pred)

**Return**: None

**Data races**:

* The container is modified.
* The elements removed are modified.
* Concurrently accessing or modifying other elements is safe, although iterating through the container is not.

**Exception safety**:

No-throw guarantee: If pred is guaranteed to not throw, the function never throws exceptions

Basic guarantee: Otherwise, if an exception is thrown, the container is left in a valid state

#include <list>

#include <iostream>

using namespace std;

// a predicate implemented as a class:

struct is\_odd {

bool operator() (const int& value) { return (value%2)==1; }

};

int main()

{

list<int> l = { 1,100,2,3,10,1, 5, 11,-1, 7, 12 };

l.remove(1); // remove both elements equal to 1

cout << "list contains: ";

for (int n : l) {

cout << n << ' ';

}

cout << '\n';

l.remove\_if([](int n){ return n > 10; }); // remove all elements greater than 10

cout << "list contains: ";

for (int n : l) {

cout << n << ' ';

}

cout << '\n';

l.remove\_if(is\_odd()); // remove all elements greater than 10

cout << "list contains: ";

for (int n : l) {

cout << n << ' ';

}

cout << '\n';

return 0;

}

Output:

list contains: 100 2 3 10 5 11 -1 7 12

list contains: 2 3 10 5 -1 7

list contains: 2 10 -1

## unique

Remove duplicate values (public member function )

|  |
| --- |
| void unique(); |
| template <class BinaryPredicate>  void unique (BinaryPredicate binary\_pred); |

* Remove duplicate values
* The version with no parameters (1), removes all but the first element from every consecutive group of equal elements in the container.
* Notice that an element is only removed from the list container if it compares equal to the element immediately preceding it.
* Thus, this function is especially useful for sorted lists.
* The second version (2), takes as argument a specific comparison function that determine the "uniqueness" of an element.
* In fact, any behavior can be implemented (and not only an equality comparison),
* but notice that the function will call binary\_pred(\*i,\*(i-1)) for all pairs of elements
* (where i is an iterator to an element, starting from the second) and remove i from the list if the predicate returns true.
* The elements removed are destroyed.

**Complexity**: Linear in (container size - 1)

**Return**: None

**Data races**:

* The container is modified.
* The elements removed are modified.
* Concurrently accessing or modifying other elements is safe, although iterating through the container is not.

**Exception safety**:

* No-throw guarantee: If binary\_pred or the comparison of elements is guaranteed to not throw, the function never throws exceptions
* Basic guarantee: if an exception is thrown, the container is left in a valid state

// list::unique

#include <iostream>

#include <cmath>

#include <list>

using namespace std;

// a binary predicate implemented as a function:

bool same\_integral\_part (double first, double second)

{ return ( int(first)==int(second) ); }

// a binary predicate implemented as a class:

struct is\_near {

bool operator() (double first, double second)

{ return (fabs(first-second)<5.0); }

};

int main ()

{

double mydoubles[]={ 12.15, 2.72, 73.0, 12.77, 3.14,

12.77, 73.35, 72.25, 15.3, 72.25 };

list<double> mylist (mydoubles,mydoubles+10);

cout << "mylist contains:";

for (list<double>::iterator it=mylist.begin(); it!=mylist.end(); ++it)

cout << ' ' << \*it;

cout << '\n';

mylist.sort(); // 2.72, 3.14, 12.15, 12.77, 12.77,

// 15.3, 72.25, 72.25, 73.0, 73.35

cout << "mylist contains:";

for (list<double>::iterator it=mylist.begin(); it!=mylist.end(); ++it)

cout << ' ' << \*it;

cout << '\n';

mylist.unique(); // 2.72, 3.14, 12.15, 12.77

// 15.3, 72.25, 73.0, 73.35

cout << "mylist contains:";

for (list<double>::iterator it=mylist.begin(); it!=mylist.end(); ++it)

cout << ' ' << \*it;

cout << '\n';

mylist.unique (same\_integral\_part); // 2.72, 3.14, 12.15

// 15.3, 72.25, 73.0

cout << "mylist contains:";

for (list<double>::iterator it=mylist.begin(); it!=mylist.end(); ++it)

cout << ' ' << \*it;

cout << '\n';

mylist.unique (is\_near()); // 2.72, 12.15, 72.25

cout << "mylist contains:";

for (list<double>::iterator it=mylist.begin(); it!=mylist.end(); ++it)

cout << ' ' << \*it;

cout << '\n';

return 0;

}

Output:

mylist contains: 12.15 2.72 73 12.77 3.14 12.77 73.35 72.25 15.3 72.25

mylist contains: 2.72 3.14 12.15 12.77 12.77 15.3 72.25 72.25 73 73.35

mylist contains: 2.72 3.14 12.15 12.77 15.3 72.25 73 73.35

mylist contains: 2.72 3.14 12.15 15.3 72.25 73

mylist contains: 2.72 12.15 72.25

## merge

Merge sorted lists (public member function )

|  |  |
| --- | --- |
| 1 | void merge (list& x); |
|  | void merge (list&& x); |
|  |  |
| 2 | template <class Compare> |
|  | void merge (list& x, Compare comp); |
|  |  |
|  | template <class Compare> |
|  | void merge (list&& x, Compare comp); |

* Merges two sorted lists into one. The lists should be sorted into ascending order.
* No elements are copied.
* The container other becomes empty after the operation.
* The function does nothing if this == &other.
* If get\_allocator() != other.get\_allocator(), the behavior is undefined.
* No iterators or references become invalidated, except that the iterators of moved elements now refer into \*this, not into other.
* The first version uses operator< to compare the elements,
* the second version uses the given comparison function comp.
* This operation is stable: for equivalent elements in the two lists,
* the elements from \*this shall always precede the elements from other,
* and the order of equivalent elements of \*this and other does not change.

**Complexity**:

at most std::distance(begin(), end()) + std::distance(other.begin(), other.end()) - 1 comparisons

**Return**: None

**Data races**:

* Both the container and x are modified.
* Concurrently accessing or modifying their elements is safe, although iterating through either container is not.

**Exception safety**:

* undefined behavior:
  + If the allocators in both containers do not compare equal,
  + if comp does not define a strict weak ordering, or
  + if the container elements are not ordered according to it,
* Otherwise, Basic guarantee: if an exception is thrown by a comparison, the container is left in a valid state
* Otherwise, strong guarantee: if an exception is thrown, there are no changes in the container

## sort

Sort elements in container (public member function )

|  |  |
| --- | --- |
| 1 | void sort(); |
|  |  |
| 2 | template <class Compare> |
|  | void sort (Compare comp); |

* Sort elements in container
* Sorts the elements in the list, altering their position within the container.
* The sorting is performed by applying an algorithm that uses either operator< (in version (1)) or comp (in version (2)) to compare elements.
* This comparison shall produce a strict weak ordering of the elements (i.e., a consistent transitive comparison, without considering its reflexiveness).
* The resulting order of equivalent elements is stable: i.e., equivalent elements preserve the relative order they had before the call.
* The entire operation does not involve the construction, destruction or copy of any element object.
* Elements are moved within the container.

**Complexity**:

Approximately **NlogN** where N is the container size

**Return**: None

**Data races**:

* The container is modified.
* All contained elements are accessed (but not modified).
* Concurrently iterating through the container is not safe.

**Exception safety**:

* Basic guarantee: if an exception is thrown, the container is in a valid state.
* It throws if the comparison or the moving operation of any element throws.

## reverse

Reverse the order of elements (public member function )

|  |
| --- |
| void reverse(); // (until C++11) |
| void reverse() noexcept; // (since C++11) |

* Reverse the order of elements
* Reverses the order of the elements in the container.
* No references or iterators become invalidated.

**Complexity**: Linear in list size

**Return**: None

**Data races**:

* The container is modified.
* No contained elements are accessed: concurrently accessing or modifying them is safe,

although iterating through the container is not.

**Exception safety**:

* No-throw guarantee: this member function never throws exceptions

#include <iostream>

#include <list>

using namespace std;

ostream& operator<<(ostream& ostr, const list<int>& list)

{

for (auto &i : list) {

ostr << " " << i;

}

return ostr;

}

int main()

{

list<int> list1 = { 5,9,0,1,3 };

list<int> list2 = { 8,7,2,6,4 };

list1.sort();

list2.sort();

cout << "list1: " << list1 << "\n";

cout << "list2: " << list2 << "\n";

list1.merge(list2);

cout << "list1: " << list1 << "\n";

cout << "list2: " << list2 << "\n";

list1.reverse();

list2.reverse();

cout << "list1: " << list1 << "\n";

cout << "list2: " << list2 << "\n";

return 0;

}

Output:

list1: 0 1 3 5 9

list2: 2 4 6 7 8

list1: 0 1 2 3 4 5 6 7 8 9

list2:

list1: 9 8 7 6 5 4 3 2 1 0

list2:

# Observers

## get\_allocator

Get allocator (public member function )

|  |
| --- |
| allocator\_type get\_allocator() const; //(until C++11) |
| allocator\_type get\_allocator() const noexcept; //(since C++11) |

* Get allocator
* Returns a copy of the allocator object associated with the list container

**Complexity**: Constant

**Return**: The associated allocator.

**Data races**:

* The container is accessed.
* No contained elements are accessed: concurrently accessing or modifying them is safe.

**Exception safety**:

* No-throw guarantee: this member function never throws exceptions.
* Copying any instantiation of the default allocator is also guaranteed to never throw.

// list::get\_allocator

#include <iostream>

#include <list>

using namespace std;

int main ()

{

list<int> mylist;

int \* p;

// allocate an array of 5 elements using mylist's allocator:

p=mylist.get\_allocator().allocate(5);

// assign some values to array

for (int i=0; i<5; ++i) p[i]=i;

cout << "The allocated array contains:";

for (int i=0; i<5; ++i) cout << ' ' << p[i];

cout << '\n';

mylist.get\_allocator().deallocate(p,5);

return 0;

}

Output:

The allocated array contains: 0 1 2 3 4

# Non-member function overloads

## relational operators (list)

Relational operators for list (function )

|  |  |
| --- | --- |
| 1 | template <class T, class Alloc>  bool operator== (const list<T,Alloc>& lhs, const list<T,Alloc>& rhs); |
| 2 | template <class T, class Alloc>  bool operator!= (const list<T,Alloc>& lhs, const list<T,Alloc>& rhs); |
| 3 | template <class T, class Alloc>  bool operator< (const list<T,Alloc>& lhs, const list<T,Alloc>& rhs); |
| 4 | template <class T, class Alloc>  bool operator<= (const list<T,Alloc>& lhs, const list<T,Alloc>& rhs); |
| 5 | template <class T, class Alloc>  bool operator> (const list<T,Alloc>& lhs, const list<T,Alloc>& rhs); |
| 6 | template <class T, class Alloc>  bool operator>= (const list<T,Alloc>& lhs, const list<T,Alloc>& rhs); |

Compares the contents of two containers

* 1-2) Checks if the contents of lhs and rhs are equal, that is, whether lhs.size() == rhs.size() and each element in lhs compares equal with the element in rhs at the same position.
* 3-6) Compares the contents of lhs and rhs lexicographically. The comparison is performed by a function equivalent to std::lexicographical\_compare.

**Complexity:** Linear in the size of the container

**Return**

1. true if the contents of the containers are equal, false otherwise
2. true if the contents of the containers are not equal, false otherwise
3. true if the contents of the lhs are lexicographically less than the contents of rhs, false otherwise
4. true if the contents of the lhs are lexicographically less than or equal the contents of rhs, false otherwise
5. true if the contents of the lhs are lexicographically greater than the contents of rhs, false otherwise
6. true if the contents of the lhs are lexicographically greater than or equal the contents of rhs, false otherwise

**Data races**:

* Both containers, lhs and rhs, are accessed.
* Up to all of their contained elements may be accessed.

**Exception safety**:

* If the type of the elements supports the appropriate operation with no-throw guarantee, the function never throws exceptions (no-throw guarantee).
* In any case, the function cannot modify its arguments.

// list comparisons

#include <iostream>

#include <list>

int main ()

{

std::list<int> a = {10, 20, 30};

std::list<int> b = {10, 20, 30};

std::list<int> c = {30, 20, 10};

if (a==b) std::cout << "a and b are equal\n";

if (b!=c) std::cout << "b and c are not equal\n";

if (b<c) std::cout << "b is less than c\n";

if (c>b) std::cout << "c is greater than b\n";

if (a<=b) std::cout << "a is less than or equal to b\n";

if (a>=b) std::cout << "a is greater than or equal to b\n";

return 0;

}

Output:

a and b are equal

b and c are not equal

b is less than c

c is greater than b

a is less than or equal to b

a is greater than or equal to b

## swap (list)

Exchanges the contents of two lists (function template )

|  |
| --- |
| template< class T, class Alloc >  void swap( list<T,Alloc>& lhs, list<T,Alloc>& rhs ); //(until C++17) |
|  |
| template< class T, class Alloc >  void swap( list<T,Alloc>& lhs, list<T,Alloc>& rhs ) noexcept; //(since C++17) |

* Specializes the std::swap algorithm for std::list.
* Swaps the contents of lhs and rhs. Calls lhs.swap(rhs).
* Both container objects must be of the same type (same template parameters), although sizes may differ.
* All iterators, references and pointers remain valid for the swapped objects.
* This is an overload of the generic algorithm swap that improves its performance by mutually transferring ownership over their assets to the other container (i.e., the containers exchange references to their data, without actually performing any element copy or movement): It behaves as if x.swap(y) was called.

**Complexity**: Constant

**Return**: (none)

**Data races**:

* Both containers, lhs and rhs, are modified

**Exception safety**:

* If the allocators in both list objects compare equal, or if their allocator traits indicate that the allocators shall propagate, the function never throws exceptions (no-throw guarantee).
* Otherwise, it causes undefined behavior.

// swap (list overload)

#include <iostream>

#include <list>

main ()

{

unsigned int i;

std::list<int> foo (3,100); // three ints with a value of 100

std::list<int> bar (5,200); // five ints with a value of 200

foo.swap(bar);

std::cout << "foo contains:";

for (std::list<int>::iterator it = foo.begin(); it!=foo.end(); ++it)

std::cout << ' ' << \*it;

std::cout << '\n';

std::cout << "bar contains:";

for (std::list<int>::iterator it = bar.begin(); it!=bar.end(); ++it)

std::cout << ' ' << \*it;

std::cout << '\n';

return 0;

}

Output:

foo contains: 200 200 200 200 200

bar contains: 100 100 100

# END