std::forward_list

Defined in header <forward_list>

```
template<
    class T,
    class Allocator = std::allocator<T>
> class forward_list;
```
(since C++11)

`std::forward_list` is a container that supports fast insertion and removal of elements from anywhere in the container. Fast random access is not supported. It is implemented as a singly-linked list and essentially does not have any overhead compared to its implementation in C. Compared to `std::list` this container provides more space efficient storage when bidirectional iteration is not needed.

Adding, removing and moving the elements within the list, or across several lists, does not invalidate the iterators currently referring to other elements in the list. However, an iterator or reference referring to an element is invalidated when the corresponding element is removed (via `erase_after`) from the list.

`std::forward_list` meets the requirements of `Container` (except for the `size` member function and that `operator==`'s complexity is always linear), `AllocatorAwareContainer` and `SequenceContainer`.

Template parameters

`T` - The type of the elements.

The requirements that are imposed on the elements depend on the actual operations performed on the container. Generally, it is required that element type is a complete type and meets the requirements of `Erasable`, but many member functions impose stricter requirements.	(until C++17)
The requirements that are imposed on the elements depend on the actual operations performed on the container. Generally, it is required that element type meets the requirements of `Erasable`, but many member functions impose stricter requirements. This container (but not its members) can be instantiated with an incomplete element type if the allocator satisfies the allocator completeness requirements.	(since C++17)

`Allocator` - An allocator that is used to acquire memory to store the elements. The type must meet the requirements of `Allocator`.

Member types

Member type	Definition
`value_type`	`T`
`allocator_type`	`Allocator`
`size_type`	Unsigned integral type (usually `std::size_t`)
`difference_type`	Signed integer type (usually `std::ptrdiff_t`)
`reference`	`value_type&`
`const_reference`	`const value_type&`
`pointer`	`std::allocator_traits<Allocator>::pointer`
`const_pointer`	`std::allocator_traits<Allocator>::const_pointer`
`iterator`	`ForwardIterator`
`const_iterator`	Constant forward iterator

Member functions

(constructor)	constructs the `forward_list` (public member function)
(destructor)	destructs the `forward_list` (public member function)
`operator=`	assigns values to the container (public member function)
`assign`	assigns values to the container (public member function)

get_allocator	returns the associated allocator (public member function)

Element access

front	access the first element (public member function)

Iterators

before_begin **cbefore_begin**	returns an iterator to the element before beginning (public member function)
begin **cbegin**	returns an iterator to the beginning (public member function)
end **cend**	returns an iterator to the end (public member function)

Capacity

empty	checks whether the container is empty (public member function)
max_size	returns the maximum possible number of elements (public member function)

Modifiers

clear	clears the contents (public member function)
insert_after	inserts elements after an element (public member function)
emplace_after	constructs elements in-place after an element (public member function)
erase_after	erases an element after an element (public member function)
push_front	inserts elements to the beginning (public member function)
emplace_front	constructs elements in-place at the beginning (public member function)
pop_front	removes the first element (public member function)
resize	changes the number of elements stored (public member function)
swap	swaps the contents (public member function)

Operations

merge	merges two sorted lists (public member function)
splice_after	moves elements from another `forward_list` (public member function)
remove **remove_if**	removes elements satisfying specific criteria (public member function)
reverse	reverses the order of the elements (public member function)
unique	removes consecutive duplicate elements (public member function)
sort	sorts the elements (public member function)

Non-member functions

operator== **operator!=** **operator<** **operator<=** **operator>** **operator>=**	lexicographically compares the values in the forward_list (function template)
std::swap(std::forward_list) (C++11)	specializes the `std::swap` algorithm (function template)

std::forward_list::forward_list

`explicit forward_list( const Allocator& alloc = Allocator() );`	(1)	(since C++11) (until C++14)
`forward_list() : forward_list( Allocator() ) {}` `explicit forward_list( const Allocator& alloc );`		(since C++14)

`forward_list( size_type count,`
`const T& value,`
`const Allocator& alloc = Allocator());`
	(2)	(since C++11)

`explicit forward_list( size_type count );`	(3)	(since C++11) (until C++14)
`explicit forward_list( size_type count, const Allocator& alloc = Allocator() );`		(since C++14)

`template< class InputIt >`
`forward_list( InputIt first, InputIt last,`
`const Allocator& alloc = Allocator() );`
	(4)	(since C++11)

`forward_list( const forward_list& other );`	(5)	(since C++11)
`forward_list( const forward_list& other, const Allocator& alloc );`	(5)	(since C++11)
`forward_list( forward_list&& other )`	(6)	(since C++11)
`forward_list( forward_list&& other, const Allocator& alloc );`	(6)	(since C++11)
`forward_list( std::initializer_list<T> init,` `const Allocator& alloc = Allocator() );`	(7)	(since C++11)

Constructs a new container from a variety of data sources, optionally using a user supplied allocator `alloc`.

1) Default constructor. Constructs an empty container.
2) Constructs the container with `count` copies of elements with value `value`.
3) Constructs the container with `count` default-inserted instances of `T`. No copies are made.
4) Constructs the container with the contents of the range `[first, last)`.

This constructor has the same effect as overload (2) if `InputIt` is an integral type.	(until C++11)
This overload only participates in overload resolution if `InputIt` satisfies `InputIterator` , to avoid ambiguity with the overload (2).	(since C++11)

5) Copy constructor. Constructs the container with the copy of the contents of `other`. If `alloc` is not provided, allocator is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator()`.
6) Move constructor. Constructs the container with the contents of `other` using move semantics. If `alloc` is not provided, allocator is obtained by move-construction from the allocator belonging to `other`.
7) Constructs the container with the contents of the initializer list `init`.

Parameters

alloc	-	allocator to use for all memory allocations of this container
count	-	the size of the container
value	-	the value to initialize elements of the container with
first, last	-	the range to copy the elements from
other	-	another container to be used as source to initialize the elements of the container with
init	-	initializer list to initialize the elements of the container with

Complexity

1) Constant
2-3) Linear in `count`
4) Linear in distance between `first` and `last`
5) Linear in size of `other`
6) Constant. If `alloc` is given and `alloc != other.get_allocator()`, then linear.
7) Linear in size of `init`

Example

Run this code

```
#include <forward_list>
#include <string>
#include <iostream>

template<typename T>
std::ostream& operator<<(std::ostream& s, const std::forward_list<T>& v) {
    s.put('[');
    char comma[3] = {'\0', ' ', '\0'};
    for (const auto& e : v) {
        s << comma << e;
        comma[0] = ',';
    }
    return s << ']';
}

int main()
{
    // c++11 initializer list syntax:
    std::forward_list<std::string> words1 {"the", "frogurt", "is", "also", "cursed"};
    std::cout << "words1: " << words1 << '\n';

    // words2 == words1
    std::forward_list<std::string> words2(words1.begin(), words1.end());
    std::cout << "words2: " << words2 << '\n';

    // words3 == words1
    std::forward_list<std::string> words3(words1);
    std::cout << "words3: " << words3 << '\n';

    // words4 is {"Mo", "Mo", "Mo", "Mo", "Mo"}
    std::forward_list<std::string> words4(5, "Mo");
    std::cout << "words4: " << words4 << '\n';
}
```

Output:

```
words1: [the, frogurt, is, also, cursed]
words2: [the, frogurt, is, also, cursed]
words3: [the, frogurt, is, also, cursed]
words4: [Mo, Mo, Mo, Mo, Mo]
```

See also

assign	assigns values to the container (public member function)
operator=	assigns values to the container (public member function)

std::forward_list::~forward_list

```
~forward_list();    (since C++11)
```

Destructs the container. The destructors of the elements are called and the used storage is deallocated. Note, that if the elements are pointers, the pointed-to objects are not destroyed.

Complexity

Linear in the size of the container.

std::forward_list::operator=

```
forward_list& operator=( const forward_list& other );          (1)  (since C++11)
forward_list& operator=( forward_list&& other );               (2)  (since C++11)
forward_list& operator=( std::initializer_list<T> ilist );     (3)  (since C++11)
```

Replaces the contents of the container.

1) Copy assignment operator. Replaces the contents with a copy of the contents of `other`. If `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment()` is `true`, the target allocator is replaced by a copy of the source allocator. If the target and the source allocators do not compare equal, the target (`*this`) allocator is used to deallocate the memory, then `other`'s allocator is used to allocate it before copying the elements. (since C++11)

2) Move assignment operator. Replaces the contents with those of `other` using move semantics (i.e. the data in `other` is moved from `other` into this container). `other` is in a valid but unspecified state afterwards. If `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment()` is `true`, the target allocator is replaced by a copy of the source allocator. If it is `false` and the source and the target allocators do not compare equal, the target cannot take ownership of the source memory and must move-assign each element individually, allocating additional memory using its own allocator as needed.

3) Replaces the contents with those identified by initializer list `ilist`.

Parameters

other - another container to use as data source

ilist - initializer list to use as data source

Return value

`*this`

Complexity

1) Linear in the size of the `other`.

2) Constant unless `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment()` is false and the allocators do not compare equal (in which case linear).

3) Linear in the size of `ilist`.

Exceptions

(since C++17)

2) `noexcept` specification:

```
noexcept(std::allocator_traits<Allocator>::is_always_equal::value)
```

Example

The following code uses to assign one `std::forward_list` to another:

Run this code

```
#include <forward_list>
#include <iostream>

void display_sizes(const std::forward_list<int> &nums1,
                   const std::forward_list<int> &nums2,
```

```
                  const std::forward_list<int> &nums3)
{
    std::cout << "nums1: " << std::distance(nums1.begin(), nums1.end())
              << " nums2: " << std::distance(nums2.begin(), nums2.end())
              << " nums3: " << std::distance(nums3.begin(), nums3.end()) << '\n';
}

int main()
{
    std::forward_list<int> nums1 {3, 1, 4, 6, 5, 9};
    std::forward_list<int> nums2;
    std::forward_list<int> nums3;

    std::cout << "Initially:\n";
    display_sizes(nums1, nums2, nums3);

    // copy assignment copies data from nums1 to nums2
    nums2 = nums1;

    std::cout << "After assigment:\n";
    display_sizes(nums1, nums2, nums3);

    // move assignment moves data from nums1 to nums3,
    // modifying both nums1 and nums3
    nums3 = std::move(nums1);

    std::cout << "After move assigment:\n";
    display_sizes(nums1, nums2, nums3);
}
```

Output:

```
Initially:
nums1: 6 nums2: 0 nums3: 0
After assigment:
nums1: 6 nums2: 6 nums3: 0
After move assigment:
nums1: 0 nums2: 6 nums3: 6
```

See also

(constructor)	constructs the `forward_list` (public member function)
`assign`	assigns values to the container (public member function)

std::forward_list::assign

`void assign( size_type count, const T& value );`	(1)	(since C++11)
`template< class InputIt >` `void assign( InputIt first, InputIt last );`	(2)	(since C++11)
`void assign( std::initializer_list<T> ilist );`	(3)	(since C++11)

Replaces the contents of the container.

1) Replaces the contents with `count` copies of value `value`
2) Replaces the contents with copies of those in the range `[first, last)`.

This overload has the same effect as overload (1) if `InputIt` is an integral type.	(until C++11)
This overload only participates in overload resolution if `InputIt` satisfies `InputIterator`.	(since C++11)

3) Replaces the contents with the elements from the initializer list `ilist`.

Parameters

`count`	-	the new size of the container
`value`	-	the value to initialize elements of the container with
`first, last`	-	the range to copy the elements from
`ilist`	-	initializer list to copy the values from

Complexity

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

Example

The following code uses `assign` to add several characters to a `std::forward_list<char>`:

Run this code

```
#include <forward_list>
#include <iostream>

int main()
{
    std::forward_list<char> characters;

    characters.assign(5, 'a');

    for (char c : characters) {
        std::cout << c << '\n';
    }

    return 0;
}
```

Output:

```
a
a
a
a
a
```

See also

(constructor)	constructs the `forward_list` (public member function)

std::forward_list::get_allocator

```
allocator_type get_allocator() const;    (since C++11)
```

Returns the allocator associated with the container.

Parameters

(none)

Return value

The associated allocator.

Complexity

Constant.

std::forward_list::front

```
reference front();                    (since C++11)
const_reference front() const;        (since C++11)
```

Returns a reference to the first element in the container.

Calling `front` on an empty container is undefined.

Parameters

(none)

Return value

reference to the first element

Complexity

Constant

Notes

For a container `c`, the expression `c.front()` is equivalent to `*c.begin()`.

Example

The following code uses `front` to display the first element of a `std::forward_list<char>`:

Run this code

```
#include <forward_list>
#include <iostream>

int main()
{
    std::forward_list<char> letters {'o', 'm', 'g', 'w', 't', 'f'};

    if (!letters.empty()) {
        std::cout << "The first character is: " << letters.front() << '\n';
    }
}
```

Output:

```
The first character is o
```

std::forward_list::before_begin, cbefore_begin

```
iterator before_begin();                          (since C++11)
const_iterator before_begin() const;              (since C++11)
const_iterator cbefore_begin() const;             (since C++11)
```

Returns an iterator to the element before the first element of the container. This element acts as a placeholder, attempting to access it results in undefined behavior. The only usage cases are in functions `insert_after()`, `emplace_after()`, `erase_after()`, `splice_after()` and the increment operator: incrementing the before-begin iterator gives exactly the same iterator as obtained from `begin()`/`cbegin()`.

Parameters

(none)

Return value

Iterator to the element before the first element.

Exceptions

`noexcept` specification: `noexcept`

Complexity

Constant.

See also

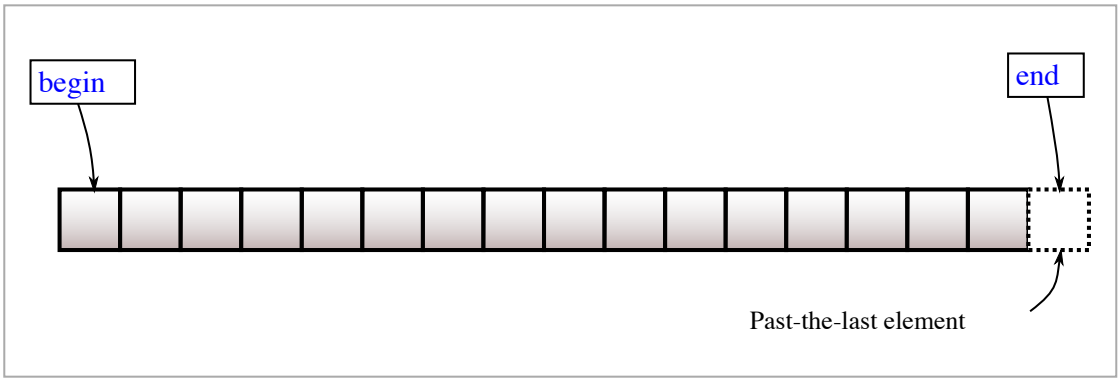
`begin` **`cbegin`**	returns an iterator to the beginning (public member function)
`end` **`cend`**	returns an iterator to the end (public member function)

std::forward_list::begin, std::forward_list::cbegin

```
iterator begin();                    (since C++11)
const_iterator begin() const;        (since C++11)
const_iterator cbegin() const;       (since C++11)
```

Returns an iterator to the first element of the container.

If the container is empty, the returned iterator will be equal to `end()`.



Parameters

(none)

Return value

Iterator to the first element

Exceptions

`noexcept` specification: `noexcept`

Complexity

Constant

Example

This section is incomplete
Reason: no example

See also

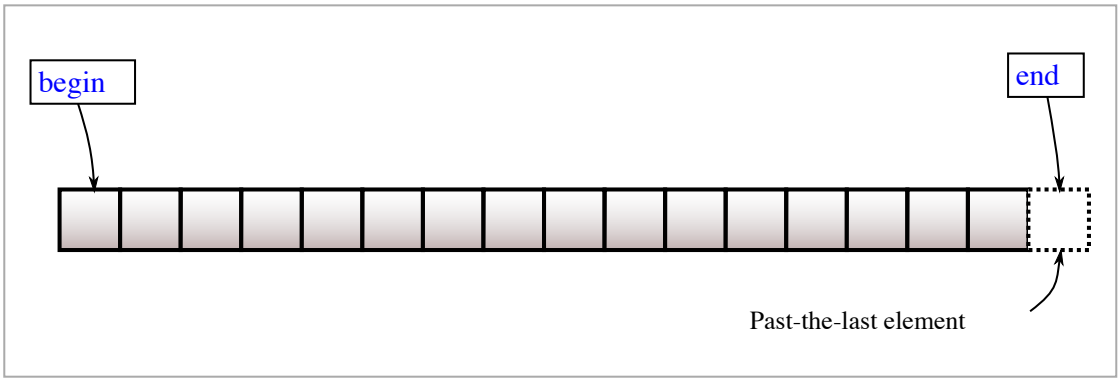
end **cend**	returns an iterator to the end (public member function)

std::forward_list::end, std::forward_list::cend

```
iterator end();                    (since C++11)
const_iterator end() const;        (since C++11)
const_iterator cend() const;       (since C++11)
```

Returns an iterator to the element following the last element of the container.

This element acts as a placeholder; attempting to access it results in undefined behavior.



Parameters

(none)

Return value

Iterator to the element following the last element.

Exceptions

`noexcept` specification: `noexcept`

Complexity

Constant.

See also

begin **cbegin**	returns an iterator to the beginning (public member function)

std::forward_list::empty

```
bool empty() const;    (since C++11)
```

Checks if the container has no elements, i.e. whether `begin() == end()`.

Parameters

(none)

Return value

`true` if the container is empty, `false` otherwise

Exceptions

`noexcept` specification: `noexcept`

Complexity

Constant.

Example

The following code uses `empty` to check if a `std::forward_list<int>` contains any elements:

Run this code

```
#include <forward_list>
#include <iostream>

int main()
{
    std::forward_list<int> numbers;
    std::cout << "Initially, numbers.empty(): " << numbers.empty() << '\n';

    numbers.push_front(42);
    numbers.push_front(13317);
    std::cout << "After adding elements, numbers.empty(): " << numbers.empty() << '\n';
}
```

Output:

```
Initially, numbers.empty(): 1
After adding elements, numbers.empty(): 0
```

std::forward_list::max_size

```
size_type max_size() const;    (since C++11)
```

Returns the maximum number of elements the container is able to hold due to system or library implementation limitations, i.e. `std::distance(begin(), end())` for the largest container.

Parameters

(none)

Return value

Maximum number of elements.

Exceptions

`noexcept` specification: `noexcept`

Complexity

Constant.

Notes

This value is typically equal to `std::numeric_limits<size_type>::max()`, and reflects the theoretical limit on the size of the container. At runtime, the size of the container may be limited to a value smaller than `max_size()` by the amount of RAM available.

Example

Run this code

```
#include <iostream>
#include <forward_list>

int main()
{
    std::forward_list<char> s;
    std::cout << "Maximum size of a 'forward_list' is " << s.max_size() << "\n";
}
```

Possible output:

```
Maximum size of a 'forward_list' is 18446744073709551615
```

std::forward_list::clear

```
void clear();    (since C++11)
```

Removes all elements from the container.

Invalidates any references, pointers, or iterators referring to contained elements. May invalidate any past-the-end iterators.

Parameters

(none)

Return value

(none)

Exceptions

`noexcept` specification: `noexcept`

Complexity

Linear in the size of the container.

`clear` is defined in terms of `erase`, which has linear complexity.	(until C++11)
complexity of `clear` is omitted	(since C++11) (until C++14)
`clear` has linear complexity for sequence containers.	(since C++14)

See also

`erase_after`	erases an element after an element (public member function)

std::forward_list::insert_after

`iterator insert_after( const_iterator pos, const T& value );`	(1)	(since C++11)
`iterator insert_after( const_iterator pos, T&& value );`	(2)	(since C++11)
`iterator insert_after( const_iterator pos, size_type count, const T& value );`	(3)	(since C++11)
`template< class InputIt >` `iterator insert_after( const_iterator pos, InputIt first, InputIt last );`	(4)	(since C++11)
`iterator insert_after( const_iterator pos, std::initializer_list<T> ilist );`	(5)	(since C++11)

Inserts elements after the specified position in the container.

1-2) inserts `value` after the element pointed to by `pos`

3) inserts `count` copies of the `value` after the element pointed to by `pos`

4) inserts elements from range `[first, last)` after the element pointed to by `pos`. The behavior is undefined if `first` and `last` are iterators into `*this`.

5) inserts elements from initializer list `ilist`.

No iterators or references are invalidated.

Parameters

pos	-	element after which the content will be inserted
value	-	element value to insert
first, last	-	the range of elements to insert
ilist	-	initializer list to insert the values from

Type requirements

- `InputIt` must meet the requirements of `InputIterator`.

Return value

1-2) Iterator to the inserted element.

3) Iterator to the last element inserted, or `pos` if `count==0`.

4) Iterator to the last element inserted, or `pos` if `first==last`.

5) Iterator to the last element inserted, or `pos` if `ilist` is empty.

Exceptions

If an exception is thrown during `insert_after` there are no effects (strong exception guarantee).

Complexity

1-2) Constant.

3) Linear in `count`

4) Linear in `std::distance(first, last)`

5) Linear in `ilist.size()`

Example

This section is incomplete
Reason: no example

See also

emplace_after	constructs elements in-place after an element (public member function)
push_front	inserts elements to the beginning (public member function)

std::forward_list::emplace_after

```
template< class... Args >
iterator emplace_after( const_iterator pos, Args&&... args );
```
(since C++11)

Inserts a new element into a position after the specified position in the container. The element is constructed in-place, i.e. no copy or move operations are performed. The constructor of the element is called with exactly the same arguments, as supplied to the function.

No iterators or references are invalidated.

Parameters

pos - iterator after which the new element will be constructed

args - arguments to forward to the constructor of the element

Return value

iterator to the new element.

Complexity

Constant.

Exceptions

If an exception is thrown (e.g. by the constructor), the container is left unmodified, as if this function was never called (strong exception guarantee).

See also

insert_after	inserts elements after an element (public member function)

std::forward_list::erase_after

`iterator erase_after( const_iterator position );`	(1)	(since C++11)
`iterator erase_after( const_iterator first, const_iterator last );`	(2)	(since C++11)

Removes specified elements from the container.

1) Removes the element following `pos`.
2) Removes the elements in the range `(first; last)`.

Parameters

pos	-	iterator to the element preceding the element to remove
first, last	-	range of elements to remove

Return value

1) Iterator to the element following the erased one, or `end()` if no such element exists.
2) `last`

Complexity

1) Constant.
2) Linear in distance between `first` and `last`.

Example

Run this code

```
#include <forward_list>
#include <iterator>
#include <iostream>
int main()
{
    std::forward_list<int> l = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    //    l.erase( l.begin() ); // ERROR: No function erase

    l.erase_after( l.before_begin() ); // Removes first element

    for( auto n : l ) std::cout << n << " ";
    std::cout << '\n';

    auto fi= std::next( l.begin() );
    auto la= std::next( fi, 3 );

    l.erase_after( fi, la );

    for( auto n : l ) std::cout << n << " ";
    std::cout << '\n';
}
```

Output:

```
2 3 4 5 6 7 8 9
2 3 6 7 8 9
```

See also

`clear`	clears the contents (public member function)

std::forward_list::push_front

```
void push_front( const T& value );    (since C++11)
void push_front( T&& value );         (since C++11)
```

Prepends the given element `value` to the beginning of the container.

No iterators or references are invalidated.

Parameters

`value` - the value of the element to prepend

Return value

(none)

Complexity

Constant.

Exceptions

If an exception is thrown, this function has no effect (strong exception guarantee).

See also

`emplace_front`	constructs elements in-place at the beginning (public member function)
`pop_front`	removes the first element (public member function)

std::forward_list::emplace_front

```
template< class... Args >
void emplace_front( Args&&... args );
```
(since C++11)

Inserts a new element to the beginning of the container. The element is constructed through `std::allocator_traits::construct`, which typically uses placement-new to construct the element in-place at the location provided by the container. The arguments `args...` are forwarded to the constructor as `std::forward<Args>(args)...`.

No iterators or references are invalidated.

Parameters

`args` - arguments to forward to the constructor of the element

Type requirements

- `T (the container's element type)` must meet the requirements of `EmplaceConstructible`.

Return value

(none)

Complexity

Constant.

Exceptions

If an exception is thrown, this function has no effect (strong exception guarantee).

See also

`push_front`	inserts elements to the beginning (public member function)

std::forward_list::pop_front

```
void pop_front();    (since C++11)
```

Removes the first element of the container.

References and iterators to the erased element are invalidated.

Parameters

(none)

Return value

(none)

Complexity

Constant.

Exceptions

Does not throw.

See also

push_front	inserts elements to the beginning (public member function)

std::forward_list::resize

```
void resize( size_type count );                                  (1)
void resize( size_type count, const value_type& value );         (2)
```

Resizes the container to contain `count` elements.

If the current size is greater than `count`, the container is reduced to its first `count` elements.

If the current size is less than `count`,

1) additional default-inserted elements are appended
2) additional copies of `value` are appended

Parameters

`count` - new size of the container
`value` - the value to initialize the new elements with

Type requirements

- `T` must meet the requirements of `DefaultInsertable` in order to use overload (1).
- `T` must meet the requirements of `CopyInsertable` in order to use overload (2).

Return value

(none)

Complexity

Linear in the difference between the current size and `count`.

Example

Run this code

```
#include <iostream>
#include <forward_list>
int main()
{
    std::forward_list<int> c = {1, 2, 3};
    std::cout << "The forward_list holds: ";
    for(auto& el: c) std::cout << el << ' ';
    std::cout << '\n';
    c.resize(5);
    std::cout << "After resize up 5: ";
    for(auto& el: c) std::cout << el << ' ';
    std::cout << '\n';
    c.resize(2);
    std::cout << "After resize down to 2: ";
    for(auto& el: c) std::cout << el << ' ';
    std::cout << '\n';
}
```

Output:

```
The forward_list holds: 1 2 3
After resize up 5: 1 2 3 0 0
After resize down to 2: 1 2
```

See also

`insert_after`	inserts elements after an element (public member function)
`erase_after`	erases an element after an element (public member function)

std::forward_list::swap

```
void swap( forward_list& other );    (since C++11)
```

Exchanges the contents of the container with those of `other`. Does not invoke any move, copy, or swap operations on individual elements.

All iterators and references remain valid. It is unspecified whether an iterator holding the past-the-end value in this container will refer to the this or the other container after the operation.

If `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is true, then the allocators are exchanged using an unqualified call to non-member `swap`. Otherwise, they are not swapped (and if `get_allocator() != other.get_allocator()`, the behavior is undefined). (since C++11)

Parameters

other - container to exchange the contents with

Return value

(none)

Exceptions

(none)	(until C++17)
`noexcept` specification: `noexcept(std::allocator_traits<Allocator>::is_always_equal::value)`	(since C++17)

Complexity

Constant.

See also

std::swap`(std::forward_list)` (C++11)	specializes the `std::swap` algorithm (function template)

std::forward_list::merge

`void merge( forward_list& other );`	(1)	(since C++11)
`void merge( forward_list&& other );`	(1)	(since C++11)
`template <class Compare>` `void merge( forward_list& other, Compare comp );`	(2)	(since C++11)
`template <class Compare>` `void merge( forward_list&& other, Compare comp );`	(2)	(since C++11)

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.

Parameters

other - another container to merge

comp - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than (i.e. is ordered *before*) the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `forward_list<T,Allocator>::const_iterator` can be dereferenced and then implicitly converted to both of them.

Return value

(none)

Exceptions

If an exception is thrown, this function has no effect (strong exception guarantee), except if the exception comes from the comparison function.

Example

Run this code

```
#include <iostream>
#include <forward_list>

std::ostream& operator<<(std::ostream& ostr, const std::forward_list<int>& list)
{
    for (auto &i : list) {
        ostr << " " << i;
    }
    return ostr;
}
```

```
int main()
{
    std::forward_list<int> list1 = { 5,9,0,1,3 };
    std::forward_list<int> list2 = { 8,7,2,6,4 };

    list1.sort();
    list2.sort();
    std::cout << "list1:  " << list1 << "\n";
    std::cout << "list2:  " << list2 << "\n";
    list1.merge(list2);
    std::cout << "merged: " << list1 << "\n";
}
```

Output:

```
list1:   0 1 3 5 9
list2:   2 4 6 7 8
merged:  0 1 2 3 4 5 6 7 8 9
```

Complexity

at most `size() + other.size() - 1` comparisons.

See also

`splice_after`	moves elements from another `forward_list` (public member function)

std::forward_list::splice_after

`void splice_after( const_iterator pos, forward_list& other );`	(1)	(since C++11)
`void splice_after( const_iterator pos, forward_list&& other );`	(1)	(since C++11)
`void splice_after( const_iterator pos, forward_list& other, const_iterator it );`	(2)	(since C++11)
`void splice_after( const_iterator pos, forward_list&& other, const_iterator it );`	(2)	(since C++11)
`void splice_after( const_iterator pos, forward_list& other, const_iterator first, const_iterator last );`	(3)	(since C++11)
`void splice_after( const_iterator pos, forward_list&& other, const_iterator first, const_iterator last );`	(3)	(since C++11)

Moves elements from another `forward_list` to `*this`.

No elements are copied. `pos` is a valid iterator in `*this` or is the `before_begin()` iterator. The behavior is undefined if `get_allocator() != other.get_allocator()`. No iterators or references become invalidated, the iterators to the moved elements now refer into `*this`, not into `other`.

1) Moves all elements from `other` into `*this`. The elements are inserted after the element pointed to by `pos`. The container `other` becomes empty after the operation. The behavior is undefined if `this == &other`
2) Moves the element pointed to by the iterator following `it` from `other` into `*this`. The element is inserted after the element pointed to by `pos`. Has no effect if `pos==it` or if `pos==++it`.
3) Moves the elements in the range `(first, last)` from `other` into `*this`. The elements are inserted after the element pointed to by `pos`. The element pointed-to by `first` is not moved. The behavior is undefined if `pos` is an iterator in the range `(first,last)`.

Parameters

pos	-	element after which the content will be inserted
other	-	another container to move the content from
it	-	iterator preceding the iterator to the element to move from `other` to `*this`
first, last	-	the range of elements to move from `other` to `*this`

Return value

(none)

Complexity

1) Linear in the size of `other`
2) Constant
3) Linear in `std::distance(first, last)`

Example

Demonstrates the meaning of open interval (first, last) in the third form of splice_after(): the first element of l1 is not moved.

Run this code

```
#include <iostream>
#include <forward_list>

int main()
{
```

```
    std::forward_list<int> l1 = {1,2,3,4,5};
    std::forward_list<int> l2 = {10,11,12};

    l2.splice_after(l2.cbegin(), l1, l1.cbegin(), l1.cend());
    // not equivalent to l2.splice_after(l2.cbegin(), l1);

    for(int n : l1)
        std::cout << n << ' ';
    std::cout << '\n';

    for(int n : l2)
        std::cout << n << ' ';
    std::cout << '\n';
}
```

Output:

```
1
10 2 3 4 5 11 12
```

See also

merge	merges two sorted lists (public member function)
remove **remove_if**	removes elements satisfying specific criteria (public member function)

std::forward_list::remove, remove_if

```
void remove( const T& value );                          (since C++11)
template< class UnaryPredicate >
void remove_if( UnaryPredicate p );                     (since C++11)
```

Removes all elements satisfying specific criteria. The first version removes all elements that are equal to `value`, the second version removes all elements for which predicate `p` returns `true`.

Parameters

value - value of the elements to remove

p - unary predicate which returns `true` if the element should be removed.

The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type &a);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it.
The type `Type` must be such that an object of type `forward_list<T,Allocator>::const_iterator` can be dereferenced and then implicitly converted to `Type`.

Return value

(none)

Complexity

Linear in the size of the container

Example

Run this code

```
#include <forward_list>
#include <iostream>

int main()
{
    std::forward_list<int> l = { 1,100,2,3,10,1,11,-1,12 };

    l.remove(1); // remove both elements equal to 1
    l.remove_if([](int n){ return n > 10; }); // remove all elements greater than 10

    for (int n : l) {
        std::cout << n << ' ';
    }
    std::cout << '\n';
}
```

Output:

```
2 3 10 -1
```

See also

remove	removes elements satisfying specific criteria
remove_if	(function template)

std::forward_list::reverse

```
void reverse();    (since C++11)
```

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

Parameters

(none)

Return value

(none)

Example

Run this code

```
#include <iostream>
#include <forward_list>

std::ostream& operator<<(std::ostream& ostr, const std::forward_list<int>& list)
{
    for (auto &i : list) {
        ostr << " " << i;
    }
    return ostr;
}

int main()
{
    std::forward_list<int> list = { 8,7,5,9,0,1,3,2,6,4 };

    std::cout << "before:     " << list << "\n";
    list.sort();
    std::cout << "ascending:  " << list << "\n";
    list.reverse();
    std::cout << "descending: " << list << "\n";
}
```

Output:

```
before:      8 7 5 9 0 1 3 2 6 4
ascending:   0 1 2 3 4 5 6 7 8 9
descending:  9 8 7 6 5 4 3 2 1 0
```

Complexity

Linear in the size of the container

See also

sort	sorts the elements (public member function)

std::forward_list::unique

```
void unique();                                          (1)  (since C++11)
template< class BinaryPredicate >
void unique( BinaryPredicate p );                       (2)  (since C++11)
```

Removes all *consecutive* duplicate elements from the container. Only the first element in each group of equal elements is left. The first version uses `operator==` to compare the elements, the second version uses the given binary predicate `p`.

Parameters

p - binary predicate which returns `true` if the elements should be treated as equal.

The signature of the predicate function should be equivalent to the following:

```
bool pred(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `forward_list<T,Allocator>::const_iterator` can be dereferenced and then implicitly converted to both of them.

Return value

(none)

Complexity

Linear in the size of the container

Example

Run this code

```
#include <iostream>
#include <forward_list>

int main()
{
  std::forward_list<int> x = {1, 2, 2, 3, 3, 2, 1, 1, 2};

  std::cout << "contents before:";
  for (auto val : x)
    std::cout << ' ' << val;
  std::cout << '\n';

  x.unique();
  std::cout << "contents after unique():";
  for (auto val : x)
    std::cout << ' ' << val;
  std::cout << '\n';

  return 0;
}
```

Output:

```
contents before: 1 2 2 3 3 2 1 1 2
contents after unique(): 1 2 3 2 1 2
```

See also

unique	removes consecutive duplicate elements in a range (function template)

std::forward_list::sort

`void sort();`	(1)	(since C++11)
`template< class Compare >` `void sort( Compare comp );`	(2)	(since C++11)

Sorts the elements in ascending order. The order of equal elements is preserved. The first version uses `operator<` to compare the elements, the second version uses the given comparison function `comp`.

Parameters

comp - comparison function object (i.e. an object that satisfies the requirements of `Compare`) which returns `true` if the first argument is *less* than (i.e. is ordered *before*) the second.

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

The signature does not need to have `const &`, but the function object must not modify the objects passed to it.
The types `Type1` and `Type2` must be such that an object of type `forward_list<T,Allocator>::const_iterator` can be dereferenced and then implicitly converted to both of them.

Return value

(none)

Example

Run this code

```
#include <iostream>
#include <functional>
#include <forward_list>

std::ostream& operator<<(std::ostream& ostr, const std::forward_list<int>& list)
{
    for (auto &i : list) {
        ostr << " " << i;
    }
    return ostr;
}

int main()
{
    std::forward_list<int> list = { 8,7,5,9,0,1,3,2,6,4 };

    std::cout << "before:     " << list << "\n";
    list.sort();
    std::cout << "ascending:  " << list << "\n";
    list.sort(std::greater<int>());
    std::cout << "descending: " << list << "\n";
}
```

Output:

```
before:      8 7 5 9 0 1 3 2 6 4
ascending:   0 1 2 3 4 5 6 7 8 9
descending:  9 8 7 6 5 4 3 2 1 0
```

Complexity

$N \cdot log(N)$ comparisons, where N is the size of the container.

operator==,!=,<,<=,>,>=(std::forward_list)

```
template< class T, class Alloc >
bool operator==( const forward_list<T,Alloc>& lhs,      (1)
                 const forward_list<T,Alloc>& rhs );
template< class T, class Alloc >
bool operator!=( const forward_list<T,Alloc>& lhs,      (2)
                 const forward_list<T,Alloc>& rhs );
template< class T, class Alloc >
bool operator<( const forward_list<T,Alloc>& lhs,       (3)
                const forward_list<T,Alloc>& rhs );
template< class T, class Alloc >
bool operator<=( const forward_list<T,Alloc>& lhs,      (4)
                 const forward_list<T,Alloc>& rhs );
template< class T, class Alloc >
bool operator>( const forward_list<T,Alloc>& lhs,       (5)
                const forward_list<T,Alloc>& rhs );
template< class T, class Alloc >
bool operator>=( const forward_list<T,Alloc>& lhs,      (6)
                 const forward_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`.

Parameters

`lhs, rhs` - containers whose contents to compare

- `T` must meet the requirements of `EqualityComparable` in order to use overloads (1-2).
- `T` must meet the requirements of `LessThanComparable` in order to use overloads (3-6). The ordering relation must establish total order.

Return value

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

Complexity

Linear in the size of the container

std::swap(std::forward_list)

```
template< class T, class Alloc >
void swap( forward_list<T,Alloc>& lhs,      (since C++11)
           forward_list<T,Alloc>& rhs );
```

Specializes the `std::swap` algorithm for `std::forward_list`. Swaps the contents of `lhs` and `rhs`. Calls `lhs.swap(rhs)`.

Parameters

`lhs, rhs` - containers whose contents to swap

Return value

(none)

Complexity

Constant.

Exceptions

(since C++17)

`noexcept` specification:

```
noexcept(noexcept(lhs.swap(rhs)))
```

See also

`swap`	swaps the contents (public member function)