forward_list Class

Describes an object that controls a varying-length sequence of elements. The sequence is stored as a singly-linked list of nodes, each containing a member of type `Type`.

Syntax

C++
```
template <class Type,
    class Allocator = allocator<Type>>
class forward_list
```

Parameters

Type
The element data type to be stored in the forward_list.

Allocator
The stored allocator object that encapsulates details about the forward_list allocation and deallocation of memory. This parameter is optional. The default value is allocator<Type>.

Remarks

A `forward_list` object allocates and frees storage for the sequence it controls through a stored object of class *Allocator* that is based on allocator Class (commonly known as `std::allocator`). For more information, see Allocators. An allocator object must have the same external interface as an object of type `allocator`.

Note

The stored allocator object is not copied when the container object is assigned.

Iterators, pointers and references might become invalid when elements of their controlled sequence are erased through `forward_list`. Insertions and splices performed on the controlled sequence through `forward_list` do not invalidate iterators.

Additions to the controlled sequence might occur by calls to forward_list::insert_after, which is the only member function that calls the constructor `Type(const T&)`. `forward_list` might also call move constructors. If such an expression throws an exception, the container object inserts no new elements and rethrows the exception. Thus, an object of type `forward_list` is left in a known state when such exceptions occur.

Members

Constructors

Name	Description
forward_list	Constructs an object of type `forward_list`.

Typedefs

Name	Description
allocator_type	A type that represents the allocator class for a forward list object.
const_iterator	A type that provides a constant iterator for the forward list.

Name	Description
const_pointer	A type that provides a pointer to a `const` element in a forward list.
const_reference	A type that provides a constant reference to an element in the forward list.
difference_type	A signed integer type that can be used to represent the number of elements of a forward list in a range between elements pointed to by iterators.
iterator	A type that provides an iterator for the forward list.
pointer	A type that provides a pointer to an element in the forward list.
reference	A type that provides a reference to an element in the forward list.
size_type	A type that represents the unsigned distance between two elements.
value_type	A type that represents the type of element stored in a forward list.

Functions

Name	Description
assign	Erases elements from a forward list and copies a new set of elements to a target forward list.
before_begin	Returns an iterator addressing the position before the first element in a forward list.
begin	Returns an iterator addressing the first element in a forward list.
cbefore_begin	Returns a const iterator addressing the position before the first element in a forward list.
cbegin	Returns a const iterator addressing the first element in a forward list.
cend	Returns a const iterator that addresses the location succeeding the last element in a forward list.
clear	Erases all the elements of a forward list.
emplace_after	Move constructs a new element after a specified position.
emplace_front	Adds an element constructed in place to the beginning of the list.
empty	Tests whether a forward list is empty.
end	Returns an iterator that addresses the location succeeding the last element in a forward list.
erase_after	Removes elements from the forward list after a specified position.
front	Returns a reference to the first element in a forward list.
get_allocator	Returns a copy of the allocator object used to construct a forward list.
insert_after	Adds elements to the forward list after a specified position.
max_size	Returns the maximum length of a forward list.
merge	Removes the elements from the argument list, inserts them into the target forward list, and orders the new, combined set of elements in ascending order or in some other specified order.
pop_front	Deletes the element at the beginning of a forward list.
push_front	Adds an element to the beginning of a forward list.
remove	Erases elements in a forward list that matches a specified value.
remove_if	Erases elements from a forward list for which a specified predicate is satisfied.
resize	Specifies a new size for a forward list.
reverse	Reverses the order in which the elements occur in a forward list.
sort	Arranges the elements in ascending order or with an order specified by a predicate.
splice_after	Restitches links between nodes.
swap	Exchanges the elements of two forward lists.
unique	Removes adjacent elements that pass a specified test.

Operators

Name	Description
operator=	Replaces the elements of the forward list with a copy of another forward list.

allocator_type

A type that represents the allocator class for a forward list object.

```
C++
typedef Allocator allocator_type;
```

Remarks

`allocator_type` is a synonym for the template parameter Allocator.

assign

Erases elements from a forward list and copies a new set of elements to a target forward list.

```
C++
void assign(
    size_type Count,
    const Type& Val);

void assign(
    initializer_list<Type> IList);

template <class InputIterator>
void assign(InputIterator First, InputIterator Last);
```

Parameters

first
The beginning of the replacement range.

last
The end of the replacement range.

count
The number of elements to assign.

val
The value to assign each element.

Type
The type of the value.

IList
The initializer_list to copy.

Remarks

If the forward_list is an integer type, the first member function behaves the same as `assign((size_type)First, (Type)Last)`. Otherwise, the first member function replaces the sequence controlled by **`*this`** with the sequence [`First, Last`), which must not overlap the initial controlled sequence.

The second member function replaces the sequence controlled by **`*this`** with a repetition of `Count` elements of value `Val`.

The third member function copies the elements of the initializer_list into the forward_list.

before_begin

Returns an iterator addressing the position before the first element in a forward list.

C++
```
const_iterator before_begin() const;
iterator before_begin();
```

Return Value

A forward iterator that points just before the first element of the sequence (or just before the end of an empty sequence).

begin

Returns an iterator addressing the first element in a forward list.

C++
```
const_iterator begin() const;
iterator begin();
```

Return Value

A forward iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

cbefore_begin

Returns a const iterator addressing the position before the first element in a forward list.

C++
```
const_iterator cbefore_begin() const;
```

Return Value

A forward iterator that points just before the first element of the sequence (or just before the end of an empty sequence).

cbegin

Returns a **`const`** iterator that addresses the first element in the range.

C++
```
const_iterator cbegin() const;
```

Return Value

A **`const`** forward-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range cannot be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **`const`**) container of any kind that supports `begin()` and `cbegin()`.

C++
```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();
// i2 is Container<T>::const_iterator
```

cend

Returns a **`const`** iterator that addresses the location just beyond the last element in a range.

C++
```
const_iterator cend() const;
```

Return Value

A forward-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **`const`**) container of any kind that supports `end()` and `cend()`.

C++
```
auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

The value returned by `cend` should not be dereferenced.

clear

Erases all the elements of a forward list.

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

Remarks

This member function calls `erase_after(before_begin(), end())`.

const_iterator

A type that provides a constant iterator for the forward list.

C++
```
typedef implementation-defined const_iterator;
```

Remarks

`const_iterator` describes an object that can serve as a constant forward iterator for the controlled sequence. It is described here as a synonym for an implementation-defined type.

const_pointer

A type that provides a pointer to a **`const`** element in a forward list.

C++
```
typedef typename Allocator::const_pointer
    const_pointer;
```

const_reference

A type that provides a constant reference to an element in the forward list.

C++
```
typedef typename Allocator::const_reference const_reference;
```

difference_type

A signed integer type that can be used to represent the number of elements of a forward list in a range between elements pointed to by iterators.

C++
```
typedef typename Allocator::difference_type difference_type;
```
`difference_type` describes an object that can represent the difference between the addresses of any two elements in the controlled sequence.

emplace_after

Move constructs a new element after a specified position.

C++
```
template <class T>
iterator emplace_after(const_iterator Where, Type&& val);
```

Parameters

Where
The position in the target forward list where the new element is constructed.

val
The constructor argument.

Return Value

An iterator that designates the newly inserted element.

Remarks

This member function inserts an element with the constructor arguments *val* just after the element pointed to by *Where* in the controlled sequence. Its behavior is otherwise the same as forward_list::insert_after.

emplace_front

Adds an element constructed in place to the beginning of the list.

C++
```
template <class Type>
    void emplace_front(Type&& val);
```

Parameters

val
The element added to the beginning of the forward list.

Remarks

This member function inserts an element with the constructor arguments `_ val` at the end of the controlled sequence.

If an exception is thrown, the container is left unaltered and the exception is rethrown.

empty

Tests whether a forward list is empty.

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

Return Value

`true` if the forward list is empty; otherwise, **`false`**.

end

Returns an iterator that addresses the location succeeding the last element in a forward list.

C++
```
const_iterator end() const;
iterator end();
```

Return Value

A forward iterator that points just beyond the end of the sequence.

erase_after

Removes elements from the forward list after a specified position.

C++
```
iterator erase_after(const_iterator Where);
iterator erase_after(const_iterator first, const_iterator last);
```

Parameters

Where
The position in the target forward list where the element is erased.

first
The beginning of the range to erase.

last
The end of the range to erase.

Return Value

An iterator that designates the first element remaining beyond any elements removed, or forward_list::end if no such element exists.

Remarks

The first member function removes the element of the controlled sequence just after *Where*.

The second member function removes the elements of the controlled sequence in the range `( first, last)` (neither end point is included).

Erasing `N` elements causes `N` destructor calls. Reallocation occurs, so iterators and references become invalid for the erased elements.

The member functions never throw an exception.

forward_list

Constructs an object of type `forward_list`.

C++
```
forward_list();
explicit forward_list(const Allocator& Al);
explicit forward_list(size_type Count);
forward_list(size_type Count, const Type& Val);
forward_list(size_type Count, const Type& Val, const Allocator& Al);
forward_list(const forward_list& Right);
forward_list(const forward_list& Right, const Allocator& Al);
forward_list(forward_list&& Right);
forward_list(forward_list&& Right, const Allocator& Al);
forward_list(initializer_list<Type> IList, const Alloc& Al);
template <class InputIterator>
    forward_list(InputIterator First, InputIterator Last);
template <class InputIterator>
    forward_list(InputIterator First, InputIterator Last, const Allocator& Al);
```

Parameters

Al
The allocator class to use with this object.

Count
The number of elements in the list constructed.

Val
The value of the elements in the list constructed.

Right
The list of which the constructed list is to be a copy.

First
The position of the first element in the range of elements to be copied.

Last
The position of the first element beyond the range of elements to be copied.

IList
The initializer_list to copy.

Remarks

All constructors store an allocator and initialize the controlled sequence. The allocator object is the argument *Al*, if present. For the copy constructor, it is `right.get_allocator()`. Otherwise, it is `Allocator()`.

The first two constructors specify an empty initial controlled sequence.

The third constructor specifies a repetition of *Count* elements of value `Type()`.

The fourth and fifth constructors specify a repetition of *Count* elements of value *Val*.

The sixth constructor specifies a copy of the sequence controlled by *Right*. If `InputIterator` is an integer type, the next two constructors specify a repetition of `(size_type)First` elements of value `(Type)Last`. Otherwise, the next two constructors specify the sequence `[First, Last)`.

The ninth and tenth constructors are the same as the sixth, but with an rvalue reference.

The last constructor specifies the initial controlled sequence with an object of class `initializer_list<Type>`.

front

Returns a reference to the first element in a forward list.

C++

```
reference front();
const_reference front() const;
```

Return Value

A reference to the first element of the controlled sequence, which must be non-empty.

get_allocator

Returns a copy of the allocator object used to construct a forward list.

C++

```
allocator_type get_allocator() const;
```

Return Value

The stored allocator object.

insert_after

Adds elements to the forward list after a specified position.

C++

```
iterator insert_after(const_iterator Where, const Type& Val);
void insert_after(const_iterator Where, size_type Count, const Type& Val);
void insert_after(const iterator Where, initializer_list<Type> IList);
iterator insert_after(const_iterator Where, Type&& Val);
template <class InputIterator>
    void insert_after(const_iterator Where, InputIterator First, InputIterator Last);
```

Parameters

Where
The position in the target forward list where the first element is inserted.

Count
The number of elements to insert.

First
The beginning of the insertion range.

Last
The end of the insertion range.

Val
The element added to the forward list.

IList
The initializer_list to insert.

Return Value

An iterator that designates the newly inserted element (first and last member functions only).

Remarks

Each of the member functions inserts—just after the element pointed to by *Where* in the controlled sequence—a sequence that' specified by the remaining operands.

The first member function inserts an element that has value *Val* and returns an iterator that designates the newly inserted element.

The second member function inserts a repetition of *Count* elements of value *Val*.

If `InputIterator` is an integer type, the third member function behaves the same as `insert(it, (size_type)First, (Type)Last)`. Otherwise, it inserts the sequence `[First, Last)`, which must not overlap the initial controlled sequence.

The fourth member function inserts the sequence that's specified by an object of class `initializer_list<Type>`.

The last member function is the same as the first, but with an rvalue reference.

Inserting `N` elements causes `N` constructor calls. Reallocation occurs, but no iterators or references become invalid.

If an exception is thrown during the insertion of one or more elements, the container is left unaltered and the exception is rethrown.

iterator

A type that provides an iterator for the forward list.

C++

```
typedef implementation-defined iterator;
```

Remarks

`iterator` describes an object that can serve as a forward iterator for the controlled sequence. It is described here as a synonym for an implementation-defined type.

max_size

Returns the maximum length of a forward list.

C++

```
size_type max_size() const;
```

Return Value

The length of the longest sequence that the object can control.

merge

Combines two sorted sequences into a single sorted sequence in linear time. Removes the elements from the argument list, and inserts them into this `forward_list`. The two lists should be sorted by the same compare function object before the call to `merge`. The combined list will be sorted by that compare function object.

C++

```
void merge(forward_list& right);
template <class Predicate>
    void merge(forward_list& right, Predicate comp);
```

Parameters

right
The forward list to merge from.

comp
The compare function object that is used to sort elements.

Remarks

`forward_list::merge` removes the elements from the `forward_list` `right`, and inserts them into this `forward_list`. Both sequences must be ordered by the same predicate, described below. The combined sequence is also ordered by that compare function object.

For the iterators `Pi` and `Pj` designating elements at positions `i` and `j`, the first member function imposes the order `!(*Pj < *Pi)` whenever `i < j`. (The elements are sorted in `ascending` order.) The second member function imposes the order `! comp(*Pj, *Pi)` whenever `i < j`.

No pairs of elements in the original controlled sequence are reversed in the resulting controlled sequence. If a pair of elements in the resulting controlled sequence compares equal (`!(*Pi < *Pj) && !(*Pj < *Pi)`), an element from the original controlled sequence appears before an element from the sequence controlled by `right`.

An exception occurs only if `comp` throws an exception. In that case, the controlled sequence is left in unspecified order and the exception is rethrown.

operator=

Replaces the elements of the forward list with a copy of another forward list.

C++

```
forward_list& operator=(const forward_list& right);
forward_list& operator=(initializer_list<Type> IList);
forward_list& operator=(forward_list&& right);
```

Parameters

right
The forward list being copied into the forward list.

IList
A brace-enclosed initializer list, which behaves just like a sequence of elements of type `Type`.

Remarks

The first member operator replaces the controlled sequence with a copy of the sequence controlled by *right*.

The second member operator replaces the controlled sequence from an object of class `initializer_list<Type>`.

The third member operator is the same as the first, but with an rvalue reference.

pointer

A type that provides a pointer to an element in the forward list.

C++

```
typedef typename Allocator::pointer pointer;
```

pop_front

Deletes the element at the beginning of a forward list.

C++

```
void pop_front();
```

Remarks

The first element of the forward list must be non-empty.

The member function never throws an exception.

push_front

Adds an element to the beginning of a forward list.

C++

```
void push_front(const Type& val);
void push_front(Type&& val);
```

Parameters

val
The element added to the beginning of the forward list.

Remarks

If an exception is thrown, the container is left unaltered and the exception is rethrown.

reference

A type that provides a reference to an element in the forward list.

C++

```
typedef typename Allocator::reference reference;
```

remove

Erases elements in a forward list that matches a specified value.

C++

```
void remove(const Type& val);
```

Parameters

val
The value which, if held by an element, will result in that element's removal from the list.

Remarks

The member function removes from the controlled sequence all elements, designated by the iterator `P`, for which `*P == val`.

The member function never throws an exception.

remove_if

Erases elements from a forward list for which a specified predicate is satisfied.

C++

```
template <class Predicate>
    void remove_if(Predicate pred);
```

Parameters

pred
The unary predicate which, if satisfied by an element, results in the deletion of that element from the list.

Remarks

The member function removes from the controlled sequence all elements, designated by the iterator `P`, for which `pred(*P)` is true.

An exception occurs only if *pred* throws an exception. In that case, the controlled sequence is left in an unspecified state and the exception is rethrown.

resize

Specifies a new size for a forward list.

C++

```
void resize(size_type _Newsize);
void resize(size_type _Newsize, const Type& val);
```

Parameters

_Newsize
The number of elements in the resized forward list.

val
The value to use for padding.

Remarks

The member functions both ensure that the number of elements in the list henceforth is *_Newsize*. If it must make the controlled sequence longer, the first member function appends elements with value `Type()`, while the second member function appends elements with value *val*. To make the controlled sequence shorter, both member functions effectively call `erase_after(begin() + _Newsize - 1, end())`.

reverse

Reverses the order in which the elements occur in a forward list.

C++

```
void reverse();
```

size_type

A type that represents the unsigned distance between two elements.

C++

```
typedef typename Allocator::size_type size_type;
```

Remarks

The unsigned integer type describes an object that can represent the length of any controlled sequence.

sort

Arranges the elements in ascending order or with an order specified by a predicate.

C++

```
void sort();
template <class Predicate> void sort(Predicate pred);
```

Parameters

pred
The ordering predicate.

Remarks

Both member functions order the elements in the controlled sequence by a predicate, described below.

For the iterators `Pi` and `Pj` designating elements at positions `i` and `j`, the first member function imposes the order `!(*Pj < *Pi)` whenever `i < j`. (The elements are sorted in `ascending` order.) The member template function imposes the order `! pred(*Pj, *Pi)` whenever `i < j`. No ordered pairs of elements in the original controlled sequence are reversed in the resulting controlled sequence. (The sort is stable.)

An exception occurs only if *pred* throws an exception. In that case, the controlled sequence is left in unspecified order and the exception is rethrown.

splice_after

Removes elements from a source forward_list and inserts them into a destination forward_list.

C++

```
// insert the entire source forward_list
void splice_after(const_iterator Where, forward_list& Source);
void splice_after(const_iterator Where, forward_list&& Source);

// insert one element of the source forward_list
void splice_after(const_iterator Where, forward_list& Source, const_iterator Iter);
void splice_after(const_iterator Where, forward_list&& Source, const_iterator Iter);

// insert a range of elements from the source forward_list
void splice_after(const_iterator Where, forward_list& Source,
    const_iterator First, const_iterator Last);

void splice_after(const_iterator Where, forward_list&& Source,
    const_iterator First, const_iterator Last);
```

Parameters

Where
The position in the destination forward_list after which to insert.

Source
The source forward_list that is to be inserted into the destination forward_list.

Iter
The element to be inserted from the source forward_list.

First
The first element in the range to be inserted from source forward_list.

Last
The first position beyond the range to be inserted from the source forward_list.

Remarks

The first pair of member functions inserts the sequence controlled by *Source* just after the element in the controlled sequence pointed to by *Where*. It also removes all elements from *Source*. (`&Source` must not equal **`this`**.)

The second pair of member functions removes the element just after *Iter* in the sequence controlled by *Source* and inserts it just after the element in the controlled sequence pointed to by *Where*. (If `Where == Iter || Where == ++Iter`, no change occurs.)

The third pair of member functions (ranged splice) inserts the subrange designated by `(First, Last)` from the sequence controlled by *Source* just after the element in the controlled sequence pointed to by *Where*. It also removes the original subrange from the sequence controlled by *Source*. (If `&Source == this`, the range `(First, Last)` must not include the element pointed to by *Where*.)

If the ranged splice inserts `N` elements, and `&Source != this`, an object of class iterator is incremented `N` times.

No iterators, pointers, or references that designate spliced elements become invalid.

Example

C++

```
// forward_list_splice_after.cpp
// compile with: /EHsc /W4
#include <forward_list>
#include <iostream>

using namespace std;

template <typename S> void print(const S& s) {
    for (const auto& p : s) {
        cout << "(" << p << ") ";
    }

    cout << endl;
}

int main()
{
    forward_list<int> c1{ 10, 11 };
    forward_list<int> c2{ 20, 21, 22 };
    forward_list<int> c3{ 30, 31 };
    forward_list<int> c4{ 40, 41, 42, 43 };

    forward_list<int>::iterator where_iter;
    forward_list<int>::iterator first_iter;
    forward_list<int>::iterator last_iter;

    cout << "Beginning state of lists:" << endl;
    cout << "c1 = ";
    print(c1);
    cout << "c2 = ";
    print(c2);
    cout << "c3 = ";
    print(c3);
    cout << "c4 = ";
    print(c4);

    where_iter = c2.begin();
    ++where_iter; // start at second element
    c2.splice_after(where_iter, c1);
    cout << "After splicing c1 into c2:" << endl;
    cout << "c1 = ";
    print(c1);
    cout << "c2 = ";
```

```
    print(c2);

    first_iter = c3.begin();
    c2.splice_after(where_iter, c3, first_iter);
    cout << "After splicing the first element of c3 into c2:" << endl;
    cout << "c3 = ";
    print(c3);
    cout << "c2 = ";
    print(c2);

    first_iter = c4.begin();
    last_iter = c4.end();
    // set up to get the middle elements
    ++first_iter;
    c2.splice_after(where_iter, c4, first_iter, last_iter);
    cout << "After splicing a range of c4 into c2:" << endl;
    cout << "c4 = ";
    print(c4);
    cout << "c2 = ";
    print(c2);
}
```

Output

```
Beginning state of lists:c1 = (10) (11)c2 = (20) (21) (22)c3 = (30) (31)c4 = (40) (41) (42)
(43)After splicing c1 into c2:c1 =c2 = (20) (21) (10) (11) (22)After splicing the first
element of c3 into c2:c3 = (30)c2 = (20) (21) (31) (10) (11) (22)After splicing a range of
c4 into c2:c4 = (40) (41)c2 = (20) (21) (42) (43) (31) (10) (11) (22)
```

swap

Exchanges the elements of two forward lists.

C++

```
void swap(forward_list& right);
```

Parameters

right
The forward list providing the elements to be exchanged.

Remarks

The member function swaps the controlled sequences between **`*this`** and *right*. If `get_allocator() == right.get_allocator()`, it does so in constant time, it throws no exceptions, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

unique

Eliminates all but the first element from every consecutive group of equal elements.

C++

```
void unique();
template <class BinaryPredicate>
void unique(BinaryPredicate comp);
```

Parameters

comp
The binary predicate used to compare successive elements.

Remarks

Keeps the first of each unique element, and removes the rest. The elements must be sorted so that elements of equal value are adjacent in the list.

The first member function removes from the controlled sequence every element that compares equal to its preceding element. For the iterators `Pi` and `Pj` designating elements at positions `i` and `j`, the second member function removes every element for which `i + 1 == j && comp(*Pi, *Pj)`.

For a controlled sequence of length `N` (> 0), the predicate `comp(*Pi, *Pj)` is evaluated `N - 1` times.

An exception occurs only if `comp` throws an exception. In that case, the controlled sequence is left in an unspecified state and the exception is rethrown.

value_type

A type that represents the type of element stored in a forward list.

C++

```
typedef typename Allocator::value_type value_type;
```

Remarks

The type is a synonym for the template parameter `Type`.