
8.8. Special Member Functions for Lists and Forward Lists

8.8.1. Special Member Functions for Lists (and Forward Lists)

```
void list::remove (const T& value)
void list::remove_if (UnaryPredicate op)
```

- `remove()` removes all elements with value `value`.
- `remove_if()` removes all elements for which the unary predicate

  $op(elem)$

  yields `true`.
- Note that *op* should not change its state during a function call. See Section 10.1.4, page 483, for details.
- Both call the destructors of the removed elements.
- The order of the remaining arguments remains stable.
- This is the special version of the `remove()` algorithm, which is discussed in Section 11.7.1, page 575.
- `T` is the type of the container elements.
- For further details and examples, see Section 7.5.2, page 294.
- The functions may throw only if the comparison of the elements may throw.
- Provided by list, forward list.

```
void list::unique ()
void list::unique (BinaryPredicate op)
```

- Remove subsequent duplicates of (forward) list elements so that the value of each element is different from that of the following element.
- The first form removes all elements for which the previous values are equal.
- The second form removes all elements that follow an element *e* and for which the binary predicate

  $op(elem, e)$

  yields `true`. In other words, the predicate is not used to compare an element with its predecessor; the element is compared with the previous element that was not removed.
- Note that *op* should not change its state during a function call. See Section 10.1.4, page 483, for details.
- Both call the destructors of the removed elements.
- These are the special versions of the `unique()` algorithms (see Section 11.7.2, page 578).
- The functions do not throw if the comparisons of the elements do not throw.
- Provided by list, forward list.

Click here to view code image

```
void list::splice (const_iterator pos, list& source)
void list::splice (const_iterator pos, list&& source)
```

- Move all elements of the list *source* into `*this` and insert them before the position of iterator *pos*.
- After the call, *source* is empty.
- If *source* and `*this` are identical, the behavior is undefined. Thus, the caller must ensure that *source* is a different list. To move elements inside the same list, you must use the following forms of `splice()`.
- The caller must ensure that *pos* is a valid position of `*this`; otherwise, the behavior is undefined.
- Pointers, iterators, and references to members of *source* remain valid. Thus, they refer to elements in `this` afterward.
- This function does not throw.
- The second form is available since C++11. Before C++11, type `iterator` was used instead of `const_iterator`.
- Provided by list.

Click here to view code image

```
void list::splice (const_iterator pos, list& source, const_iterator sourcePos)
void list::splice (const_iterator pos, list&& source, const_iterator sourcePos)
```

- Move the element at the position *sourcePos* of the list *source* into `*this` and insert it before the position of iterator *pos*.
- *source* and `*this` may be identical. In this case, the element is moved inside the list.
- If *source* is a different list, it contains one element less after the operation.
- The caller must ensure that *pos* is a valid position of `*this`, that *sourcePos* is a valid iterator of *source*, and that *sourcePos* is not *source* `.end()`; otherwise, the behavior is undefined.
- Pointers, iterators, and references to members of *source* remain valid. Thus, they belong to `this` afterward.
- Pointers, iterators, and references to members of *source* remain valid. However, for the spliced element, they refer to an element in `this` afterward.
- This function does not throw.
- The second form is available since C++11. Before C++11, type `iterator` was used instead of `const_iterator`.
- Provided by list.

Click here to view code image

```
void list::splice (const_iterator pos, list& source,
                   const_iterator sourceBeg, const_iterator sourceEnd)
void list::splice (const_iterator pos, list&& source,
                   const_iterator sourceBeg, const_iterator sourceEnd)
```

- Move the elements of the range `[` *sourceBeg,sourceEnd* `)` of the list *source* to `*this` and insert them before the position of iterator *pos*.
- *source* and `*this` may be identical. In this case, *pos* must not be part of the moved range, and the elements are moved inside the list.
- If *source* is a different list, it contains fewer elements after the operation.
- The caller must ensure that *pos* is a valid position of `*this` and that *sourceBeg* and *sourceEnd* define a valid range that is part of *source*; otherwise, the behavior is undefined.
- Pointers, iterators, and references to members of *source* remain valid. However, for the spliced elements, they refer to elements in `this` afterward.
- This function does not throw.
- The second form is available since C++11. Before C++11, type `iterator` was used instead of `const_iterator`.
- Provided by list.

```
void list::sort ()
void list::sort (CompFunc cmpPred)
```

- Sort the elements.
- The first form sorts all elements with operator `<`.
- The second form sorts all elements by calling *cmpPred* to compare two elements:

$$op(elem1, elem2)$$

- The order of elements that have an equal value remains stable unless an exception is thrown.
- These are the special versions of the `sort()` and `stable_sort()` algorithms (see Section 11.9.1, page 596).
- Provided by list, forward list.

```
void list::merge (list& source)
void list::merge (list&& source)
void list::merge (list& source, CompFunc cmpPred)
void list::merge (list&& source, CompFunc cmpPred)
```

- Merge all elements of the (forward) list *source* into `*this`.
- After the call, *source* is empty.
- The first two forms use operator `<` as the sorting criterion.
- The last two forms use *cmpPred* as the optional sorting criterion and to compare two elements:

cmpPred(*elem*,*sourceElem*)

- The order of elements that have an equivalent value remains stable.
- If `*this` and *source* are sorted on entry according to the sorting criterion `<` or *cmpPred*, the resulting (forward) list is also sorted and equivalent elements of `*this` precede equivalent elements of *source*. Strictly speaking, the standard requires that both (forward) lists be sorted on entry. In practice, however, merging is also possible for unsorted lists. However, you should check this before you rely on it.
- This is the special version of the `merge()` algorithm (see Section 11.10.2, page 614).
- If the comparisons of the elements do not throw, the functions either succeed or have no effect.
- Provided by list, forward list.

`void` **`list::reverse`** `()`

- Reverses the order of the elements in a (forward) list.
- This is the special version of the `reverse()` algorithm (see Section 11.8.1, page 583).
- This function does not throw.
- Provided by list, forward list.

8.8.2. Special Member Functions for Forward Lists Only

`iterator` **`forwardlist::before_begin`** `()`
`const_iterator` **`forwardlist::before_begin`** `() const`
`const_iterator` **`forwardlist::cbefore_begin`** `() const`

- Return an iterator for the the position before the first element.
- Because you can't iterate backward, this member function allows you to yield the position to insert a new or delete the first element.
- Provided by forward list.

Click here to view code image

`iterator` *forwardlist*`::`**`insert_after`** `(const_iterator` *pos*`, const T&` *value*`)`
`iterator` *forwardlist*`::`**`insert_after`** `(const_iterator` *pos*`, T&&` *value*`)`

- Insert *value* right after the position of iterator *pos*.
- The first form copies *value*.
- The second form moves *value* to the container, so the state of *value* is undefined afterward.
- Return the position of the new element.
- The functions either succeed or have no effect.
- Passing `end()` or `cend()` of a container as *pos* results in undefined behavior.
- Provided by forward list.

`iterator` **`forwardlist::emplace_after`** `(const_iterator` *pos*`,` *args*`)`

- Inserts a new element initialized by *args* right after the position of iterator *pos*.
- Returns the position of the new element.
- The function either succeeds or has no effect.[2]

[2] Currently, the standard does not say this, which likely is a defect.

- Passing `end()` or `cend()` of a container as *pos* results in undefined behavior.
- Provided by forward list.

Click here to view code image

`iterator` *forwardlist*`::`**`insert_after`** `(const_iterator` *pos*`,`
`size_type` *num*`, const T&` *value*`)`

- Inserts *num* copies of *value* right behind the position of iterator *pos*.
- Returns the position of the last inserted element or *pos* if *num* `==0` .
- The function either succeeds or has no effect.
- Passing `end()` or `cend()` of a container as *pos* results in undefined behavior.
- Provided by forward list.

`iterator` **`forwardlist::insert_after`** `(const_iterator` *pos*`,` *initializer-list*`)`

- Inserts copies of the elements of *initializer-list* right after the position of iterator *pos*.
- Returns the position of the last inserted element or *pos* if *initializer-list* is empty.
- The function either succeeds or has no effect.
- Passing `end()` or `cend()` of a container as *pos* results in undefined behavior.
- Available since C++11. forward list.

Click here to view code image

```
iterator forwardlist::insert_after (const_iterator pos,
                                    InputIterator beg, InputIterator end)
```

- Inserts copies of all elements of the range `[` *beg* `,` *end* `)` right after the position of iterator *pos*.
- Returns the position of the last inserted element or *pos* if *beg* `==` *end*.
- This function is a member template (see Section 3.2, page 34). Thus, the elements of the source range may have any type convertible into the element type of the container.
- The function either succeeds or has no effect.
- Passing `end()` or `cend()` of a container as *pos* results in undefined behavior.
- Provided by forward list.

```
iterator forwardlist::erase_after (const_iterator pos)
```

- Removes the element right after the position of iterator *pos*.
- Returns the position of the following element (or `end()`).
- Calls the destructor of the removed element.
- Iterators and references to other elements remain valid.
- The caller must ensure that the iterator *pos* is valid, which excludes to pass `end()` and the position before `end()` .
- The function does not throw.
- Passing `end()` or `cend()` of a container as *pos* results in undefined behavior.
- Provided by forward list.

```
void forwardlist::erase_after (const_iterator beg, const_iterator end)
```

- Removes the elements of the range `[` *beg* `,` *end* `)` . Note that this is *not* a half-open range, because it excludes both *beg* and *end*. For example:

```
coll.erase_after(coll.before_begin(),coll.end()); // OK: erases all elements
```

- Returns *end*.
- Calls the destructors of the removed elements.
- The caller must ensure that *beg* and *end* define a valid range that is part of the container.
- The function does not throw.
- Iterators and references to other elements remain valid.
- Provided by forward list.

Click here to view code image

```
void forwardlist::splice_after (const_iterator pos, forwardlist& source)
void forwardlist::splice_after (const_iterator pos, forwardlist&& source)
```

- Move all elements of *source* into `*this` and insert them at the position right after iterator *pos*.
- After the call, *source* is empty.
- If *source* and `*this` are identical, the behavior is undefined. Thus, the caller must ensure that *source* is a different list. To move elements inside the same list, you must use the following forms of `splice_after()` .
- The caller must ensure that *pos* is a valid position of `*this` ; otherwise, the behavior is undefined.
- Pointers, iterators, and references to members of *source* remain valid. Thus, they refer to elements in `this` afterward.
- This function does not throw.
- Passing `end()` or `cend()` of a container as *pos* results in undefined behavior.
- Provided by forward list.

Click here to view code image

```
void forwardlist::splice_after (const_iterator pos,
                                forwardlist& source, const_iterator sourcePos)
void forwardlist::splice_after (const_iterator pos,
                                forwardlist&& source, const_iterator sourcePos)
```

- Move the element right after the position *sourcePos* of the list *source* into `*this` and insert it at the position right after iterator *pos*.
- *source* and `*this` may be identical. In this case, the element is moved inside the list.
- If *source* is a different list, it contains one element less after the operation.
- The caller must ensure that *pos* is a valid position of `*this`, that *sourcePos* is a valid iterator of *source*, and that *sourcePos* is not *source* `.end()`; otherwise, the behavior is undefined.
- Pointers, iterators, and references to members of *source* remain valid. However, for the spliced element, they refer to an element in `this` afterward.
- This function does not throw.
- Passing `end()` or `cend()` of a container as *pos* results in undefined behavior.
- Provided by forward list.

Click here to view code image

```
void forwardlist::splice_after (const_iterator pos, forwardlist& source,
                                const_iterator sourceBeg,
                                const_iterator sourceEnd)
void forwardlist::splice_after (const_iterator pos, forwardlist&& source,
                                const_iterator sourceBeg,
                                const_iterator sourceEnd)
```

Move the elements of the range `(` *sourceBeg,sourceEnd* `)` of the list *source* to `*this` and insert them at the position right after iterator *pos*. Note that the last two arguments are *not* a half-open range, because it excludes both *beg* and *end*. For example, the following call moves all elements of `coll2` to the beginning of `coll`:

```
coll.splice_after(coll.before_begin(), coll2,
                  coll2.before_begin(), coll2.end());
```

- *source* and `*this` may be identical. In this case, *pos* must not be part of the moved range, and the elements are moved inside the list.
- If *source* is a different list, it contains fewer elements after the operation.
- The caller must ensure that *pos* is a valid position of `*this` and that *sourceBeg* and *sourceEnd* define a valid range that is part of *source*; otherwise, the behavior is undefined.
- Pointers, iterators, and references to members of *source* remain valid. However, for the spliced elements, they refer to elements in `this` afterward.
- This function does not throw.
- Passing `end()` or `cend()` of a container as *pos* results in undefined behavior.
- Provided by forward list.