
7.6. Forward Lists

A forward list (an instance of the container class `forward_list<>`), which was introduced with C++11, manages its elements as a singly linked list (Figure 7.8). As usual, the C++ standard library does not specify the kind of the implementation, but it follows from the forward list's name, constraints, and specifications.

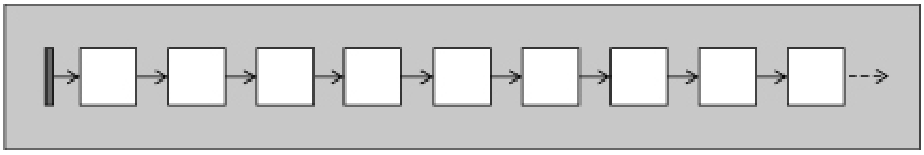

Figure 7.8. Structure of a Forward List

To use a forward list, you must include the header file `<forward_list>`:

```
#include <forward_list>
```

There, the type is defined as a class template inside namespace `std`:

```
namespace std {
    template <typename T,
              typename Allocator = allocator<T> >
    class  forward_list;
}
```

The elements of a forward list may have any type `T`. The optional second template parameter defines the memory model (see Chapter 19). The default memory model is the model `allocator`, which is provided by the C++ standard library.

7.6.1. Abilities of Forward Lists

Conceptually, a forward list is a list (object of class `list<>`) restricted such that it is not able to iterate backward. It provides no functionality that is not also provided by lists. As benefits, it uses less memory and provides slightly better runtime behavior. The standard states: "*It is intended that* `forward_list` *have zero space or time overhead relative to a hand-written C-style singly linked list. Features that would conflict with that goal have been omitted.*"

Forward lists have the following limitations compared to lists:

- A forward list provides only forward iterators, not bidirectional iterators. As a consequence, no reverse iterator support is provided, which means that types, such as `reverse_iterator`, and member functions, such as `rbegin()`, `rend()`, `crbegin()`, and `crend()`, are not provided.
- A forward list does not provide a `size()` member function. This is a consequence of omitting features that create time or space overhead relative to a handwritten singly linked list.
- The anchor of a forward list has no pointer to the last element. For this reason, a forward list does not provide the special member functions to deal with the last element, `back()`, `push_back()`, and `pop_back()`.
- For all member functions that modify forward lists in a way that elements are inserted or deleted at a specific position, special versions for forward lists are provided. The reason is that you have to pass the position of the element *before* the first element that gets manipulated, because there you have to assign a new successor element. Because you can't navigate backwards (at least not in constant time), for all these member functions you have to pass the position of the preceding element. Because of this difference, these member functions have a `_after` suffix in their name. For example, instead of `insert()`, `insert_after()` is provided, which inserts new elements after the element passed as first argument; that is, it *appends* an element at that position.
- For this reason, forward lists provide `before_begin()` and `cbefore_begin()`, which yield the position of a virtual element before the first element (technically speaking, the anchor of the linked list), which can be used to let built-in algorithms ending with `_after` exchange even the first element.

The decision not to provide `size()` might be especially surprising because `size()` is one of the operations required for all STL containers (see Section 7.1.2, page 254). Here, you can see the consequences of the design goal to have "zero space or time overhead relative to a hand-written C-style singly linked list." The alternative would have been either to compute the size each time `size()` is called, which would have linear complexity, or to provide an additional field in the `forward_list` object for the size, which is updated with each and every operation that changes the number of elements. As the design paper for the forward list, [N2543:FwdList], mentions: "It's a cost that all users would have to pay for, whether they need this feature or not." So, if you need the size, either track it outside the `forward_list` or use a `list` instead.

Other than these differences, forward lists behave just like lists:

- A forward list does not provide random access. For example, to access the fifth element, you must navigate the first four elements, following the chain of links. Thus, using a forward list to access an arbitrary element is slow.
- Inserting and removing elements is fast at each position, if you are there. You can always insert and delete an element in constant time, because no other elements have to be moved. Internally, only some pointer values are manipulated.
- Inserting and deleting elements does not invalidate iterators, references, and pointers to other elements.
- A forward list supports exception handling in such a way that almost every operation succeeds or is a no-op. Thus, you can't get into an intermediate state in which only half of the operation is complete.
- Forward lists provide many special member functions for moving and removing elements. These member functions are faster versions of general algorithms, because they only redirect pointers rather than copy and move the values. However, when element positions are involved, you have to pass the preceding position, and the member function has the suffix `_after` in its name.

7.6.2. Forward List Operations

Create, Copy, and Destroy

The ability to create, copy, and destroy forward lists is the same as it is for every sequence container. See Table 7.27 for the forward list operations that do this. See also Section 7.1.2, page 254, for some remarks about possible initialization sources.

Table 7.27. Constructors and Destructor of Forward Lists

Operation	Effect
`forward list<Elem> c`	Default constructor; creates an empty forward list without any elements
`forward list<Elem> c(c2)`	Copy constructor; creates a new forward list as a copy of *c2* (all elements are copied)
`forward list<Elem> c = c2`	Copy constructor; creates a new forward list as a copy of *c2* (all elements are copied)
`forward list<Elem> c(rv)`	Move constructor; creates a new forward list, taking the contents of the rvalue *rv* (since C++11)
`forward list<Elem> c = rv`	Move constructor; creates a new forward list, taking the contents of the rvalue *rv* (since C++11)
`forward list<Elem> c(n)`	Creates a forward list with *n* elements created by the default constructor
`forward list<Elem> c(n,elem)`	Creates a forward list initialized with *n* copies of element *elem*
`forward list<Elem> c(beg,end)`	Creates a forward list initialized with the elements of the range [*beg*,*end*)
`forward list<Elem> c(initlist)`	Creates a forward list initialized with the elements of initializer list *initlist* (since C++11)
`forward list<Elem> c = initlist`	Creates a forward list initialized with the elements of initializer list *initlist* (since C++11)
`c.~forward list()`	Destroys all elements and frees the memory

Nonmodifying Operations

With one exception, forward lists provide the usual operations for size and comparisons: Forward lists provide no `size()` operation. The reason is that it is not possible to store or compute the current number of elements in constant time. And to make the fact visible that `size()` is an expensive operation, it is not provided. If you have to compute the number of elements, you can use `distance()` (see Section 9.3.3, page 445):

```
#include <forward_list>
#include <iterator>
std::forward_list<int> l;
…
std::cout << "l.size(): " << std::distance(l.begin(),l.end())
          << std::endl;
```

But note that `distance()` is a call with linear complexity here.

See Table 7.28 for a complete list of the nonmodifying operations of forward lists and Section 7.1.2, page 254, for more details about the other operations.

Table 7.28. Nonmodifying Operations of Forward Lists

Operation	Effect
`c.empty()`	Returns whether the container is empty
`c.max_size()`	Returns the maximum number of elements possible
c1 `==` *c2*	Returns whether *c1* is equal to *c2* (calls == for the elements)
c1 `!=` *c2*	Returns whether *c1* is not equal to *c2* (equivalent to `!(`*c1*`==`*c2*`)`)
c1 `<` *c2*	Returns whether *c1* is less than *c2*
c1 `>` *c2*	Returns whether *c1* is greater than *c2* (equivalent to *c2*`<`*c1*)
c1 `<=` *c2*	Returns whether *c1* is less than or equal to *c2* (equivalent to `!(`*c2*`<`*c1*`)`)
c1 `>=` *c2*	Returns whether *c1* is greater than or equal to *c2* (equivalent to `!(`*c1*`<`*c2*`)`)

Assignments

Forward lists also provide the usual assignment operations for sequence containers (Table 7.29). As usual, the insert operations match the constructors to provide different sources for initialization (see Section 7.1.2, page 254, for details).

Table 7.29. Assignment Operations of Forward Lists

Operation	Effect
`c = `*c2*	Assigns all elements of *c2* to `c`
`c = `*rv*	Move assigns all elements of the rvalue *rv* to `c` (since C++11)
`c = `*initlist*	Assigns all elements of the initializer list *initlist* to `c` (since C++11)
`c.assign(`*n*`,`*elem*`)`	Assigns *n* copies of element *elem*
`c.assign(`*beg*`,`*end*`)`	Assigns the elements of the range [*beg*,*end*)
`c.assign(`*initlist*`)`	Assigns all the elements of the initializer list *initlist*
c1`.swap(`*c2*`)`	Swaps the data of *c1* and *c2*
`swap(`*c1*`,`*c2*`)`	Swaps the data of *c1* and *c2*

Element Access

To access all elements of a forward list, you must use range-based `for` loops (see Section 3.1.4, page 17), specific operations, or iterators. In contrast to lists, the only element you can access directly is the first element, if any. For this reason, only `front()` is provided to access elements directly (Table 7.30).

Table 7.30. Direct Element Access of Forward Lists

Operation	Effect
`c.front()`	Returns the first element (*no* check whether a first element exists)

As usual, this operation does *not* check whether the container is empty. If the container is empty, calling `front()` results in undefined behavior. In addition, in multithreaded contexts, you need synchronization mechanisms to ensure that the container is not modified between the check for its size and the access to an element (see Section 18.4.3, page 984).

Iterator Functions

To access all elements of a forward list, you must use iterators. However, because you can traverse elements only in forward order, the iterators are forward iterators, and no support for reverse iterators is provided (Table 7.31).

Table 7.31. Iterator Operations of Forward Lists

Operation	Effect
`c.begin()`	Returns a forward iterator for the first element
`c.end()`	Returns a forward iterator for the position after the last element
`c.cbegin()`	Returns a constant forward iterator for the first element (since C++11)
`c.cend()`	Returns a constant forward iterator for the position after the last element (since C++11)
`c.before_begin()`	Returns a forward iterator for the position before the first element
`c.cbefore_begin()`	Returns a constant forward iterator for the position before the first element

Thus, you can't call algorithms that require bidirectional iterators or random-access iterators. All algorithms that manipulate the order of elements a lot, especially sorting algorithms, are in this category. However, for sorting the elements, forward lists provide the special member function `sort()` (see Section 8.8.1, page 422).

In addition, `before_begin()` and `cbefore_begin()` are provided to yield the position of a virtual element before the first element, which is necessary to be able to modify the next element even if the next element is the first element.

Note that `before_begin()` and `cbefore_begin()` do not represent a valid position of a forward list. Therefore, dereferencing these positions results in undefined behavior. Thus, using any ordinary algorithm with `before_begin()` as first argument passed results in a runtime error:

```
// RUNTIME ERROR: before_begin() is only valid with ..._after() operations
std::copy (fwlist.before_begin(), fwlist.end(),
           ...);
```

Besides copying and assignments, the only valid operations for return values of `before_begin()` are `++`, `==`, and `!=`.

Inserting and Removing Elements

Table 7.32 shows the operations provided for forward lists to insert and to remove elements. Due to the nature of lists in general and forward lists in particular, we have to discuss them in detail.

Table 7.32. Insert and Remove Operations of Forward Lists

Operation	Effect
`c.push_front(`*elem*`)`	Inserts a copy of *elem* at the beginning
`c.pop_front()`	Removes the first element (does not return it)
`c.insert_after(`*pos*`,`*elem*`)`	Inserts a copy of *elem* after iterator position *pos* and returns the position of the new element
`c.insert_after(`*pos*`,`*n*`,`*elem*`)`	Inserts *n* copies of *elem* after iterator position *pos* and returns the position of the first new element (or pos if there is no new element)
`c.insert_after(`*pos*`,`*beg*`,`*end*`)`	Inserts a copy of all elements of the range [*beg*,*end*) after iterator position *pos* and returns the position of the first new element (or *pos* if there is no new element)
`c.insert_after(`*pos*`,`*initlist*`)`	Inserts a copy of all elements of the initializer list *initlist* after iterator position *pos* and returns the position of the first new element (or *pos* if there is no new element)
`c.emplace_after(`*pos*`,`*args...*`)`	Inserts a new element initialized with *args* after iterator position *pos* and returns the position of the new element (since C++11)
`c.emplace_front(`*args...*`)`	Inserts a new element initialized with *args* at the beginning (returns nothing; since C++11)
`c.erase_after(`*pos*`)`	Removes the element after iterator position *pos* (returns nothing)
`c.erase_after(`*beg*`,`*end*`)`	Removes all elements of the range [*beg*,*end*) (returns nothing)
`c.remove(`*val*`)`	Removes all elements with value *val*
`c.remove_if(`*op*`)`	Removes all elements for which *op*`(`*elem*`)` yields `true`
`c.resize(`*num*`)`	Changes the number of elements to *num* (if `size()` grows new elements are created by their default constructor)
`c.resize(`*num*`,`*elem*`)`	Changes the number of elements to *num* (if `size()` grows new elements are copies of *elem*)
`c.clear()`	Removes all elements (empties the container)

First, the usual general hints apply:

- As usual when using the STL, you must ensure that the arguments are valid. Iterators must refer to valid positions, and the beginning of a range must have a position that is not behind the end.
- Inserting and removing is faster if, when working with multiple elements, you use a single call for all elements rather than multiple calls.

Then, as for lists, forward lists provide special implementations of the `remove()` algorithms (see Section 11.7.1, page 575). These member functions are faster than the `remove()` algorithms because they manipulate only internal pointers rather than the elements. For more details, see the description of these operations for lists in Section 7.5.2, page 294.

Note that for all the insert, emplace, and erase member functions provided for forward lists, you have a problem: They usually get a position of an element, where you have to insert a new element or must delete. But this requires a modification of the *preceding* element, because there the pointer to the next element has to get modified. For lists, you can just go backward to the previous element to manipulate it, but for forward lists, you can't. For this reason, the member functions behave differently than for lists, which is reflected by the name of the member functions. All end with `_after`, which means that they insert a new element *after* the one passed (i.e., they *append*)or delete the element *after* the element passed.

In combination with `before_begin()` to ensure that the first element is covered, when you use these member functions, a typical access of forward lists is as follows (see Figure 7.9):

```
std::forward_list<int> fwlist = { 1, 2, 3 };

// insert 77, 88, and 99 at the beginning:
fwlist.insert_after (fwlist.before_begin(),   // position
                     { 77, 88, 99 } );         // values
```

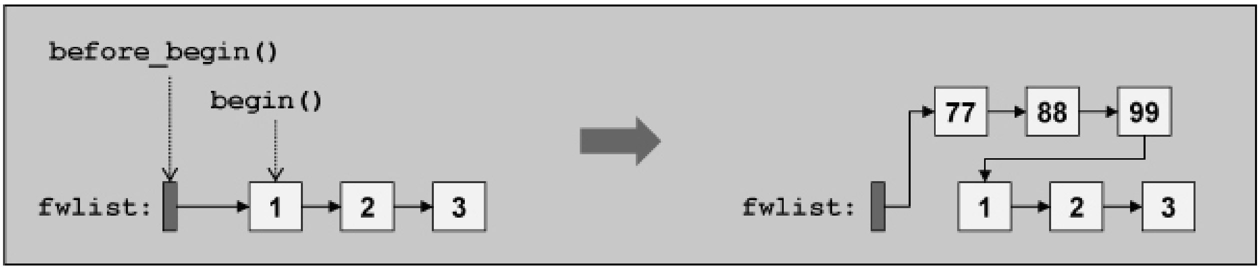

Figure 7.9. Inserting Elements at the Beginning of a Forward List

Note that calling an `_after` member function with `end()` or `cend()` results in undefined behavior because to append a new element at the end of a forward list, you have to pass the position of the last element (or `before_begin()` if none):

```
// RUNTIME ERROR: appending element after end is undefined behavior
fwlist.insert_after(fwlist.end(),9999);
```

Find and Remove or Insert

The drawbacks of having a singly linked list, where you can only traverse forward, get even worse when trying to find an element to insert or delete something there. The problem is that when you find the element, you are too far, because to insert or delete something there you have to manipulate the element before the element you are searching for. For this reason, you have to find an element by determining whether the *next* element fits a specific criterion. For example:

Click here to view code image

```
// cont/forwardlistfind1.cpp

#include <forward_list>
#include "print.hpp"
using namespace std;

int main()
{
    forward_list<int> list = { 1, 2, 3, 4, 5, 97, 98, 99 };

    // find the position before the first even element
    auto posBefore = list.before_begin();
    for (auto pos=list.begin(); pos!=list.end(); ++pos, ++posBefore) {
        if (*pos % 2 == 0) {
            break;   // element found
        }
    }

    // and insert a new element in front of the first even element
    list.insert_after(posBefore,42);
    PRINT_ELEMENTS(list);
}
```

Here, `pos` iterates over the list to find a specific element, whereas `posBefore` is always before `pos` to be able to return the position of the element before the element searched for (see Figure 7.10). So, the program has the following output:

```
1 42 2 3 4 5 97 98 99
```

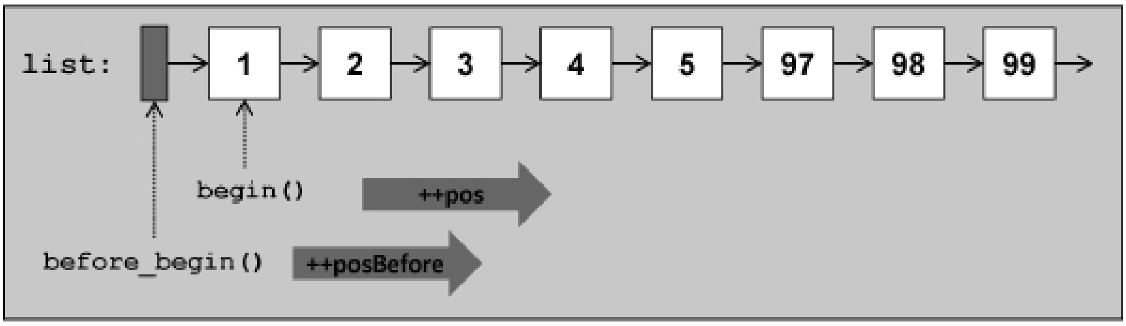

Figure 7.10. Searching for a Position to Insert or Delete

Alternatively, you can use the `next()` convenience function for iterators, which is available since C++11 (see Section 9.3.2, page 443):

```
#include <iterator>
...
auto posBefore = list.before_begin();
for ( ;  next(posBefore)!=list.end(); ++posBefore) {
    if (*next(posBefore) % 2 == 0) {
        break;   // element found
    }
}
```

If this is something you need more often, you might define your own algorithms to find a position before the element that has a specific value or fulfills a specific condition:

Click here to view code image

```
// cont/findbefore.hpp

template <typename InputIterator, typename Tp>
inline InputIterator
find_before (InputIterator first, InputIterator last, const Tp& val)
{
    if (first==last) {
        return first;
    }
    InputIterator next(first);
    ++next;
    while (next!=last && !(*next==val)) {
        ++next;
        ++first;
    }
    return first;
}

template <typename InputIterator, typename Pred>
inline InputIterator
find_before_if (InputIterator first, InputIterator last, Pred pred)
{
    if (first==last) {
        return first;
    }
    InputIterator next(first);
    ++next;
    while (next!=last && !pred(*next)) {
        ++next;
        ++first;
    }
    return first;
}
```

With these algorithms, you can use lambdas to find the corresponding position (for the complete example, see `cont/fwlistfind2.cpp`):

Click here to view code image

```
// find the position before the first even element
auto posBefore = find_before_if (list.before_begin(), list.end(),
                                 [] (int i) {
                                     return i%2==0;
                                 });
// and insert a new element in front of it
list.insert_after(posBefore,42);
```

You have to call `find_before_if()` with the position returned by `before_begin()`. Otherwise, you skip the first element. To avoid undefined behavior if you pass `begin()`, the algorithms first check whether the beginning of the range is equal to the end. A better approach would have been to let forward lists provide corresponding member functions, but this is, unfortunately, not the case.

Splice Functions and Functions to Change the Order of Elements

As with lists, forward lists have the advantage that you can remove and insert elements at any position in constant time. If you move elements from one container to another, this advantage doubles in that you need to redirect only some internal pointers. For this reason, forward lists provide almost the same member functions to splice lists or to change the order of elements. You can call these operations to move elements inside a single list or between two lists, provided that the lists have the same type. The only difference from lists is that `splice_after()` is provided instead of `splice()`, because the position of the element in front of the element where the splice applies is passed.

Table 7.33 lists these functions. They are covered in detail in Section 8.8, page 420. The following program demonstrates how to use the splice functions for forward lists. Here the first element with value `3` in the forward list `l1` is moved before the first element with value `99` in `l2`:

Click here to view code image

```
// cont/forwardlistsplice1.cpp

#include <forward_list>
#include "print.hpp"
using namespace std;

int main()
{
    forward_list<int> l1 = { 1, 2, 3, 4, 5 };
    forward_list<int> l2 = { 97, 98, 99 };

    //find 3 in l1
    auto pos1=l1.before_begin();
    for (auto pb1=l1.begin(); pb1 != l1.end(); ++pb1, ++pos1) {
        if (*pb1 == 3) {
            break;  //found
        }
    }

    //find 99 in l2
    auto pos2=l2.before_begin();
    for (auto pb2=l2.begin(); pb2 != l2.end(); ++pb2, ++pos2) {
        if (*pb2 == 99) {
            break;  //found
        }
    }

    //splice 3 from l1 to l2 before 99
    l1.splice_after(pos2, l2,     //destination
                    pos1);        //source

    PRINT_ELEMENTS(l1,"l1: ");
    PRINT_ELEMENTS(l2,"l2: ");
}
```

Table 7.33. Special Modifying Operations for Forward Lists

Operation	Effect
`c.unique()`	Removes duplicates of consecutive elements with the same value
`c.unique(op)`	Removes duplicates of consecutive elements, for which *op*() yields `true`
`c.splice_after(pos,c2)`	Moves all elements of *c2* to `c` right behind the iterator position *pos*
`c.splice_after(pos,c2,c2pos)`	Moves the element behind *c2pos* in *c2* right after *pos* of forward list `c` (`c` and *c2* may be identical)
`c.splice_after(pos,c2, c2beg,c2end)`	Moves all elements between *c2beg* and *c2end* (both not included) in *c2* right after *pos* of forward list `c` (`c` and *c2* may be identical)
`c.sort()`	Sorts all elements with operator <
`c.sort(op)`	Sorts all elements with *op*()
`c.merge(c2)`	Assuming that both containers contain the elements sorted, moves all elements of *c2* into `c` so that all elements are merged and still sorted
`c.merge(c2,op)`	Assuming that both containers contain the elements sorted by the sorting criterion *op*(), moves all elements of *c2* into `c` so that all elements are merged and still sorted according to *op*()
`c.reverse()`	Reverses the order of all elements

First, in `l1` we search for the position before the first element with value `3`. Then, in `l2` we search for the position before the first element with value `99`. Finally, with both positions `splice_after()` is called, which just modifies the internal pointers in the lists (see Figure 7.11).

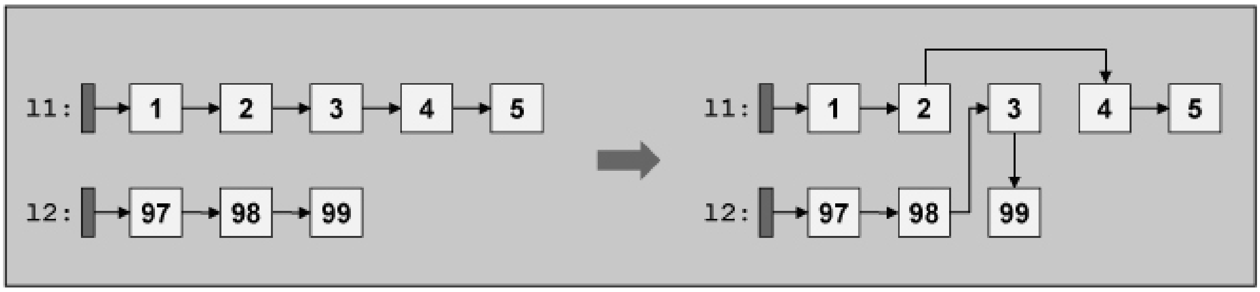

Figure 7.11. Effect of `splice_after()` with Forward Lists

Again, with our `find_before()` algorithms, the code looks a lot simpler:

```
// splice 3 from l1 to l2 before 99
l1.splice_after(l2.find_before(99), l2,    // destination
                l1.find_before(3));        // source
```

Note that source and destination for splice operations might be the same. Thus, you can move elements inside a forward list. However, note that calling `splice_after()` with `end()` results in undefined behavior, as all `_after` functions do with `end()`:

Click here to view code image

```
// RUNTIME ERROR: move first element to the end is not possible that way
fwlist.splice_after(fwlist.end(),        // destination position
                    fwlist,              // source list
                    fwlist.begin());     // source position
```

7.6.3. Exception Handling

Forward lists give the same guarantees that lists give regarding exception handling, provided that the corresponding member function is available. See Section 7.5.3, page 296, for details.

7.6.4. Examples of Using Forward Lists

The following example shows the use of the special member functions for forward lists:

[Click here to view code image](#)

```
// cont/forwardlist1.cpp

#include <forward_list>
#include <iostream>
#include <algorithm>
#include <iterator>
#include <string>
using namespace std;

void printLists (const string& s, const forward_list<int>& l1,
                                  const forward_list<int>& l2)
{
    cout << s << endl;
    cout << " list1: ";
    copy (l1.cbegin(), l1.cend(), ostream_iterator<int>(cout," "));
    cout << endl << " list2: ";
    copy (l2.cbegin(), l2.cend(), ostream_iterator<int>(cout," "));
    cout << endl;
}

int main()
{
    // create two forward lists
    forward_list<int> list1 = { 1, 2, 3, 4 };
    forward_list<int> list2 = { 77, 88, 99 };
    printLists ("initial:", list1, list2);

    // insert six new element at the beginning of list2
    list2.insert_after(list2.before_begin(),99);
    list2.push_front(10);
    list2.insert_after(list2.before_begin(), {10,11,12,13} );
    printLists ("6 new elems:", list1, list2);

    // insert all elements of list2 at the beginning of list1
    list1.insert_after(list1.before_begin(),
                       list2.begin(),list2.end());
    printLists ("list2 into list1:", list1, list2);

    // delete second element and elements after element with value 99
    list2.erase_after(list2.begin());
    list2.erase_after(find(list2.begin(),list2.end(),
                           99),
                      list2.end());
    printLists ("delete 2nd and after 99:", list1, list2);

    // sort list1, assign it to list2, and remove duplicates
    list1.sort();
    list2 = list1;
    list2.unique();
    printLists ("sorted and unique:", list1, list2);

    // merge both sorted lists into list1
    list1.merge(list2);
    printLists ("merged:", list1, list2);
}
```

The program has the following output:

```
initial:
 list1: 1 2 3 4
 list2: 77 88 99
6 new elems:
 list1: 1 2 3 4
 list2: 10 11 12 13 10 99 77 88 99
list2 into list1:
 list1: 10 11 12 13 10 99 77 88 99 1 2 3 4
 list2: 10 11 12 13 10 99 77 88 99
delete 2nd and after 99:
 list1: 10 11 12 13 10 99 77 88 99 1 2 3 4
 list2: 10 12 13 10 99
sorted and unique:
 list1: 1 2 3 4 10 10 11 12 13 77 88 99 99
 list2: 1 2 3 4 10 11 12 13 77 88 99
merged:
 list1: 1 1 2 2 3 3 4 4 10 10 10 11 11 12 12 13 13 77 77 88 88 99 99 99
 list2:
```

See Section 7.5.4, page 298, for a corresponding example using a list.