【C++】Day34

Class	C++
Date	@January 6, 2022
Material	
Series Number	
Summary	

【Ch9】Sequential Containers

9.3.4 Specialized forward_list Operations

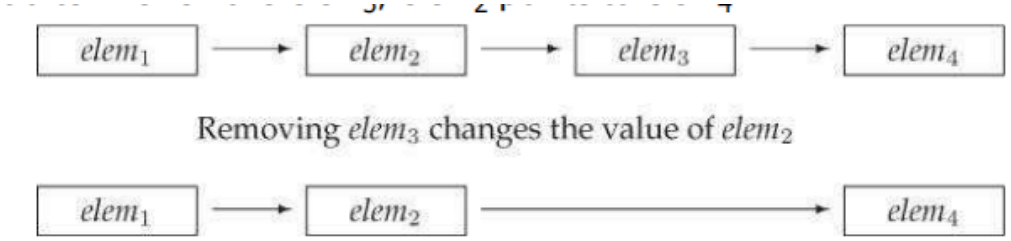

Figure 9.1. forward_list Specialized Operations

When we add or reomove an element in a linked list, the element before the one we added or removed has a different successor.

To add or remove an element, we need access to is predecessor in order to update that element's links. However, `forward_list` is a singly linked list, and thus havs no easy way to get to an element's predecessor.

For this reason, the operations to add or remove elements in a `forward_list` operate by changing the element after the given element.

Table 9.8. Operations to Insert or Remove Elements in a forward_list

`lst.before_begin()` `lst.cbefore_begin()`	Iterator denoting the nonexistent element just before the beginning of the list. This iterator may not be dereferenced. `cbefore_begin()` returns a `const_iterator`.

`lst.insert_after(p,t)`
`lst.insert_after(p,n,t)`
`lst.insert_after(p,b,e)`
`lst.insert_after(p,il)`
	Inserts element(s) *after* the one denoted by iterator `p`. `t` is an object, `n` is a count, `b` and `e` are iterators denoting a range (`b` and `e` must not refer to `lst`), and `il` is a braced list. Returns an iterator to the *last* inserted element. If the range is empty, returns `p`. Undefined if `p` is the off-the-end iterator.

`emplace_after(p, args)`	Uses *args* to construct an element after the one denoted by iterator `p`. Returns an iterator to the new element. Undefined if `p` is the off-the-end iterator.
`lst.erase_after(p)` `lst.erase_after(b,e)`	Removes the element *after* the one denoted by iterator `p` or the range of elements from the one *after* the iterator `b` up to but not including the one denoted by `e`. Returns an iterator to the element after the one deleted, or the off-the-end iterator if there is no such element. Undefined if `p` denotes the last element in `lst` or is the off-the-end iterator.

Because these operations behave differently from the operations on the other containers, `forward_list` does not define `insert`, `emplace`, or `erase`.

Instead, it defiens members named `insert_after`, `emplace_after`, and `erase_after`.

To support these operations, `forward_list` also defines `before_begin`, which returns an off-the-beginning iterator. This iterator lets us add or remove elements “after” the nonexistent element before the first one in the list.

For example, we will rewrite the code that removes the odd-valued elements from a list to use `forward_list`:

```
forward_list<int> flst = { 0, 1, 2, 3, 4, 5, 6, 7, 8 };
auto prev = flst.before_begin(); //the "off-the-start" iterator
auto curr = flst.begin(); //the start iterator
while(curr != flst.end()) {
  if(*curr % 2)
    curr = flst.erase_after(prev); //erase and move curr
  else {
    prev = curr++; //move the iterator to denote next, and move the curr iterator by one
  }
}
```

Exercise

Exercise 9.27: Write a program to find and remove the odd-valued elements in a `forward_list<int>`.

See 9_27.cpp for code

Exercise 9.28: Write a function that takes a `forward_list<string>` and two additional `string` arguments. The function should find the first `string` and insert the second immediately following the first. If the first `string` is not found, then insert the second `string` at the end of the list.

See 9_28.cpp for code