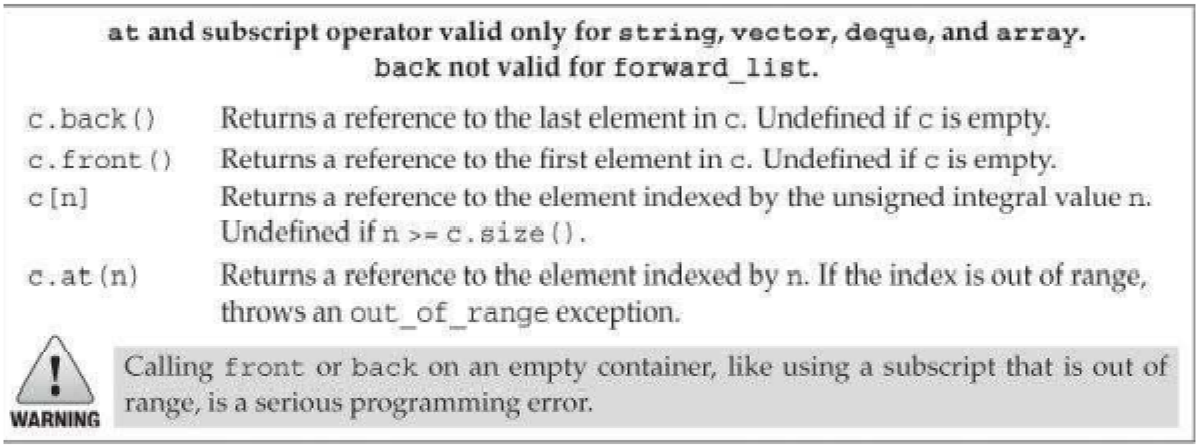
【C++】Day33(2)

Class	C++
Date	@January 5, 2022
Material	
Series Number	
Summary	

【Ch9】Sequential Container

9.3.2 Accessing Elements

The following table lists the operations we can use to access elements in a sequential container.



at and subscript operator valid only for string, vector, deque, and array.
back not valid for forward_list.

c.back()	Returns a reference to the last element in c. Undefined if c is empty.
c.front()	Returns a reference to the first element in c. Undefined if c is empty.
c[n]	Returns a reference to the element indexed by the unsigned integral value n. Undefined if n >= c.size().
c.at(n)	Returns a reference to the element indexed by n. If the index is out of range, throws an out_of_range exception.

WARNING

Calling front or back on an empty container, like using a subscript that is out of range, is a serious programming error.

These operations, `front()` and `back()`, return a reference to the first and last element, respectively:

```
//check that there are elements before dereferencing an iterator or calling front or back
if(!c.empty()) {
  //val1 and val2 are copies of the value of the first element in c
  auto val = *c.begin(), val2 = c.front();
  //val3 and val4 are copies of the last element in c;
  auto last = c.end();
  auto val3 = *(--last); //can't decrement forward_list ieterators
  auto val4 = c.back(); //not supported by forward_list
}
```

The important thing is that we check `c` is not empty before calling `front()` or `back()`.

The Access Members Return References

The members that access elements in a container(i.e. front, back, subscript, and at) return references.

```
if(!c.empty()) {
  c.front() = 42; //assigns 42 to the first element in c
  auto &v = c.back(); //get a reference to the last element
  v = 1024; //chagnes the element in c
  auto v2 = c.back(); //v2 is not a reference, it's a copy of c.backK()
  v2 = 0; //no change to the elemet in c
}
```

Subscripting and Safe Random Access

The subscript operator `[]` doesn’t check the validity of index.

If we want to ensure that our index is valid, we can use the `at` member instead. The at member acts like the subscript operator, but if the index is invalid, at throws an `out_of_range` exception:

```
vector<string> svec; //empty vector
cout << svec[0]; //run-time error: there are no elements in svec!
cout << svec.at(0); //throws an out_of_range exception
```

9.3.3 Erasing Elements

There are also ways to remove elements, listed below:

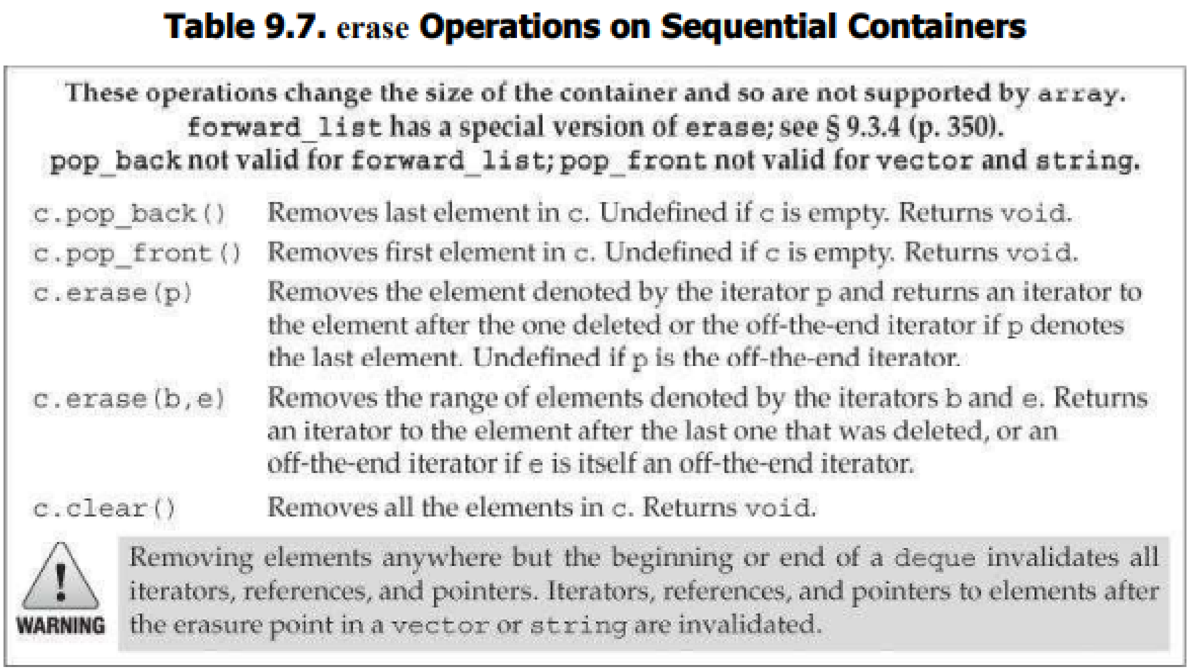


Table 9.7. erase Operations on Sequential Containers

These operations change the size of the container and so are not supported by `array`.
`forward_list` has a special version of `erase`; see § 9.3.4 (p. 350).
`pop_back` not valid for `forward_list`; `pop_front` not valid for `vector` and `string`.

`c.pop_back()`	Removes last element in `c`. Undefined if `c` is empty. Returns `void`.
`c.pop_front()`	Removes first element in `c`. Undefined if `c` is empty. Returns `void`.
`c.erase(p)`	Removes the element denoted by the iterator `p` and returns an iterator to the element after the one deleted or the off-the-end iterator if `p` denotes the last element. Undefined if `p` is the off-the-end iterator.
`c.erase(b,e)`	Removes the range of elements denoted by the iterators `b` and `e`. Returns an iterator to the element after the last one that was deleted, or an off-the-end iterator if `e` is itself an off-the-end iterator.
`c.clear()`	Removes all the elements in `c`. Returns `void`.

WARNING

Removing elements anywhere but the beginning or end of a `deque` invalidates all iterators, references, and pointers. Iterators, references, and pointers to elements after the erasure point in a `vector` or `string` are invalidated.

Warning: The members that remove elements do not check their arguments. The programmer must ensure that elements exist before removing them

The pop_front and pop_back Members

The `pop_front` and pop_back functions remove the first and last elements, respectively.

Like the element access members, we may not use a `pop` operation on an empty container.

These operations return void. If you need the value you are about to pop, you must store that value before doing the `pop`:

```
while(ilist.empty()) {
  process(ilist.front()); //do somethign with the current top of list
  ilist.opo_front();
}
```

Removing an Element from within the Container

The `erase` members remove elements at a specified point in the container.

We can delete a single element denoted by an iterator or a range of elements marked by a pair of iterators.

Both forms of erase return an iterator referring to the location after the element that was removed. That is, if j is the element following i, then `erase(i)` will return an iterator referring to j.

As an example, the following loop erases the odd elements in a list:

```
list<int> lst = {0, 1, 2, 3, 4, 5, 6, 7, 8};
auto it = lst.begin();
while(it != lst.end()) {
  if(*it % 2) //if the element is odd
    it = lst.erase(it); //erase this element
  else
    ++it;
}
```

Removing Multiple Elements

The iterator-pair version of erase let us delete a range of elements:

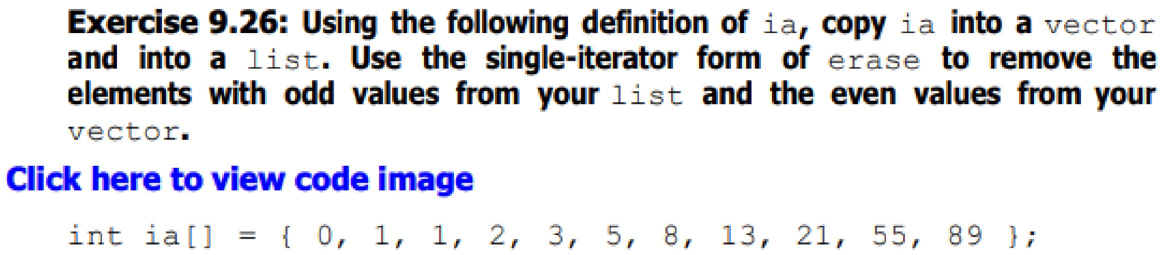
```
//delete the range of element between two iterators
//returns an iterator to the element jsut after the last removed element
elem1 = slist.erase(elem1, elem2); //after the call elem1 == elem2
```

The iterator `elem1` refers to the first element we want to erase, an `elem2` refers to one past the last element that we want to remove.

To delete all the elements in a container, we can either call clear or pass iterators from `begin` and `end` to erase:

```
slist.clear(); //delete all the elements within the container
slist.erase(slist.begin(), slist.end());
```

Exercise



Exercise 9.26: Using the following definition of `ia`, copy `ia` into a `vector` and into a `list`. Use the single-iterator form of `erase` to remove the elements with odd values from your `list` and the even values from your `vector`.

Click here to view code image

```
int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 89 };
```

See 9_26.cpp for code