【C++】Day nine(4)

Class	C++
Date	@November 27, 2021
Material	
Series Number	
Summary	

【Ch3】Iterators

3.4.1 Using Iterators

Iteratior Operations

Iterators support only a few operations. We can compare two valid iterators using `==` or `!=`. Iterators are equal if they denote the same element or if they are both off-the-end iterators for the same container. Otherwise, they are unequal.

The following table lists all of the operations of iterators:

Table 3.6. Standard Container Iterator Operations

`*iter`	Returns a reference to the element denoted by the iterator `iter`.
`iter->mem`	Dereferences `iter` and fetches the member named `mem` from the underlying element. Equivalent to `(*iter).mem`.
`++iter`	Increments `iter` to refer to the next element in the container.
`--iter`	Decrements `iter` to refer to the previous element in the container.
`iter1 == iter2` `iter1 != iter2`	Compares two iterators for equality (inequality). Two iterators are equal if they denote the same element or if they are the off-the-end iterator for the same container.

As with pointers, we can dereference an iterator to obtain the element denoted by an iterator. Also, like pointers, we may dereference only a vlid iterator that denotes an element. Dereferencing an invalid iterator or an off-the-end iterator has undefined behavior.

See the following code for an example:

```
string s("some string");
if(s.begin() != s.end()) { //make sure s is not empty
  auto it = s.begin(); //it denotes the first character in s
  *it = toupper(*it); //make that character uppercase
}
```

Moving Iterators from One Element to Another

Iterators use the increment `++` operator to move from one element to the next. Incrementing an iterator is a logically similar operation to incrementing an integer. In this case, the effect is to "advance the iterator by one position."

Note: Because the iterator returned from `end` does not denote an element, it may not be incremented or dereferenced.

Using the increment operator, we can rewrite our program that change the case of the first word in a string to use iterators instead:

```
//process characters in s until we run out of characters
for(auto it = s.begin(); it != s.end(); it++)
  *it = toupper(*it); //capitalize the current character
```

Iterator Types

We generally do not know the precise type of an iterator. Instead, as with `size_type`, the library types that have iterators define types named `iterator` and `const_iterator` that represent actually iterator types:

```
vector<int>::iterator it; //it rcan read and write vector<int> elements
vector<int>::const_iterator it3; //it can read but not write elements
```

A `const_iterato`r behaves like a const pointer. A const_iterator may read but not write elements.

The begin and end Operations

The type returned by `begin` and `end` depends on whether the object on which they operate on is `const`. If the object is const, then begin and end return a `const_iterator`; if the object is not const, they return `iterator`:

```
vector<int> v;
const vector<int> cv;
auto it1 = v.begin(); //it1 has type vector<int>::iterator
auto it2 = cv.begin(); //it2 has type vector<int>::const_iterator
```

Often this default behavior is not what we want. It is usually best to use a `const` type(such as `const_iterator`) when we need to read but do not need to write to an object. To let us ask specifically for the `const_iterator` type, the new standard introduced two new functions names `cebing` and `cend`;

```
auto it3 = v.cbegin(); //it3 has type vector<int>::const_iterator
```

Combining Dereference and Member Access

We can use the arrow operator `->` to access member functions of the object referenced by an iterator:

```
vector<string> k{""};
auto it = k.begin();
if(it->empty())
  cout << "Empty" << endl;
```

Warning: For now, it is important to realize that loops that use iterators should not add elements to the container to which the iterators refer.