【C++】Day31

Class	C++
Date	@January 3, 2022
Material	
Series Number	
Summary	Container Type, begin and end members, and Container Initialization

【Ch9】Sequential Container

9.2.2 Container Type Members

In addition to the iterator types we've already used, most containers provide reverse iterators.

A reverse iterator is an iterator that goes backward through a container and inverts the meaning of the iterator operations.

For example, saying `++` on a reverse iterator yields the previous element.

The remaining types aliases let us use the type of the elements stored in a container without knowing what that type is. If we need the element type, we refer to the container's `value_type`.

If we need a reference to that type, we use `reference` or `const_reference`.

To use one of these types, we must name the class of which they are a member:

```
//iter is the iterator type defined by list<string>
list<string>::iterator iter;
//count is the difference_type type defined by vector<int>
vector<int>::difference_type count;
```

Exercise

Exercise 9.7: What type should be used as the index into a `vector` of `int`s?

We should use `std::vector<int>::size_type`

Exercise 9.8: What type should be used to read elements in a `list` of `string`s? To write them?

We should use `std::list<std::string>::const_iterator` or `std::list<std::string>::iterator` to read elements.

We should use `std::list<std::string>::iterator` to write elements.

9.2.3 begin and end Members

The `begin` and `end` operations yield iterators that refer to the first and one past the last element in the container.

There are several versions of begin and end:

- The version with an `r` return reverse iterators
- Those that start with a `c` return the `const` version of the related iterator:

```
list<string> a = {"Milton", "Shakespeare", "Austen"};
auto it1 = a.begin(); //list<string>::iterator
auto it2 = a.rbegin(); //list<string>::reverse_iterator
auto it3 = a.cbegin(); //list<string>::const_iterator
auto it4 = a.crbegin(); //list<string>::const_reverse_iterator
```

The functions that do not begin with a `c` are overloaded.

- One is a `const` member that returns the container's `const_iterator` type.
- The other is `nonconst` and returns the container's `iterator` type.

As with pointers and references to `const`, we can convert a plain iterator to `const_iterator` but not vice versa.

Best Practices: When write access is not needed, use `cbegin` and `cend`.

Exercise

Exercise 9.10: What are the types of the following four objects?

Click here to view code image

```
vector<int> v1;
const vector<int> v2;
auto it1 = v1.begin(), it2 = v2.begin();
auto it3 = v1.cbegin(), it4 = v2.cbegin();
```

The type of

- `it1` is `std::vector<int>::iterator`
- `it2`, `it3`, and `it4` are `std::vector<int>const_iterator`

9.2.4 Defining and Initializing a Container

Every container type defines a default constructor.

With the exception of array, the default constructor creates an empty container of the specified type.

Initializing a Container as a Copy of Another Container

There are two ways to create a new container as a copy of another one:

- We can directly copy the container
- Or we can copy a range of elements denoted by a pair of iterators

To create a container as a copy of another container, the container and element types must match.

When we pass iterators, there is no requirement that the container types be identical.

Moreover, the element types in the new and original containers can differ as long as it is possible to convert the elements we're copying to the element type of the container we are initializing:

```
//each container has three elements, initialized from the given initializers
list<string> authors = {"Milton", "Shakespeare", "Austen"};
vector<const char*> articles = {"a", "an", "the"};
list<string> list2(authors); //ok: types match
deque<string> authList(authors); //error: container types don't match
vector<string> words(articles); //error: element types must match
//ok: converts const char* elements to string
forward_list<string> words(articles.begin(), articles.end());
```

Note: When we initialize a container as a copy of another container, the container type and element type of both containers must be identical.

Because the iterators denote a range, we can use this constructor to copy a subsequence of a container.

```
//copies up to but not including the element denoted by it
deque<string> authList(authors.begin(), it);
```

List Initialization

Under the new standard, we can list initialize a container:

```
//each cnotainer has three elements, initialized from the given initializers
list<string> authors = {"Milton", "Shakespeare", "Austen"};
```

For types other than array, the initializer list also implicitly specifies the size of the container.

Sequential Container Size-Related Constructors

We can also initialize the sequential containers from a size and an (optional) element initializer. If we do not supply an element initializer, the library creates a value-initialized one for us:

```
vector<int> ivec(10, -1); //ten int elements, each initialized to -1
list<string> svec(10, "hi!"); //ten strings: each element is “hi!"
```

```
forward_list<int> ivec(10); //ten elements: each initialized to 0
deque<string> svec(10); //ten elements, each an empty string
```

Note: the constructors that take a size are valid only for sequential containers; they are not supported for the associative containers.

Library arrays Have Fixed Size

The size of a library array is part of its type. When we define an array, in addition to specifying the element type, we also specify the container size:

```
array<int, 42> //type is: array that holds 42 ints
```

To use an array type we must specify both the element type and the size:

```
array<int, 10>::size_type i; //array type includes element type and size
array<int>::size_type j; //error: array<int> is not a type
```

If the element type is a class type, the class must have a default constructor in order to permit value initialization:

```
array<int, 10> ial; //ten default-initialized ints
array<int, 10> ia2 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
array<int, 10> ia3 = {42}; //ia3[0] is 42, remaining elements are 0
```

Exercise

Exercise 9.11: Show an example of each of the six ways to create and initialize a `vector`. Explain what values each `vector` contains.

```
    //default constructor initialize
    vector<int> vec1;

    //list initialization
    vector<int> vec2 = {1, 2, 3};
```

```
    vector<int> vec7{1, 2, 3};

    //constructor with size
    vector<int> vec3(10);

    //constructor with size and value
    vector<int> vec4(10, 0);

    //copy from another vector
    vector<int> vec5(vec2);

    //copy from another vector of the same type
    vector<int> vec6 = vec2;

    //copy using iterator
    vector<int> vec8(vec2.begin(), vec2.end());
```

Exercise 9.12: Explain the differences between the constructor that takes a container to copy and the constructor that takes two iterators.

The constructor that take a container require the container to be the same type of our destination container.

However, the element that the iterators point to only need to be convertible to that of the container.

Exercise 9.13: How would you initialize a `vector<double>` from a `list<int>`? From a `vector<int>`? Write code to check your answers.

```
vector<double> doubleVec(10, 5.0);
list<int> list(doubleVec.cbegin(), doubleVec.cend());
vector<int> intvec(doubleVec.begin(), doubleVec.end());
```