【C++】Day30(2)

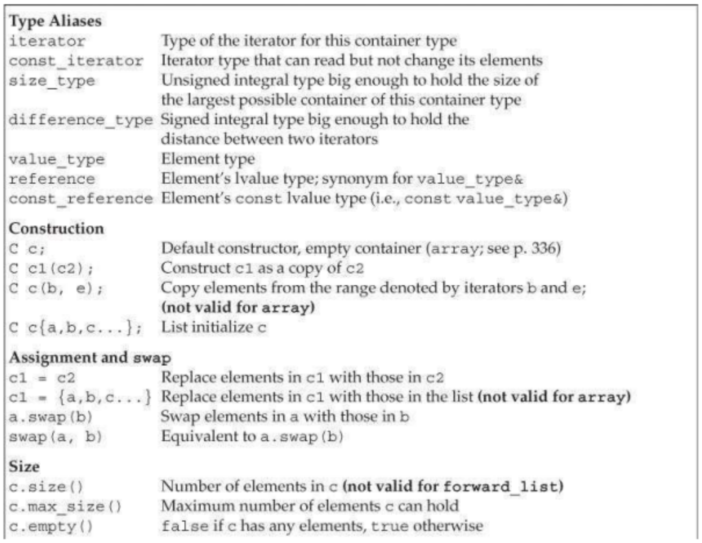
Class	C++
Date	@January 2, 2022
Material	
Series Number	
Summary	

【Ch9】Sequential Containers

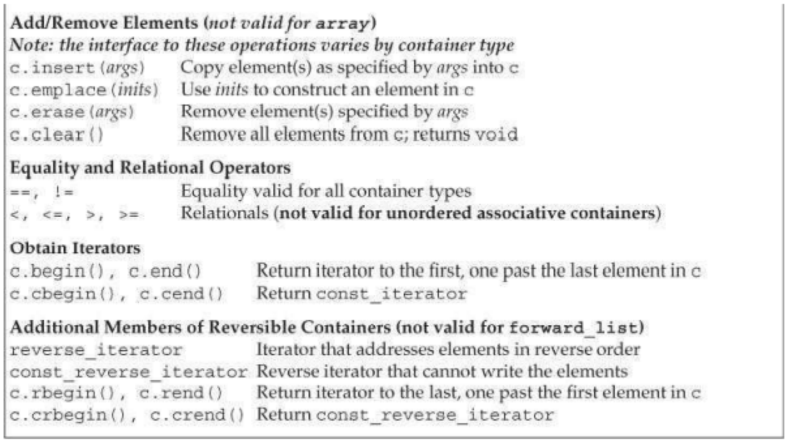
9.2 Container Library Overview

Some operations(listed below) are provided by all container types.

Table 9.2. Container Operations



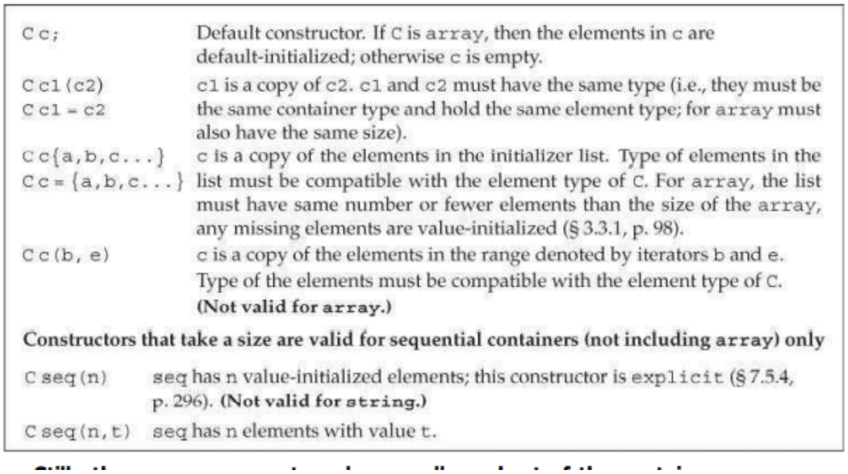
Type Aliases	
`iterator`	Type of the iterator for this container type
`const_iterator`	Iterator type that can read but not change its elements
`size_type`	Unsigned integral type big enough to hold the size of the largest possible container of this container type
`difference_type`	Signed integral type big enough to hold the distance between two iterators
`value_type`	Element type
`reference`	Element's lvalue type; synonym for `value_type&`
`const_reference`	Element's `const` lvalue type (i.e., `const value_type&`)
Construction	
`C c;`	Default constructor, empty container (`array`; see p. 336)
`C c1(c2);`	Construct `c1` as a copy of `c2`
`C c(b, e);`	Copy elements from the range denoted by iterators `b` and `e`; **(not valid for `array`)**
`C c{a,b,c...};`	List initialize `c`
Assignment and `swap`	
`c1 = c2`	Replace elements in `c1` with those in `c2`
`c1 = {a,b,c...}`	Replace elements in `c1` with those in the list **(not valid for `array`)**
`a.swap(b)`	Swap elements in `a` with those in `b`
`swap(a, b)`	Equivalent to `a.swap(b)`
Size	
`c.size()`	Number of elements in `c` **(not valid for `forward_list`)**
`c.max_size()`	Maximum number of elements `c` can hold
`c.empty()`	`false` if `c` has any elements, `true` otherwise



Add/Remove Elements (*not valid for* `array`)	
Note: the interface to these operations varies by container type	
`c.insert(`*args*`)`	Copy element(s) as specified by *args* into `c`
`c.emplace(`*inits*`)`	Use *inits* to construct an element in `c`
`c.erase(`*args*`)`	Remove element(s) specified by *args*
`c.clear()`	Remove all elements from `c`; returns `void`
Equality and Relational Operators	
`==, !=`	Equality valid for all container types
`<, <=, >, >=`	Relationals **(not valid for unordered associative containers)**
Obtain Iterators	
`c.begin(), c.end()`	Return iterator to the first, one past the last element in `c`
`c.cbegin(), c.cend()`	Return `const_iterator`
Additional Members of Reversible Containers (not valid for `forward_list`)	
`reverse_iterator`	Iterator that addresses elements in reverse order
`const_reverse_iterator`	Reverse iterator that cannot write the elements
`c.rbegin(), c.rend()`	Return iterator to the last, one past the first element in `c`
`c.crbegin(), c.crend()`	Return `const_reverse_iterator`

The following operations(listed below) are specific to the sequential containers.

Table 9.3. Defining and Initializing Containers



`C c;`	Default constructor. If `C` is `array`, then the elements in `c` are default-initialized; otherwise `c` is empty.

`C c1(c2)`
`C c1 = c2`
	`c1` is a copy of `c2`. `c1` and `c2` must have the same type (i.e., they must be the same container type and hold the same element type; for `array` must also have the same size).

`C c{a,b,c...}`
`C c = {a,b,c...}`
	`c` is a copy of the elements in the initializer list. Type of elements in the list must be compatible with the element type of `C`. For `array`, the list must have same number or fewer elements than the size of the `array`, any missing elements are value-initialized (§ 3.3.1, p. 98).

`C c(b, e)`	`c` is a copy of the elements in the range denoted by iterators `b` and `e`. Type of the elements must be compatible with the element type of `C`. **(Not valid for `array`.)**
Constructors that take a size are valid for sequential containers (not including `array`) only	
`C seq(n)`	`seq` has `n` value-initialized elements; this constructor is `explicit` (§ 7.5.4, p. 296). **(Not valid for `string`.)**
`C seq(n,t)`	`seq` has `n` elements with value `t`.

In general, each container is defined in a header file with the same name as the type.

That is, deque is in the deque header, list in the list header, and so on.

Constraints on Types That a Container Can Hold

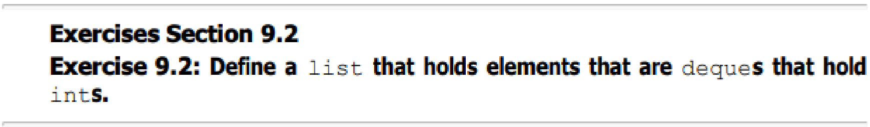
Almost any type can be used as the element type of a sequential container. In particular, we can define a container whose element type is itself another container.

We define such containers exactly as we do any other container type: We specify the element type inside angle brackets:

```
vector<vector<string>> lines; //vector of vectors of strings
```

Note: Older compilers may require a space between the angle brackets, for example, `vector< vector<string> >`.

Exercise



Exercises Section 9.2

Exercise 9.2: Define a `list` that holds elements that are `deque`s that hold `int`s.

```
#include <deque>
#include <list>

int main() {
  std::list<std::deque<int>> list_of_deque;
  return 0;
}
```

9.2.1 Iterators

Iterators support increment(`++`) and decrement(`--`) operators except for the one for `forward_list`.

Iterator Ranges

Note: The concept of an iterator range is fundamental to the standard library.

An iterator range is denoted by a pair of iterators each of which refers to an element, or to one past the last element, in the same container.

These two iterators, often referred to as `begin` and `end` mark a range of elements from the container.

The iterator `end` may be equal to `begin` but must not refer to an element before the one denoted by `begin`.

Requirements on Iterators Forming an Iterator Range

Two iterators, `begin` and `end`, form an iterator range, if

- They refer to elements of, or one past the end of, the same container
- It is possible to reach end by repeatedly incrementing `begin`. In other words, `end` must not precede `begin`.

Warning: The compiler cannot enforce these requirements. It is up to us to ensure that our programs follow these conventions.

Programming Implications of Using Left-Inclusive Ranges

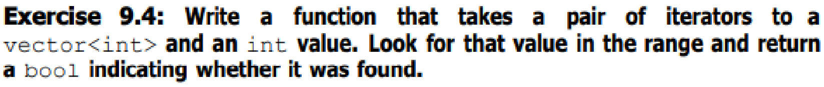
Assuming begin and end denote a valid iterator range, then

- If `begin` equals `end`, the range is empty
- If `begin` is not equal to `end`, there is at least one element in the range, and `begin` refers to the first element in that range
- We can increment `begin` some number of times until `begin == end`

These properties mean that we can safely write loops such as the following:

```
while(begin != end) {
  *begin = val; //ok: range isn't empty so begin denotes an element
  ++begin; //advance the iterator to get the next element
}
```

Exercise



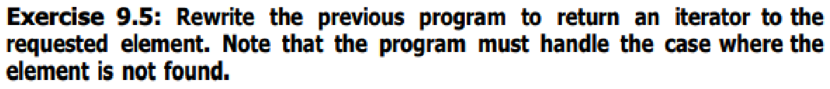
Exercise 9.4: Write a function that takes a pair of iterators to a `vector<int>` **and an** `int` **value. Look for that value in the range and return a** `bool` **indicating whether it was found.**

```
#include <vector>
#include <iostream>

bool isInVector(std::vector<int>::const_iterator begin, std::vector<int>::const_iterator end, int key) {
  while(begin != end) {
    //order of operator evaluation: 1.dereference 2.increment the iterator
    if(*begin++ == key)
      return true;
  }
  return false;
}

int main() {
    std::vector<int> vec{1, 2, 3, 4, 5};
    if(isInVector(vec.cbegin(), vec.cend(), 3))
```

```
        std::cout << "The element exists" << std::endl;
    return 0;
}
```



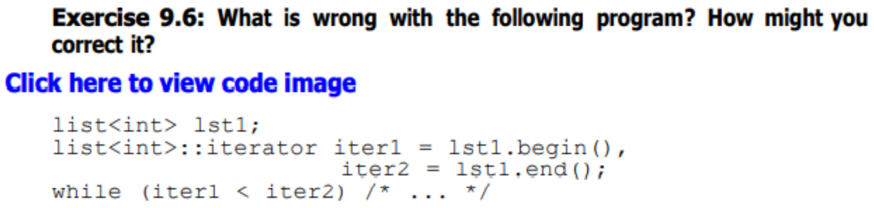
Exercise 9.5: Rewrite the previous program to return an iterator to the requested element. Note that the program must handle the case where the element is not found.

```
#include <vector>
#include <iostream>

std::vector<int>::const_iterator isInVector(std::vector<int>::const_iterator begin, std::vector<int>::const_iterator end, int key) {
  while(begin < end) {

    if(*begin == key)
      return begin;
    ++begin;
  }
  return end;
}

int main() {
    std::vector<int> vec{1, 2, 3, 4, 5};
    if(isInVector(vec.cbegin(), vec.cend(), 6) == vec.cend())
        std::cout << "The element does not exist" << std::endl;
    else
        std::cout << "The element exists" << std::endl;
    return 0;
}
```



Exercise 9.6: What is wrong with the following program? How might you correct it?

Click here to view code image

```
list<int> lst1;
list<int>::iterator iter1 = lst1.begin(),
                    iter2 = lst1.end();
while (iter1 < iter2) /* ... */
```

The `list` container does not support iterator arithmetic `>` and `<`. Thus, we should use `iter1 != iter2`.