【C++】Day38

Class	C++
Date	@January 16, 2022
Material	
Series Number	
Summary	

【Ch10】Generic Algorithms

10.2.2 Algorithms That Write Container Elements

When we use an algorithm that assigns to elements, we must take care to ensure that the sequence into which the algorithm writes is at least as large as the number of elements we ask the algorithm to write.

As one example, the `fill` algorithm takes a pair of iterators that denote a range and a third argument that is a value. `fill` assigns the given value to each element in the input sequence:

```
fill(vec.begin(), vec.end(), 3); //reset each element to 3
//set a subsequent element to 10
fill(vec.begin(), ve.begin() + vec.size() / 2, 10);
```

Because `fill` writes to its given input sequence, so long as we pass a valid input sequence, the writes will be safe.

Algorithms Do Not Check Write Operations

Some algorithm takes an iterator that dentoes a separate destination. These aglrotihms assign new values to the elements of a sequence starting at the element denoted by the destination iterator.

For example, the `fill_n` function takes a single iterator, a count, and a value. It assigns the given value to the specified number of elements starting at the element denoted to by the iterator:

```
vector<int> vec; //empty vector
//use vec giving it various values
fill_n(vec.begin(), vec.size(), 3);
```

The `fill_n` function assumes that it is safe to write the specified number of elements.

It is a common beginner mistake to call `fill_n`(or similar algorithms) that write to elements on a container that has no elements:

```
vector<int> vec; //empty vector
//disaster: attempts to write to ten elements in vec
fill_n(vec.begin(), 10, 0);
```

Warning: Algorithms that write to a detination tierator assume the destination is large enough to hold the number of elements begin written

Introducing back_inserter

One way to ensure that an algorithm has enough elements to hold the output is to use an insert iterator. An insert iterator is an iterator that adds elements to a container.

When we assign though an insert iterator, a new element equal to the right-hand value is added to the container.

In order to illustrate how to use algorithms that write to a container, we will use `back_inserter` which is a function defined in the `iterator` header.

`back_inserter` takes a reference to a container and returns an insert iterator bound to that container. When we assign through that iterator, the assignment calls `push_back` to add an element with the given value to the container.

```
vector<int> vec; //empty vector
auto it = back_inserter(vec); //assigning through it adds to vec
*it = 42;
```

We frequently use `back_inserter` to create an iterator to use as the destination of an algorithm. For example:

```
vector<int> vec; //empty vector
fill_n(back_inserter(vec), 10, 0); //appends ten elements to vec
```

On each iteration, `fill_n` assigns to an element in the given sequence. Because we passed an iterator returned by `back_inserter`, each assignment will call `push_back` on vec. As a result, this call to `fill_n` adds ten elements to the end of vec, each of which has the value 0.

Copy Algorithms

The `copy` algorithm is another example of an algorithm that wrties to the elements of an output sequence denoted by a destination iterator.

This algorithm takes three iterators. The first two denote an input range; the thrid denotes the beginning of the destination sequence. This algorithm copies elements from its input range into elements in the destination.

As one example, we can use copy to copy one built-in array to another:

```
int a1[] = {1, 2, 3, 4, 5};
int a2[sizeof(a1) / sizeof(*a1)];
//ret points just past the last element copied into a2
auto ret = copy(begin(a1), end(a1), a2);
```

Several algorithms provide so-called “copying” versions. These algorithms compute new element values, but instead of putting them back into their input sequence, the algorithms create a new sequence to contain the results.

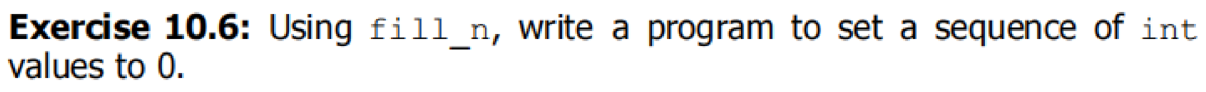
For example, the `replace` algorithm reads a sequence and replaces every instance of a given value with another value. This algorithm takes four parameters: two iterators denoting the input range, and two values. It replaces each element that is equal to the first value with the second:

```
//replace any element with the value 0 with 42
replace(ilst.begin(), ilst.end(), 0, 42);
```

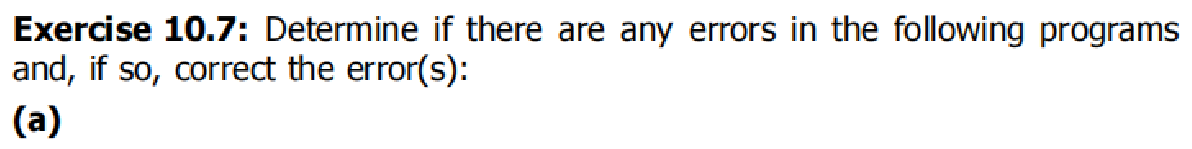
If we want to leave the original sequence unchanged, we can call `replace_copy`. That algorithm takes a third iterator argument denoting a destination in which to write the adjusted sequence:

```
//use back_inserter to grow destination as needed
replace_copy(ilst.cbegin(), ilst.cend(), back_inserter(ivec), 0, 42);
```

Exercise

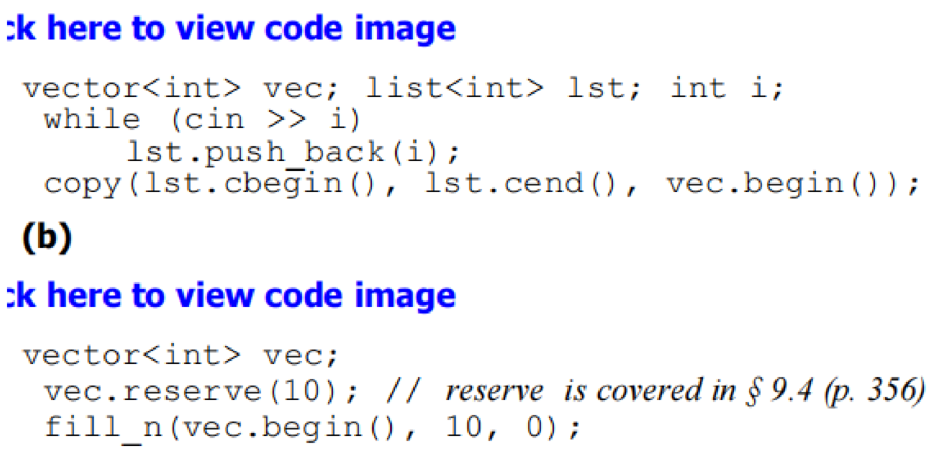

Exercise 10.6: Using `fill_n`, write a program to set a sequence of `int` values to 0.

See 10_6.cpp for code



Exercise 10.7: Determine if there are any errors in the following programs and, if so, correct the error(s):

(a)



k here to view code image

```
vector<int> vec; list<int> lst; int i;
 while (cin >> i)
     lst.push_back(i);
 copy(lst.cbegin(), lst.cend(), vec.begin());
```

(b)

k here to view code image

```
vector<int> vec;
 vec.reserve(10); // reserve is covered in § 9.4 (p. 356)
 fill_n(vec.begin(), 10, 0);
```

- (a): `vec` does not have enough elements to be copied to
- (b): `reserve()` allocates memory for ten elements, but the `fill_n` function needs ten elements to be copied to. Not just memories.