【C++】Day34(2)

Class	C++
Date	@January 6, 2022
Material	
Series Number	
Summary	

【Ch9】Sequential Container

9.3.6 Container Operations May Invalidate Iterators

Operations that add or remove elements from a container can invalidate pointers, references, or iterators to container elements.

After an operation that adds elements to a container:

- Iterators, pointers, and references to a `vector` or `string` are invalid if the container was reallocated. If no reallocation happens, indirect references to elements before the insertion remain valid; those to elements after the insertion are invalid.
- Iterators, pointers, and references to a `deque` are invalid if we add elements anywhere but at the front or back. If we add at the front or back, iterators are invalidated, but references and pointers to existing elements are not
- Iterators, pointers, and references to a `list` or `forward_list` remain valid.

Warning: It is a seriours run-time error to use an iterator, pointer, or reference that has been invalidated.

Advice: Managing Iterators

When we use an iterator(or a reference or pointer to a container element), it is a good idea to minimize the part of the program during which an iterator must stay valid.

Because code that adds or removes elements to a container can invalidate iterators, you need to ensure that the iterator is repositioned, as appropriate, after each operation

that changes the container. This advice is especially important for vector, string, and deque.

Writing Loops that Change a Container

Loops that add or remove elements of a vector, string, or deque must cater to the fact that iterators, references, or pointers might be invalidated.

The program must ensure that the iterator, reference, or pointer is refreshed on each trip through the loop.

Refreshing an iterator is easy if the loop calls insert or erase. Those operations return iterators, which we can use to reset the iterator:

```
vector<int> vi = {0, 1, 2, 3, 4, 5, 6, 7, 8};
auto iter = vi.begin();
while(iter != vi.end()) {
  if(*iter %2) {
    iter = vi.insert(iter, *iter); //duplicate the element
    iter += 2;
  } else {
    iter = vi.erase(iter);
    //don;t advance the iterator; iter denotes the element after the one we erased
  }
}
```

Avoid Storing the Iterator Returned from end

Loops that add or remove elements should always call `end` rather than use a stored copy. Partly for this reason, C++ standard libraries are usually implemented so that calling `end()` is a very fast operation.

Tip: Don’t cache the iterator returned from end() in loops that insert or delete elements in a deque, string, or vector.

Exercise

Exercise 9.31: The program on page 354 to remove even-valued elements and duplicate odd ones will not work on a `list` or `forward_list`. Why? Revise the program so that it works on these types as well.

forward_list:

```
#include <iostream>
#include <string>
#include <forward_list>


int main(int argc, char **argv) {
  std::forward_list<int> flst = { 0, 1, 2, 3, 4, 5, 6, 7, 8 };
  auto prev = flst.before_begin();
  auto cur = flst.begin();
  while(cur != flst.end()) {
    if(*cur % 2) {
      prev = flst.insert_after(prev, *cur);
      cur++;
      prev++;
    } else {
      cur = flst.erase_after(prev);
    }
  }

  for(auto a : flst)
    std::cout << a << " ";
  return 0;
}
```

list:

```
#include <iostream>
#include <string>
#include <forward_list>
#include <list>


int main(int argc, char **argv) {
  std::list<int> lst = {0, 1, 2, 3, 4, 5, 6, 7, 8};
  auto begin = lst.begin();
  while(begin != lst.end()) {
    if(*begin % 2) {
      begin = lst.insert(begin, *begin);
      begin++; begin++;
      //begin+=2 list doesn't support iterator arithmetic
    } else {
      begin = lst.erase(begin);
```

```
    }
  }

  for(auto a : lst)
    std::cout << a << " ";
  return 0;
}
```

Exercise 9.33: In the final example in this section what would happen if we did not assign the result of `insert` to `begin`? Write a program that omits this assignment to see if your expectation was correct.

If we don't update `begin`, begin will become invalid after the first insertion and any following operation on `begin` would be undefined.

Exercise 9.34: Assuming `vi` is a container of `int`s that includes even and odd values, predict the behavior of the following loop. After you've analyzed this loop, write a program to test whether your expectations were correct.

Click here to view code image

```
iter = vi.begin();
while (iter != vi.end())
    if (*iter % 2)
        iter = vi.insert(iter, *iter);
    ++iter;
```

If the container contains any odd number, then the program will run infinitely. `iter` points to the newly copied odd number, incrementing it have it point to the original element.