Lecture 5 Associative Containers & Iterators

#CS106L

- **Container Adaptors**
- **Assignment 1**
- **Associative Containers**
- **Iterators**
- **Map iterators**

`Stack`:Just limit the functionality of a `vector/deque` to only allow `push_back` and `pop_back`.

`Queue`:Just limit the functionality of a `deque` to only allow `push_back` and `pop_front`.

A `set` is just a specific case of a map that doesn’t have a value! ○ Or you can think of the value as being true (if present) or false

General style advice: Use C++, not C, Decompose and use constants.

Iterators allow iteration over any container,whether it is ordered or not.

A summary of the essential iterator operations:

- Create iterator

- Dereference iterator to read value currently pointed to

- Advance iterator(` `++iter` is better)

- Compare against another iterator (especially `.end()` iterator)

Many scenarios require looking at elements, regardless of what type of container is storing those elements . Iterators let us go through sequences of elements in a standardised way.

Q1:How can we implement `stack` and `deque` using the containers we have?if we only use `vector` and `deque` and C++ basic syntax.

Here is an example of how you can implement a stack using a `std::vector`:

```
#include <vector>



class Stack {

    public:

        void push(int value) {

            data.push_back(value);

        }



        int pop() {

            int value = data.back();

            data.pop_back();

            return value;

        }



        int top() {

            return data.back();

        }



        bool empty() {

            return data.empty();

        }

    private:

        std::vector<int> data;

};
```

In this example, the `push` method adds an element to the back of the vector, the `pop` method removes and returns the last element of the vector, the `top` method returns the last element of the vector, and the `empty` method returns whether the vector is empty or not.

Here is an example of how you can implement a queue using a `std::deque`:

```
#include <deque>



class Queue {

    public:

        void push(int value) {

            data.push_back(value);

        }



        int pop() {

        int value = data.front();

        data.pop_front();

        return value;

    }



    int front() {

        return data.front();

    }



    int back() {

        return data.back();

    }



    bool empty() {

        return data.empty();

    }

private:

    std::deque<int> data;
```

In this example, the `push` method adds an element to the back of the deque, the `pop` method removes and returns the first element of the deque, the `front` method returns the first element of the deque, the `back` method returns the last element of the deque, and the `empty` method returns whether the deque is empty or not.

It's worth noting that the above examples are the basic implementation of stack and queue, you can add other functionalities to them like checking for overflow and underflow and also you can add a constructor and destructor to them.

In summary, you can implement a stack using a `std::vector`, by adding elements to the back and removing elements from the back and you can implement a queue using a `std::deque`, by adding elements to the back and removing elements from the front.

Q2:How do we iterate over associative containers in c++? They have no notion of a sequence/indexing.

Associative containers, such as `std::map` and `std::unordered_map`, are designed to store key-value pairs and are typically implemented as balanced trees or hash tables. They do not have a notion of a sequence or indexing, so you cannot use an index to iterate over the elements of an associative container. However, you can still iterate over the elements of an associative container using iterators.

Here's an example of how you can iterate over the elements of a `std::map` using an iterator:

```
#include <iostream>

#include <map>



int main() {

    std::map<std::string, int> my_map;

    my_map["apple"] = 1;

    my_map["banana"] = 2;

    my_map["cherry"] = 3;



    // Iterating over the elements of the map

    for (std::map<std::string, int>::iterator it = my_map.begin(); it !=
my_map.end(); it++) {

        std::cout << it->first << ": " << it->second << std::endl;

    }

    return 0;
}
```

You can also use C++11 range-based for loop to iterate over elements of the associative container:

```
for(auto element : my_map) {

    std::cout << element.first << ": " << element.second << std::endl;

}
```

The above code will also produce the same output as previous example.

You can do the same for `std::unordered_map` as well.

Note that the order of elements in associative container is not guaranteed to be in the order of insertion, it's guaranteed to be sorted by key.

In summary, you can iterate over the elements of an associative container, such as `std::map` and `std::unordered_map`, using iterators or C++11 range-based for loop, but they do not have a notion of a sequence or indexing, so you cannot use an index to iterate over the elements.

Q3:How are iterators able to represent a non-linear collection in a “sequential” way?

Iterators are a mechanism that allows you to traverse through the elements of a container in a linear, sequential way, even if the underlying container is not stored in a linear fashion.

For example, an associative container, such as `std::map` or `std::unordered_map`, is typically implemented as a balanced tree or hash table, which allows for efficient insertion, deletion, and lookup of elements, but does not have a linear ordering of elements. However, by providing iterators, the container can be traversed in a linear, sequential way.

The `begin()` and `end()` methods of the container return iterators to the first and last elements of the container, respectively. The iterators can be incremented or decremented to move through the elements of the container one at a time.

For example, when you call `std::map::begin()`, it returns an iterator that points to the first element in the container(first element according to the ordering of the keys), and when you call `std::map::end()`, it returns an iterator that points to one past the last element in the container. The iterator can be incremented to point to the next element, and this process can be repeated until the end of the container is reached.

The container internally keeps track of the elements and their position in the tree or hash table, so the iterators can navigate through the elements in a linear fashion.

It's worth noting that the iterators are only able to traverse the container in a linear way if the container is sorted, otherwise the order of elements will not be guaranteed.

In C++, iterators are typically implemented as a separate class or struct, which defines the various operators required for traversing through the elements of a container. The container class then provides `begin()` and `end()` methods that return iterators to the first and last elements of the container, respectively.

Here is an example of how you can implement a simple iterator for a singly-linked list:

```
template<typename T>

struct SinglyLinkedList {

    struct Node {

        T value;

        Node* next;

    };

    Node* head;



    struct Iterator {

        Node* current;



        Iterator& operator++() {
            current = current->next;

            return *this;

        }



        T operator*() const {

            return current->value;

        }



        bool operator!=(const Iterator& other) const {

            return current != other.current;

        }

    };



    Iterator begin() {

        return { head };

    }



    Iterator end() {

        return { nullptr };

    }

};
```

You can use the iterator like this:

```
        SinglyLinkedList<int> my_list;



// Add elements to the list

        my_list.push_front(1);

        my_list.push_front(2);

        my_list.push_front(3);



// Iterate through the elements of the list

        for (SinglyLinkedList<int>::Iterator it = my_list.begin(); it !=
my_list.end(); it++) {

    std::cout << *it << std::endl;

}
```

In summary, to implement an iterator for a container class, you need to define a separate class or struct for the iterator, and define the various operators required for traversing through the elements of the container. The container class should then provide `begin()` and `end()` methods that return iterators to the first and last elements of the container, respectively. And you can use these iterators to traverse through the elements of the container in a linear, sequential way.