The iterator pattern

START

Team:

Phan Huỳnh Anh Thư - 17095
Võ Công Minh - 10421040

The iterator pattern

START

Example:

All roads lead to Rome

Definition:

The Iterator Pattern is a design pattern in object-oriented programming that provides a way to traverse through a collection of objects in a systematic and efficient way without exposing the underlying implementation of the collection. It is one of the most commonly used patterns in software development and is often used to manage large sets of data.

Structure of Iterator Pattern

Iterator interface: This interface defines the operations on the iterators - hasNext(), next(), remove() etc. This interface is implemented by the concrete iterators.

Collection interface: This interface defines the createIterator() method which returns an iterator for the collection. This allows multiple traversals of the same collection.

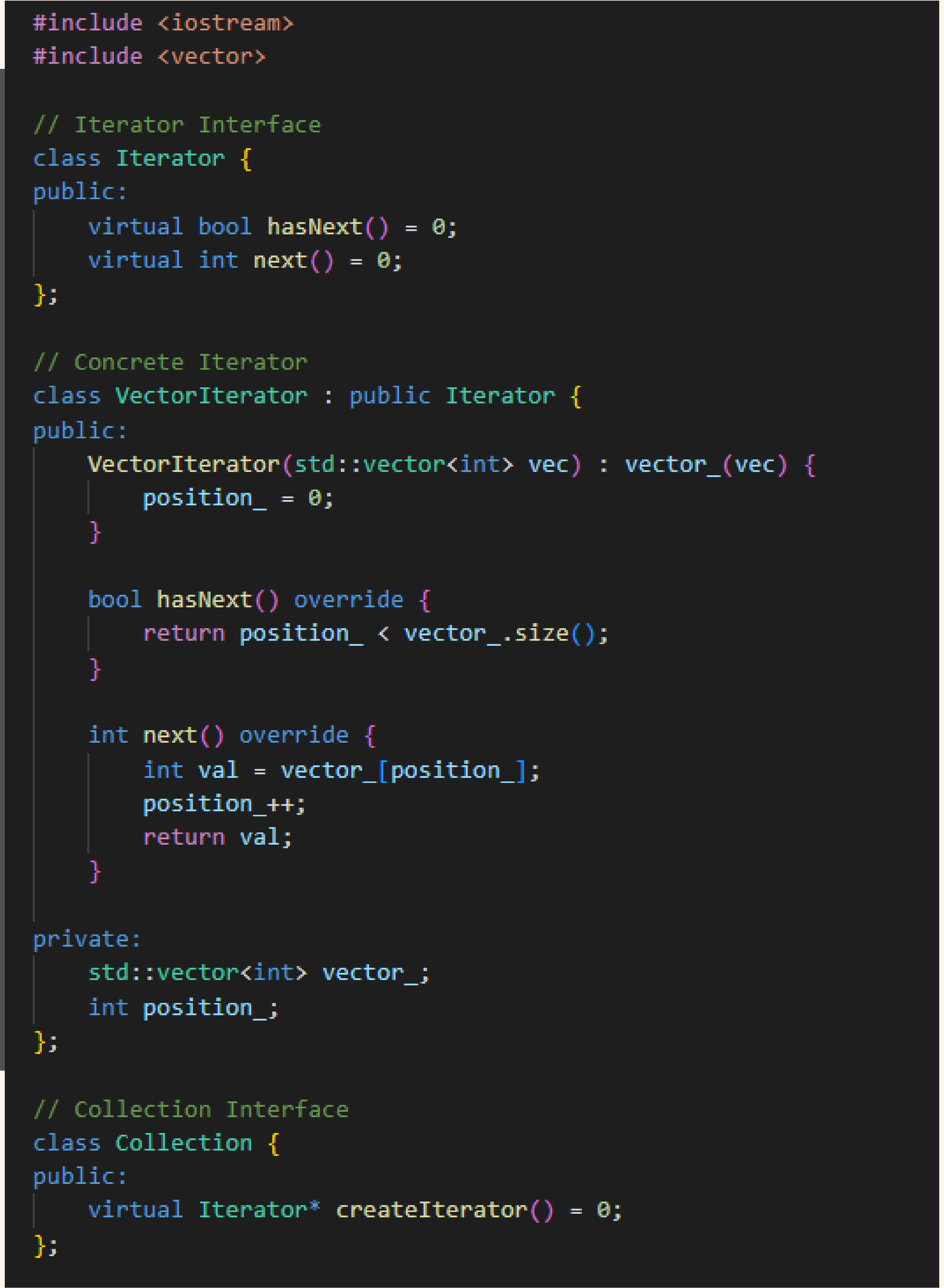
Concrete Iterators: These implement the iterator interface and keep track of the current position in the traversal of the collection. They maintain the traversal logic. Different iterators can be defined for the same collection, allowing multiple traversals of the collection in different ways.

Structure of Iterator Pattern

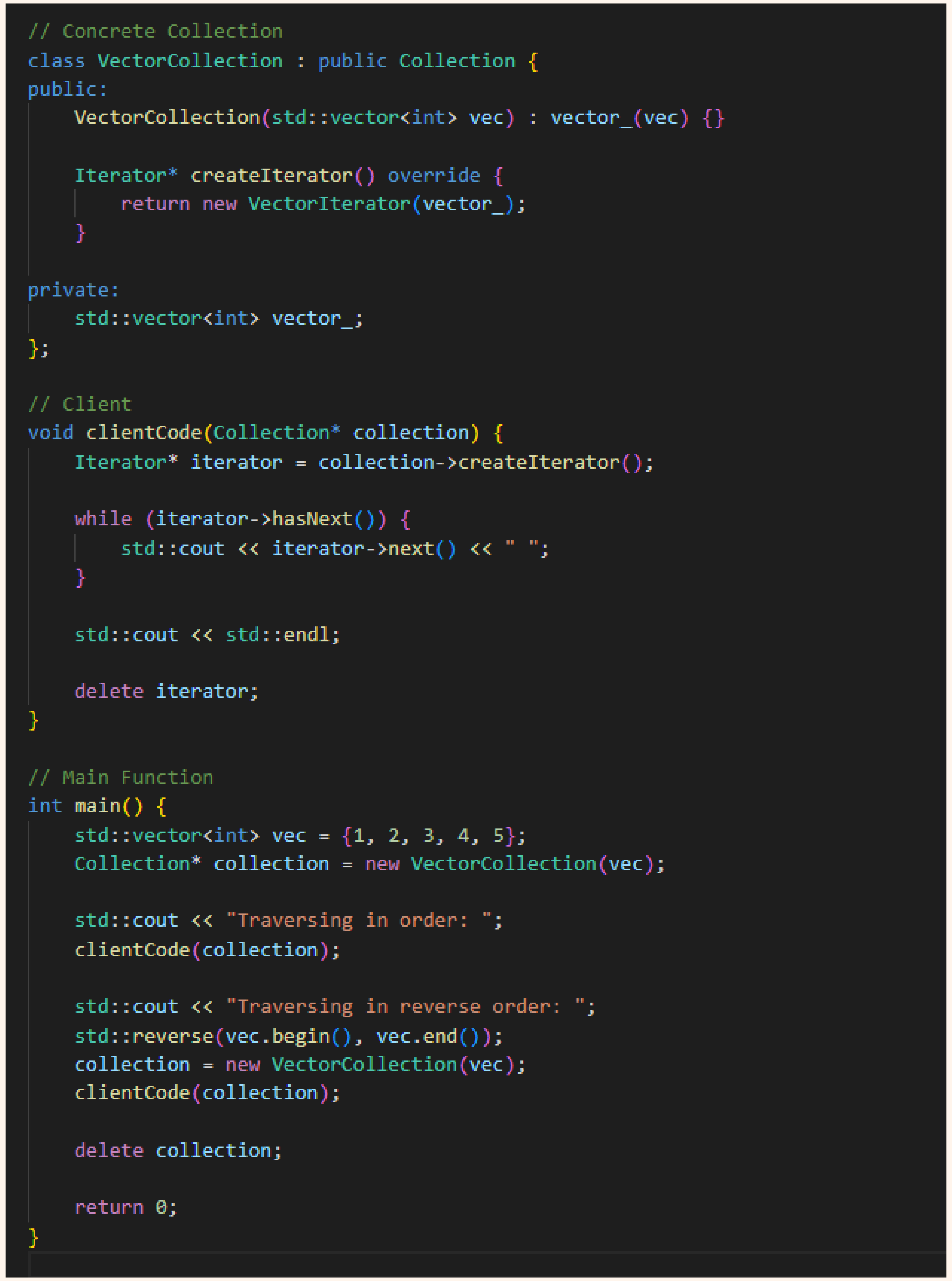
Concrete Collections: These implement the collection interface and provide the actual storage for elements. They return concrete iterators that can traverse the collection.

Client: The client interacts with the collections through the iterators to traverse elements in different ways. The client is decoupled from the implementation details of the collection and how it is traversed.

Implemetation



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

// Iterator Interface
class Iterator {
public:
    virtual bool hasNext() = 0;
    virtual int next() = 0;
};

// Concrete Iterator
class VectorIterator : public Iterator {
public:
    VectorIterator(std::vector<int> vec) : vector_(vec) {
        position_ = 0;
    }

    bool hasNext() override {
        return position_ < vector_.size();
    }

    int next() override {
        int val = vector_[position_];
        position_++;
        return val;
    }

private:
    std::vector<int> vector_;
    int position_;
};

// Collection Interface
class Collection {
public:
    virtual Iterator* createIterator() = 0;
};
```



```cpp
// Concrete Collection
class VectorCollection : public Collection {
public:
    VectorCollection(std::vector<int> vec) : vector_(vec) {}

    Iterator* createIterator() override {
        return new VectorIterator(vector_);
    }

private:
    std::vector<int> vector_;
};

// Client
void clientCode(Collection* collection) {
    Iterator* iterator = collection->createIterator();

    while (iterator->hasNext()) {
        std::cout << iterator->next() << " ";
    }

    std::cout << std::endl;

    delete iterator;
}

// Main Function
int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    Collection* collection = new VectorCollection(vec);

    std::cout << "Traversing in order: ";
    clientCode(collection);

    std::cout << "Traversing in reverse order: ";
    std::reverse(vec.begin(), vec.end());
    collection = new VectorCollection(vec);
    clientCode(collection);

    delete collection;

    return 0;
}
```

Avantages

1 Encapsulation

3 Separation of Concerns

2 Flexibility

4 Efficiency

Disavantages

1 Overhead

2 Complexity

3 Limited Functionality

4 Learning Curve

THANK YOU!