Московский Авиационный Институт
(Национальный Исследовательский Университет)

Кафедра 806 «Вычислительная информатика и программирование»
Факультет: «Информационные технологии и прикладная математика»

Лабораторная работа
Дисциплина: «Объектно-ориентированное программирование»
III семестр
Задание 6: «Основы работы с коллекциями : итераторы»

Группа:	М8О-206Б-18, №27
Студент:	Шорохов Алексей Павлович
Преподаватель:	Журавлёв Андрей Андреевич
Оценка:	
Дата:	

Москва, 2019

Задание

27.	Прямоугольник	Динамический массив	Стек

Адрес репозитория на GitHub

Код программы на C++

CMakeLists.txt
```
cmake_minimum_required(VERSION 3.2)

project(run)

add_executable(run
        Source.cpp
        )

set_property(TARGET run PROPERTY CXX_STANDARD 17)
```

Allocator.h
```
#pragma once

#include <iostream>
#include <algorithm>
#include <list>
#include "Stack.h"

enum class MemoryNodeType {
    Hole,
    Occupied
};

struct MemoryNode {
    char* beginning;
    size_t capacity;
    MemoryNodeType type;
};

template <typename T, size_t ALLOC_SIZE>
class Allocator {
public:
    using value_type = T;
    using size_type = std::size_t;
    using difference_type = std::ptrdiff_t;
    using is_always_equal = std::false_type;

    Allocator(const Allocator&) = delete;
```

```
    Allocator(Allocator&&) = delete;

    template<class V>
    struct rebind {
        using other = Allocator<V, ALLOC_SIZE>;
    };

    Allocator() {
        data = (char *) malloc(ALLOC_SIZE);
        mem_list.Push({data, ALLOC_SIZE, MemoryNodeType::Hole});
    }

    ~Allocator() {
        free(data);
    }

    T* allocate(size_t mem_size) {
        mem_size *= sizeof(T);
        auto it = std::find_if(mem_list.begin(), mem_list.end(), [&mem_size] (const
MemoryNode& node) {
            return node.type == MemoryNodeType::Hole && node.capacity >=
mem_size;
        });
        if (it == mem_list.end()) {
            throw std::runtime_error("No memory");
        }
        if (it->capacity == mem_size) {
            it->type = MemoryNodeType::Occupied;
        } else {
            auto next = std::next(it);
            mem_list.Insert(std::next(it), MemoryNode{it->beginning + mem_size, it-
>capacity - mem_size, MemoryNodeType::Hole});
            it->type = MemoryNodeType::Occupied;
            it->capacity -= mem_size;
        }
        return (T*)it->beginning;
    }


    void deallocate(T* typed_ptr, size_t) {
        auto cur_it = std::find_if(mem_list.begin(), mem_list.end(), [&typed_ptr]
(const MemoryNode& node) {
            return node.type == MemoryNodeType::Occupied && node.beginning ==
(char*) typed_ptr;
        });
```

```
        auto prev_it = mem_list.end();
        for (auto it = mem_list.begin(); it != mem_list.end(); ++it) {
            if (std::next(it) == cur_it) {
                prev_it = it;
                break;
            }
        }
        if (cur_it == mem_list.end()) {
            throw std::runtime_error("Wrong ptr to deallocate");
        }
        if (cur_it != mem_list.begin() && prev_it->type == MemoryNodeType::Hole)
{
            cur_it = prev_it;
            cur_it->capacity += std::next(cur_it)->capacity;
            mem_list.Erase(std::next(cur_it));
        }
        if (std::next(cur_it) != mem_list.end() && std::next(cur_it)->type ==
MemoryNodeType::Hole) {
            cur_it->capacity += std::next(cur_it)->capacity;
            mem_list.Erase(std::next(cur_it));
        }
    }

private:
    Containers::Stack<MemoryNode> mem_list;
    char* data;
};
```

Stack.h

```
#pragma once

#include <memory>
#include <exception>

namespace Containers {

    template <typename T>
    struct StackNode {
        T data;
        std::shared_ptr<StackNode> next;
        std::weak_ptr<StackNode> prev;
    };

    template <typename T>
    struct StackIterator {
        using value_type = T;
```

```
using reference = T&;
using pointer = T*;
using difference_type = ptrdiff_t;
using iterator_category = std::forward_iterator_tag;

StackIterator(std::shared_ptr<StackNode<T>> ptr)
: ptr_(ptr){}

T& operator * () {
    std::shared_ptr<StackNode<T>> locked = ptr_.lock();
    if (!locked) {
        throw std::runtime_error("Iterator does not exist");
    }
    return locked->data;
}

T* operator -> () {
    std::shared_ptr<StackNode<T>> locked = ptr_.lock();
    if (!locked) {
        throw std::runtime_error("Iterator does not exist");
    }
    return &locked->data;
}

StackIterator& operator++() {
    std::shared_ptr<StackNode<T>> locked = ptr_.lock();
    if (!locked || locked->next == nullptr) {
        throw std::runtime_error("Out of bounds");
    }
    ptr_ = locked->next;
    return *this;
}

const StackIterator operator++(int) {
    auto copy = *this;
    ++(*this);
    return copy;
}

bool operator == (const StackIterator& other) const {
    return ptr_.lock() == other.ptr_.lock();
}

bool operator != (const StackIterator& other) const {
    return !(*this == other);
```

```
    }

    std::weak_ptr<StackNode<T>> ptr_;
  };

  template <typename T, typename Allocator = std::allocator<T>>
  class Stack {
  public:
    using allocator_type = typename Allocator::template
rebind<StackNode<T>>::other;

    struct deleter {
      deleter(allocator_type* allocator) : allocator_(allocator) {}

      void operator() (StackNode<T>* ptr) {
        std::allocator_traits<allocator_type >::destroy(*allocator_, ptr);
        allocator_->deallocate(ptr,1);
      }

    private:
      allocator_type* allocator_;
    };

    Stack() {
      StackNode<T>* ptr = allocator_.allocate(1);
      std::allocator_traits<allocator_type >::construct(allocator_, ptr);
      std::shared_ptr<StackNode<T>> new_elem(ptr, deleter(&allocator_));
      tail = new_elem;
      head = tail;
      tail->next = nullptr;
    }

    Stack(const Stack&) = delete;
    Stack(Stack&&) = delete;

    bool Empty() const {
      return head == tail;
    }

    void Pop() {
      if (Empty()){
        throw std::runtime_error("Pop from empty queue");
      }
      if (head->next == tail) {
        head = tail;
```

```
        return;
    }
    std::shared_ptr<StackNode<T>> prev_ptr = tail->prev.lock()->prev.lock();
    prev_ptr->next = tail;
    tail->prev = prev_ptr;
}

void Push(T elem) {
    StackNode<T>* ptr = allocator_.allocate(1);
    std::allocator_traits<allocator_type>::construct(allocator_, ptr);
    std::shared_ptr<StackNode<T>> new_elem(ptr, deleter(&allocator_));
    new_elem->data = std::move(elem);
    if (Empty()) {
        head = new_elem;
        tail->prev = head;
        head->next = tail;
        return;
    }
    std::shared_ptr<StackNode<T>> prev_ptr = tail->prev.lock();
    prev_ptr->next = new_elem;
    tail->prev = new_elem;
    new_elem->next = tail;
    new_elem->prev = prev_ptr;
}

StackIterator<T> begin() {
    return StackIterator<T>(head);
}

StackIterator<T> end() {
    return StackIterator<T>(tail);
}

void Insert(StackIterator<T> iter, T elem) {
    StackNode<T>* ptr = allocator_.allocate(1);
    std::allocator_traits<allocator_type>::construct(allocator_, ptr);
    std::shared_ptr<StackNode<T>> new_elem(ptr, deleter(&allocator_));
    new_elem->data = std::move(elem);
    if (iter == begin()) {
        new_elem->next = head;
        head->prev = new_elem;
        head = new_elem;
    } else {
        std::shared_ptr<StackNode<T>> prev_ptr = iter.ptr_.lock()->prev.lock();
        prev_ptr->next = new_elem;
```

```
            tail->prev = new_elem;
            new_elem->next = tail;
            new_elem->prev = prev_ptr;
        }
    }

    void Erase(StackIterator<T> iter) {
        if (iter == end()) {
            throw std::runtime_error("Erasind end iterator");
        }
        std::shared_ptr<StackNode<T>> ptr = iter.ptr_.lock();
        if (iter == begin()) {
            head = head->next;
            ptr->next = nullptr;
        } else {
            std::shared_ptr<StackNode<T>> prev_ptr = ptr->prev.lock();
            std::shared_ptr<StackNode<T>> next_ptr = ptr->next;
            prev_ptr->next = next_ptr;
            next_ptr->prev = prev_ptr;
        }
    }

  private:
    allocator_type allocator_;
    std::shared_ptr<StackNode<T>> head;
    std::shared_ptr<StackNode<T>> tail;
  };

}
```

Vector_.h

```
#ifndef VECTOR_H_
#define VECTOR_H_

#include "vertex.h"
#include <cmath>
#include <numeric>
#include <limits>

template<class T>
struct Vector {
        explicit Vector(T a, T b);
        double length() const;
        double x;
        double y;
        double operator* (Vector b) ;
```

```
        bool operator== (Vector b);
};

template<class T>
Vector<T>::Vector(T a, T b) {
        x = b.x - a.x;
        y = b.y - a.y;
}

template<class T>
double Vector<T>::length() const{
        return sqrt(x * x + y * y);
}

template<class T>
double Vector<T>::operator* (Vector<T> b) {
        return x * b.x + y * b.y;
}

template<class T>
bool Vector<T>::operator== (Vector<T> b) {
        return std::abs(x - b.x) < std::numeric_limits<double>::epsilon() * 100
        && std::abs(y - b.y) < std::numeric_limits<double>::epsilon() * 100;
}

template<class T>
bool isParallel(const Vector<T> a, const Vector<T> b) {
        return (a.x * b.y - a.y * b.y) == 0;
}

template<class T>
bool isPerpendicular(const Vector<T> a, const Vector<T> b) {
        return (a.x * b.x + a.y * b.y) == 0;
}


#endif
```

Vertex.h

```
#ifndef D_VERTEX_H
#define D_VERTEX_H 1

#include <iostream>

template<class T>
```

```
struct vertex {
        T x;
        T y;

};

template<class T>
std::istream& operator>> (std::istream& is, vertex<T>& p) {
        is >> p.x >> p.y;
        return is;
}

template<class T>
std::ostream& operator<< (std::ostream& os, const vertex<T>& p) {
        os << p.x << ' ' << p.y << '\n';
        return os;
}

template<class T>
vertex<T> operator+(vertex<T> lhs,vertex<T> rhs){
   vertex<T> res;
   res.x = lhs.x + rhs.x;
   res.y = lhs.y + rhs.y;
   return res;
}

template<class T>
bool operator == (vertex<T> a, vertex<T> b) {
        return (a.x == b.x && a.y == b.y);
}

template<class T>
bool operator != (vertex<T> a, vertex<T> b) {
        return (a.x != b.x || a.y != b.y);
}


template<class T>
vertex<T>& operator/= (vertex<T>& vertex, int number) {
   vertex.x = vertex.x / number;
   vertex.y = vertex.y / number;
   return vertex;
}

#endif // D_VERTEX_H
```

Rectangle.h

```
#ifndef D_RECTANGLE_H_
#define D_RECTANGLE_H_ 1

#include <algorithm>
#include <iostream>

#include "vertex.h"
#include "vector_.h"

template<class T>
struct rectangle {
	vertex<T> vertices[4];
	bool existance;

	rectangle(std::istream& is);
	rectangle() = default;

	vertex<double> center() const;

	bool operator==(const rectangle<T>& comp) const;

	double area() const;
	void print() const;
};



template<class T>
rectangle<T>::rectangle(std::istream& is) {
	for(int i = 0; i < 4; ++i){
		is >> vertices[i];
	}

	if (isPerpendicular(Vector< vertex<T> >(vertices[0], vertices[1]), Vector<
vertex<T> >(vertices[0], vertices[3])) && isPerpendicular(Vector< vertex<T>
>(vertices[0], vertices[1]), Vector< vertex<T> >(vertices[1], vertices[2])) &&
		isPerpendicular(Vector< vertex<T> >(vertices[1], vertices[2]),
Vector< vertex<T> >(vertices[2], vertices[3])) && isPerpendicular(Vector<
vertex<T> >(vertices[2], vertices[3]), Vector< vertex<T> >(vertices[0],
vertices[3]))) {
```

```
	} else if (isPerpendicular(Vector< vertex<T> >(vertices[0], vertices[3]),
Vector< vertex<T> >(vertices[3], vertices[1])) && isPerpendicular(Vector<
vertex<T> >(vertices[3], vertices[1]), Vector< vertex<T> >(vertices[1],
vertices[2])) &&
			isPerpendicular(Vector< vertex<T> >(vertices[1], vertices[2]),
Vector< vertex<T> >(vertices[2], vertices[0])) && isPerpendicular(Vector<
vertex<T> >(vertices[0], vertices[2]), Vector< vertex<T> >(vertices[0],
vertices[3]))) {

				vertex<T> tmp;
				tmp = vertices[0];
				vertices[0] = vertices[3];
				vertices[3] = tmp;

	} else if (isPerpendicular(Vector< vertex<T> >(vertices[0], vertices[1]),
Vector< vertex<T> >(vertices[1], vertices[3])) && isPerpendicular(Vector<
vertex<T> >(vertices[1], vertices[3]), Vector< vertex<T> >(vertices[3],
vertices[2])) &&
			isPerpendicular(Vector< vertex<T> >(vertices[3], vertices[2]),
Vector< vertex<T> >(vertices[2], vertices[0])) && isPerpendicular(Vector<
vertex<T> >(vertices[2], vertices[0]), Vector< vertex<T> >(vertices[0],
vertices[1]))) {

				vertex<T> tmp;
				tmp = vertices[2];
				vertices[2] = vertices[3];
				vertices[3] = tmp;

	} else if (vertices[0] == vertices[1] || vertices[0] == vertices[2] || vertices[0]
== vertices[3] || vertices[1] == vertices[2] || vertices[1] == vertices[3] || vertices[2]
== vertices[3]) {
			throw std::logic_error("No points are able to be equal");
	} else {
			throw std::logic_error("That's not a Rectangle, sides are not
Perpendicular");
	}

	if (!(Vector< vertex<T> >(vertices[0], vertices[1]).length() == Vector<
vertex<T> >(vertices[2], vertices[3]).length() && Vector< vertex<T>
>(vertices[1], vertices[2]).length() == Vector< vertex<T> >(vertices[0],
vertices[3]).length())) {
			throw std::logic_error("That's not a Rectangle, sides are not equal");
	}

	existance = true;
```

```
}

template<class T>
double rectangle<T>::area() const {
	if (existance == false) std::logic_error("Object doesn't exist");
	return Vector< vertex<T> >(vertices[0], vertices[1]).length() * Vector<
vertex<T> >(vertices[1], vertices[2]).length();
}

template<class T>
void rectangle<T>::print() const {
	if (existance == true) std::cout << vertices[0] << vertices[1] << vertices[2]
<< vertices[3] << '\n';

}

template<class T>
vertex<double> rectangle<T>::center() const {
	if (existance == false) std::logic_error("Object doesn't exist");
	vertex<double> p;
	p.x = (vertices[0].x + vertices[1].x + vertices[2].x + vertices[3].x) / 4;
	p.y = (vertices[0].y + vertices[1].y + vertices[2].y + vertices[3].y) / 4;
	return p;
}

template<class T>
bool rectangle<T>::operator==(const rectangle<T>& comp) const {
	for (int i = 0; i < 4; i++) {
		if (vertices[i] != comp.vertices[i]) return false;
	}
	return true;
}

template<class T>
std::ostream& operator<< (std::ostream& os, const rectangle<T>& rect) {
	if (rect.existance) os << rect.vertices[0] << rect.vertices[1] <<
rect.vertices[2] << rect.vertices[3];
	return os;
}

#endif // D_TRIANGLE_H_
```

Vector.h

```
#pragma once

#include <memory>
```

```
#include <exception>

namespace Containers {

    template <typename T, typename Allocator>
    class Vector;

    template <typename T>
    class VectorIterator;

    template<typename T, typename Allocator = std::allocator<T>>
    class Vector {
    public:
        friend VectorIterator<T>;

        struct deleter {
            deleter(Allocator* allocator) : allocator_(allocator) {}
            void operator() (T* ptr) {
                if (ptr != nullptr) {
                    std::allocator_traits<Allocator>::destroy(*allocator_,ptr);
                    allocator_->deallocate(ptr, 1);
                }
            }
        private:
            Allocator* allocator_;
        };

        Vector() = default;
        ~Vector() = default;

        Vector(const Vector&) = delete;
        Vector(Vector&&) = delete;

        T &operator[](size_t index) {
            if (index >= size_) {
                throw std::out_of_range("Out of bounds");
            }
            return data_.get()[index];
        }

        const T &operator[](size_t index) const {
            if (index >= size_) {
                throw std::out_of_range("Out of bounds");
            }
            return data_.get()[index];
```

```
    }

    void Resize(size_t new_size) {
      if (new_size == 0) {
        data_ = nullptr;
        return;
      }
      T* ptr = allocator_.allocate(new_size);
      std::allocator_traits<Allocator>::construct(allocator_, ptr);
      std::shared_ptr<T> new_elem(ptr, deleter(&allocator_));
      for (size_t i = 0; i < std::min(new_size, size_); ++i) {
        *(new_elem.get() + i) = *(data_.get() + i);
      }
      data_ = new_elem;
      size_ = new_size;
    }


    VectorIterator<T> begin() {
      return VectorIterator<T>(data_, &size_, 0);
    }

    VectorIterator<T> end() {
      return VectorIterator<T>(data_, &size_, size_);
    }


    size_t Size() const {
      return size_;
    }

  private:
    Allocator allocator_;
    std::shared_ptr<T> data_ = nullptr;
    size_t size_ = 0;
  };




template <typename T>
class VectorIterator {
public:
  using value_type = T;
  using reference = T&;
```

```
    using pointer = T*;
    using difference_type = ptrdiff_t;
    using iterator_category = std::forward_iterator_tag;


    VectorIterator(std::shared_ptr<T> ptr, size_t* size, size_t pos)
            : ptr_(ptr), size_(size), pos_(pos) {}

    T& operator* () {
        std::shared_ptr<T> locked = ptr_.lock();
        if (locked) {
            if (pos_ >= *size_) {
                throw std::logic_error("Wrong operation");
            }
            return locked.get()[pos_];
        } else {
            throw std::runtime_error("Broken iterator");
        }
    }

    bool operator == (const VectorIterator& other) {
        return ptr_.lock() == other.ptr_.lock() && size_ == other.size_ && pos_ ==
other.pos_;
    }

    bool operator != (const VectorIterator& other) {
        return !(*this == other);
    }

    VectorIterator& operator++() {
        if (pos_ + 1 > *size_) {
            throw std::runtime_error("Out of bounds");
        } else {
            pos_++;
        }
        return *this;
    };

private:
    std::weak_ptr<T> ptr_;
    size_t* size_;
    size_t pos_;
};

}
```

Source.cpp

```
#include <iostream>
#include <map>
#include <string>
#include <algorithm>
#include <tuple>
#include <list>

#include "rectangle.h"
#include "Stack.h"
#include "Allocator.h"
#include "Vector.h"
#include "Allocator.h"
#include <map>

void menu() {
	std::cout << "0 : EXIT\n";
	std::cout << "1 : FILL THE VECTOR\n";
	std::cout << "2 : GET ITEM CENTER BY INDEX\n";
	std::cout << "3 : GET AMOUNT OF OBJECTS WITH SQUARE LESS
THAN...\n";
	std::cout << "4 : GO THROUGH VECTOR WITH ITERATOR AND
SHOW EVERY STEP\n";
	std::cout << "5 : CHANGE OBJECT BY INDEX\n";
	std::cout << "6 : RESIZE VECTOR\n";
	std::cout << "> ";
}

int main() {
   std::map<int,int,std::less<>, Allocator<int,100000>> m;

   for (int i = 0; i < 10; ++i) {
		m[i] = i * i;
	}

	m.erase(1);
	m.erase(2);

	int cmd;

	std::cout << "Enter size of your vector : ";
	size_t size;
	std::cin >> size;

	Containers::Vector< rectangle< int > > vec;
```

```
	vec.Resize(size);

	while(true) {

		menu();
		std::cin >> cmd;

		if (cmd == 0) return 0;
		else if (cmd == 1) {

			for (int i = 0; i < vec.Size(); i++) {

				std::cout << "Element number " << i << '\n';
				std::cout << "Enter vertices : \n";
				rectangle<int> rect(std::cin);
				vec[i] = rect;

			}

		} else if (cmd == 2) {

			std::cout << "Enter index : ";
			int index;
			std::cin >> index;
			std::cout << vec[index].center();

		} else if (cmd == 3) {

			int res = 0;
			std::cout << "Enter your square : ";
			double square;
			std::cin >> square;

			int cmdcmd;
			std::cout << "Do you want to use std::count_if? : 1 - yes; 0 - no;
: ";
			std::cin >> cmdcmd;

			if (cmdcmd == 1) res = std::count_if(vec.begin(), vec.end(),
[&square](rectangle<int>& i) {return i.area() < square;});
			else {
				auto it = vec.begin();
				auto end = vec.end();

				while (it != end) {
```

```
                        if ((*it).area() < square) res++;
                        ++it;
                    }
                }

                std::cout << "Amount is " << res << '\n';

            } else if (cmd == 4) {

                int cmdcmd;
                std::cout << "Do you want to use std::for_each? : 1 - yes; 0 - no;
: ";
                std::cin >> cmdcmd;

                if (cmdcmd == 1) std::for_each(vec.begin(), vec.end(),
[](rectangle<int>& i) -> void{i.print();});
                else {
                    auto it = vec.begin();
                    auto end = vec.end();

                    int n = 0;

                    while (it != end) {
                        std::cout << "___OBJECT_" << n << "__\n";
                        std::cout << *it;
                        ++it;
                        n++;
                    }

                }

            } else if (cmd == 5) {

                int index;
                std::cout << "Enter index : ";
                std::cin >> index;

                if (index < 0 || index >= vec.Size()) {

                    std::cout << "Out of range.\n";

                } else {

                    std::cout << "Enter vertices : \n";
                    rectangle<int> rect(std::cin);
```

```
                vec[index] = rect;

            }

        } else if (cmd == 6) {

            int size;
            std::cin >> size;

            if (size < 0) {

                std::cout << "Can't resize to non positive numbers.\n";

            } else {

                vec.Resize(size);

            }

        }

    }

}
```

Объяснение результатов

Программа получает на вход команды из меню. В зависимости от команды совершается одно из действий: заполнение вектора, поолучение центра по индексу, получение количества элементов с площадью меньше данной, проход по вектору с итератором и вывод элементов на экран, изменение элемента вектора по индексу, изменение размера вектора.

Вывод

Выполняя данную лабораторную работу, я получил опыт работы с аллокаторами и умными указателями. Узнал о применении аллокаторов и научился создавать контейнеры, их использующие.