Московский Авиационный Институт
(Национальный исследовательский Университет)

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

Лабораторная работа
по курсу «ООП»

Тема:
Итераторы и аллокаторы.

Студент:	Косогоров В.В.
Группа:	М8О-206Б-18
Преподаватель:	Журавлев А.А.
Вариант:	10
Оценка:	
Дата:	

Москва
2019

1. Код программы:

allocator.h

```
#ifndef D_ALLOCATOR_H_
#define D_ALLOCATOR_H_ 1

#include <cstdlib>
#include <iostream>
#include <type_traits>
#include <list>

#include "list.h"

namespace allocator {

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

        template<class L>
        struct rebind {
            using other = my_allocator<L, ALLOC_SIZE>;
        };

        my_allocator() :
            pool_begin(new char[ALLOC_SIZE]),
            pool_end(pool_begin + ALLOC_SIZE),
            pool_tail(pool_begin)
        {}

        my_allocator(const my_allocator&) = delete;
        my_allocator(my_allocator&&) = delete;

        ~my_allocator() {
            delete[] pool_begin;
        }

        T* allocate(std::size_t n);
        void deallocate(T* ptr, std::size_t n);
```

```
    private:
        char* pool_begin;
        char* pool_end;
        char* pool_tail;
        std::list<char*> free_blocks;
    };

    template<class T, size_t ALLOC_SIZE>
    T* my_allocator<T, ALLOC_SIZE>::allocate(std::size_t n) {
        if (n != 1) {
            throw std::logic_error("Allocating arrays is unavaliable");
        }
        if (size_t(pool_end - pool_tail) < sizeof(T)) {
            if (free_blocks.size()) {
                auto it = free_blocks.begin();
                char* ptr = *it;
                free_blocks.pop_front();
                return reinterpret_cast<T*>(ptr);
            }
            throw std::bad_alloc();
        }
        T* result = reinterpret_cast<T*>(pool_tail);
        pool_tail += sizeof(T);
        return result;
    }

    template<class T, size_t ALLOC_SIZE>
    void my_allocator<T, ALLOC_SIZE>::deallocate(T* ptr, std::size_t n) {
        if (n != 1) {
            throw std::logic_error("Allocating arrays is unavaliable, thus deallocating
is unavalivable as well");
        }
        if (ptr == nullptr) {
            return;
        }
        free_blocks.push_back(reinterpret_cast<char*>(ptr));
    }
};

#endif // D_ALLOCATOR_H_
```

list.h

```
#pragma once
```

```
#include <iterator>
#include <memory>
#include <iostream>

namespace container {

template<class T, class Allocator = std::allocator<T>>
class list {
private:
    struct node_t;
    size_t size = 0;

public:
    struct forward_iterator {
        using value_type = T;
        using reference = T&;
        using pointer = T*;
        using difference_type = ptrdiff_t;
        using iterator_category = std::forward_iterator_tag;

        explicit forward_iterator(node_t* ptr);
        T& operator*();
        forward_iterator& operator++();
        forward_iterator operator++(int);
        bool operator==(const forward_iterator& it) const;
        bool operator!=(const forward_iterator& it) const;
        private:
            node_t* ptr_;
            friend list;
    };

    forward_iterator begin();
     forward_iterator end();
     void push(const T& value);
     void push_b(const T& value);
     T& front();
     T& back();
     void popFront();
     void popBack();
     size_t length();
     bool empty();
     void erase(forward_iterator d_it);
     void erase(size_t N);
     void insert_by_it(forward_iterator ins_it, T& value);
     void insert(size_t N, T& value);
```

```
    list& operator=(list& other);
    T& operator[](size_t index);

private:
   using allocator_type = typename Allocator::template rebind<node_t>::other;

        struct deleter {
        private:
            allocator_type* allocator_;
        public:
            deleter(allocator_type* allocator) : allocator_(allocator) {}

            void operator() (node_t* ptr) {
                if (ptr != nullptr) {
                    std::allocator_traits<allocator_type>::destroy(*allocator_, ptr);
                    allocator_->deallocate(ptr, 1);
                }
            }

        };

   using unique_ptr = std::unique_ptr<node_t, deleter>;

        struct node_t {
            T value;
            unique_ptr next_element = { nullptr, deleter{nullptr} };
            node_t* prev_element = nullptr;
            node_t(const T& value_) : value(value_) {}
            forward_iterator next();
        };

    allocator_type allocator_{};
    unique_ptr head{ nullptr, deleter{nullptr} };
    node_t* tail = nullptr;
};

template<class T, class Allocator>
    typename list<T, Allocator>::forward_iterator list<T, Allocator>::begin() {//+
        return forward_iterator(head.get());
    }

    template<class T, class Allocator>
    typename list<T, Allocator>::forward_iterator list<T, Allocator>::end() {//+
        return forward_iterator(nullptr);
    }
```

```
    template<class T, class Allocator>
    size_t list<T, Allocator>::length() {
        return size;
    }
    template<class T, class Allocator>
    bool list<T, Allocator>::empty() {
        return length() == 0;
    }

    template<class T, class Allocator>
    void list<T, Allocator>::push(const T& value) {
        size++;
        node_t* result = this->allocator_.allocate(1);
        std::allocator_traits<allocator_type>::construct(this->allocator_, result,
value);
        unique_ptr tmp = std::move(head);
        head = unique_ptr(result, deleter{ &this->allocator_ });
        head->next_element = std::move(tmp);
        if(head->next_element != nullptr)
            head->next_element->prev_element = head.get();
        if (size == 1) {
            tail = head.get();
        }
        if (size == 2) {
            tail = head->next_element.get();
        }
    }

    template<class T, class Allocator>
    void list<T, Allocator>::push_b(const T& value) {
        node_t* result = this->allocator_.allocate(1);
        std::allocator_traits<allocator_type>::construct(this->allocator_, result,
value);
        if (!size) {
            head = unique_ptr(result, deleter{ &this->allocator_ });
            tail = head.get();
            size++;
            return;
        }
        tail->next_element = unique_ptr(result, deleter{ &this->allocator_ });
        node_t* temp =  tail;
        tail = tail->next_element.get();
        tail->prev_element = temp;
        size++;
    }
```

```
template<class T, class Allocator>
void list<T, Allocator>::popFront() {
    if (size == 0) {
        throw std::logic_error("Deleting from empty list");
    }
    if (size == 1) {
        head = nullptr;
        tail = nullptr;
        size--;
        return;
    }
    unique_ptr tmp = std::move(head->next_element);
    head = std::move(tmp);
    head->prev_element = nullptr;
    size--;
}

template<class T, class Allocator>
void list<T, Allocator>::popBack() {
    if (size == 0) {
        throw std::logic_error("Deleting from empty list");
    }
    if (tail->prev_element){
        node_t* tmp = tail->prev_element;
        tail->prev_element->next_element = nullptr;
        tail = tmp;
    }
    else{
        head = nullptr;
        tail = nullptr;
    }
    size--;
}

template<class T, class Allocator>
T& list<T, Allocator>::front() {
    if (size == 0) {
        throw std::logic_error("No elements");
    }
    return head->value;
}

template<class T, class Allocator>
```

```
    list<T,Allocator>& list<T, Allocator>::operator=(list<T, Allocator>& other) {
        size = other.size;
        head = std::move(other.head);
    }

    template<class T, class Allocator>
    void list<T, Allocator>::erase(container::list<T, Allocator>::forward_iterator d_it)
{
        if (d_it == this->end()) throw std::logic_error("Out of bounds");
        if (d_it == this->begin()) {
            this->popFront();
            return;
        }
        if (d_it.ptr_ == tail) {
            this->popBack();
            return;
        }
        if (d_it.ptr_ == nullptr) throw std::logic_error("Out of bounds");
        auto temp = d_it.ptr_->prev_element;
        unique_ptr temp1 = std::move(d_it.ptr_->next_element);
        d_it.ptr_ = d_it.ptr_->prev_element;
        d_it.ptr_->next_element = std::move(temp1);
        d_it.ptr_->next_element->prev_element = temp;
        size--;
    }

    template<class T, class Allocator>
    void list<T, Allocator>::erase(size_t N) {
        forward_iterator it = this->begin();
        for (size_t i = 0; i < N; ++i) {
            ++it;
        }
        this->erase(it);
    }

    template<class T, class Allocator>
    void list<T, Allocator>::insert_by_it(container::list<T,
Allocator>::forward_iterator ins_it, T& value) {

        if (ins_it == this->begin()) {
            this->push(value);
            return;
        }
        if(ins_it.ptr_ == nullptr){
            this->push_b(value);
```

```
            return;
        }

        node_t* tmp = this->allocator_.allocate(1);
        std::allocator_traits<allocator_type>::construct(this->allocator_, tmp, value);

        tmp->prev_element = ins_it.ptr_->prev_element;
        ins_it.ptr_->prev_element = tmp;
        tmp->next_element = std::move(tmp->prev_element->next_element);
        tmp->prev_element->next_element = unique_ptr(tmp, deleter{ &this->allocator_ });

        size++;
    }

    template<class T, class Allocator>
    void list<T, Allocator>::insert(size_t N, T& value) {
        forward_iterator it = this->begin();
        if (N >= this->length())
            it = this->end();
        else
        for (size_t i = 0; i < N; ++i) {
            ++it;
        }
        this->insert_by_it(it, value);
    }

    template<class T, class Allocator>
    typename list<T,Allocator>::forward_iterator list<T, Allocator>::node_t::next() {
        return forward_iterator(this->next_element.get());
    }

    template<class T, class Allocator>
    list<T, Allocator>::forward_iterator::forward_iterator(container::list<T, Allocator>::node_t *ptr) {
        ptr_ = ptr;
    }

    template<class T, class Allocator>
    T& list<T, Allocator>::forward_iterator::operator*() {
        return this->ptr_->value;
    }
    template<class T, class Allocator>
    T& list<T, Allocator>::operator[](size_t index) {
        if (index < 0 || index >= size) {
```

```
            throw std::out_of_range("Out of list bounds");
        }
        forward_iterator it = this->begin();
        for (size_t i = 0; i < index; i++) {
            it++;
        }
        return *it;
    }

    template<class T, class Allocator>
    typename list<T, Allocator>::forward_iterator& list<T,
Allocator>::forward_iterator::operator++() {
        if (ptr_ == nullptr) throw std::logic_error("Out of list bounds");
        *this = ptr_->next();
        return *this;
    }

    template<class T, class Allocator>
    typename list<T, Allocator>::forward_iterator list<T,
Allocator>::forward_iterator::operator++(int) {
        forward_iterator old = *this;
        ++*this;
        return old;
    }

    template<class T, class Allocator>
    bool list<T, Allocator>::forward_iterator::operator==(const forward_iterator&
other) const {
        return ptr_ == other.ptr_;
    }

    template<class T, class Allocator>
    bool list<T, Allocator>::forward_iterator::operator!=(const forward_iterator&
other) const {
        return ptr_ != other.ptr_;
    }
}
```

square.h

```
#ifndef D_SQUARE_H_
#define D_SQUARE_H_ 1

#include <algorithm>
#include <iostream>
```

```
#include <cmath>
#include <cassert>

#include "vertex.h"

template<class T>
struct square {
    vertex<T> vertices[4];
    square(std::istream& is);
    square(const vertex<T>& a, const vertex<T>& b, const vertex<T>& c, const
vertex<T>& d);
    vertex<double> center() const;
    double area() const;
    void print(std::ostream& os) const;
};

template<class T>
square<T>::square(std::istream& is) {
    for(int i = 0; i < 4; ++i){
        is >> vertices[i];
    }
    assert(((vertices[1].x - vertices[0].x)*(vertices[3].x - vertices[0].x))+((vertices[1].y
- vertices[0].y)*(vertices[3].y - vertices[0].y)) == 0);
    assert(((vertices[2].x - vertices[1].x)*(vertices[2].x - vertices[3].x))+((vertices[2].y
- vertices[1].y)*(vertices[2].y - vertices[3].y)) == 0);
    assert(((vertices[3].x - vertices[2].x)*(vertices[1].x - vertices[2].x))+((vertices[3].y
- vertices[2].y)*(vertices[1].y - vertices[2].y)) == 0);
    assert((vertices[1].x - vertices[0].x) == (vertices[0].y - vertices[3].y));
    assert((vertices[2].x - vertices[1].x) == (vertices[1].y - vertices[0].y));
    assert((vertices[3].x - vertices[2].x) == (vertices[2].y - vertices[1].y));
}

template<class T>
square<T>::square(const vertex<T>& a, const vertex<T>& b, const vertex<T>& c,
const vertex<T>& d) {
    vertices[0] = a;
    vertices[1] = b;
    vertices[2] = c;
    vertices[3] = d;
}

template<class T>
vertex<double> square<T>::center() const {
    return {(vertices[0].x + vertices[1].x + vertices[2].x + vertices[3].x) * 0.25,
(vertices[0].y + vertices[1].y + vertices[2].y + vertices[3].y) * 0.25};
```

```
}

template<class T>
double square<T>::area() const {
    const T d1 = vertices[0].x - vertices[1].x;
    const T d2 = vertices[3].x - vertices[0].x;
    return abs(d1 * d1) + abs(d2 * d2);
}

template<class T>
void square<T>::print(std::ostream& os) const {
    os << "Square ";
    for(int i = 0; i < 4; ++i){
        os << "[" << vertices[i] << "]";
        if(i + 1 != 4){
        os << " ";
        }
    }
    os << '\n';
}

#endif // D_SQUARE_H_
```

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;
    return os;
}

#endif // D_VERTEX_H_
```

main.cpp

```
#include <iostream>
#include <algorithm>

#include "list.h"
#include "allocator.h"
#include "square.h"

int main() {
    container::list<square<double>, allocator::my_allocator<square<double>, 500>>
list;
    int command, pos;

    while(true) {
        std::cout << std::endl;
        std::cout << "0 - Quit" << std::endl;
        std::cout << "1 - Add element to list (push front / by index)" << std::endl;
        std::cout << "2 - Delete element from list (pop front / erase by index / erase by
iterator)" << std::endl;
        std::cout << "3 - Print all elements" << std::endl;
        std::cout << "4 - Count_if example" << std::endl;
        std::cout << "5 - Print element by [index]" << std::endl << std::endl;
        std::cin >> command;

        if (command == 0) {
            break;

        } else if(command == 1) {
            std::cout << "Enter coordinates" << std::endl;
            square<double> square(std::cin);

            std::cout << "1 - PushFront" << std::endl;
            std::cout << "2 - Insert by index" << std::endl;
            std::cin >> command;
            if(command == 1) {
                list.push(square);
                continue;
            } else if(command == 2) {
```

```
            std::cout << "Enter index" << std::endl;
            std::cin >> pos;
            list.insert(pos, square);
            continue;
        } else {
            std::cout << "Wrong command" << std::endl;
            std::cin >> command;
            continue;
        }

    } else if(command == 2) {
        std::cout << "1 - Erase by index" << std::endl;
        std::cout << "2 - Erase by iterator" << std::endl;
        std::cout << "3 - Pop front" << std::endl;
        std::cin >> command;
        if (command == 1) {
            std::cout << "Enter index" << std::endl;
            std::cin >> pos;
            list.erase(pos);
            continue;
        } else if (command == 2) {
            std::cout << "Enter index" << std::endl;
            std::cin >> pos;
            auto temp = list.begin();
            for(int i = 0; i < pos; ++i) {
                ++temp;
            }
            list.erase(temp);
            continue;

        } else if (command == 3) {
            try {
                list.popFront();
            } catch(std::exception& e) {
                std::cout << e.what() << std::endl;
                continue;
            }
        } else {
            std::cout << "Wrong command" << std::endl;
            std::cin >> command;
            continue;
        }
    } else if(command == 3) {
        for(const auto& item : list) {
            item.print(std::cout);
```

```
            std::cout << "Center: [" << item.center() << "]" << std::endl;
            std::cout << "Area: " << item.area() << std::endl;
            continue;
        }

    } else if(command == 4) {
        std::cout << "Enter required area" << std::endl;
        std::cin >> pos;
        std::cout << "Number of squares with area less than " << pos << " equals ";
        std::cout << std::count_if(list.begin(), list.end(), [pos](square<double>
square) {return square.area() < pos;}) << std::endl;
        continue;

    } else if (command == 5) {
        std::cout << "Enter index to print for" << std::endl;
        std::cin >> pos;
        try {
            list[pos].print(std::cout);
            std::cout << "Center: [" << list[pos].center() << "]" << std::endl;
            std::cout << "Area: " << list[pos].area() << std::endl;
        } catch(std::exception& e) {
            std::cout << e.what() << std::endl;
            continue;
        }
        continue;

    } else {
        std::cout << "Wrong command" << std::endl;
        continue;
    }
  }

  return 0;
}
```

2. Ссылка на репозиторий на GitHub

https://github.com/vladiq/oop_exercise_06

3. Набор testcases.

test_01.test

```
1
```

0 0 0 0 0 0 0 0
1
1
1 1 1 1 1 1 1 1
2
0
1
2 2 2 2 2 2 2 2
2
1
1
3 3 3 3 3 3 3 3
1
3
2
3
3
0

test_02.test

1
0 0 0 0 0 0 0 0
1
1
1 1 1 1 1 1 1 1
2
0
1
2 2 2 2 2 2 2 2
2
1
1
3 3 3 3 3 3 3 3
1
3
2
3
3
2
2
1
3
4
1

5
0
2
3
3
0

4. Результаты выполнения тестов.

test_01.result

Square [3 3] [3 3] [3 3] [3 3]
Center: [3 3]
Area: 0
Square [1 1] [1 1] [1 1] [1 1]
Center: [1 1]
Area: 0
Square [2 2] [2 2] [2 2] [2 2]
Center: [2 2]
Area: 0
Square [0 0] [0 0] [0 0] [0 0]
Center: [0 0]
Area: 0

Square [1 1] [1 1] [1 1] [1 1]
Center: [1 1]
Area: 0
Square [2 2] [2 2] [2 2] [2 2]
Center: [2 2]
Area: 0
Square [0 0] [0 0] [0 0] [0 0]
Center: [0 0]
Area: 0

test_02.result

Square [3 3] [3 3] [3 3] [3 3]
Center: [3 3]
Area: 0
Square [1 1] [1 1] [1 1] [1 1]
Center: [1 1]
Area: 0

Square [2 2] [2 2] [2 2] [2 2]
Center: [2 2]
Area: 0
Square [0 0] [0 0] [0 0] [0 0]
Center: [0 0]
Area: 0

Square [1 1] [1 1] [1 1] [1 1]
Center: [1 1]
Area: 0
Square [2 2] [2 2] [2 2] [2 2]
Center: [2 2]
Area: 0
Square [0 0] [0 0] [0 0] [0 0]
Center: [0 0]
Area: 0

Square [1 1] [1 1] [1 1] [1 1]
Center: [1 1]
Area: 0
Square [0 0] [0 0] [0 0] [0 0]
Center: [0 0]
Area: 0

Enter required area
Number of squares with area less than 1 equals 2

Enter index to print for
Square [1 1] [1 1] [1 1] [1 1]
Center: [1 1]
Area: 0

Square [0 0] [0 0] [0 0] [0 0]
Center: [0 0]
Area: 0

4. Объяснение результатов работы программы.

Пользователь может вставить заданный им квадрат в начало списка или по индексу, удалить элемент из начала списка, по индексу или по итератору. Также реализован метод печати информации об элементе по индексу путём перегрузки оператора [].

5. Вывод.

Выполняя данную лабораторную, я получил опыт работы с итераторами и аллокаторами в C++, а также ознакомился с библиотекой Boost Test, с помощью которой проверил корректность вычисления площади и координат центра для заданного квадрата.