Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа
по курсу «Объектно-ориентированное программирование»
III Семестр

Задание 6
Вариант 8
Основы работы с коллекциями: аллокаторы

Студент:	Жерлыгин М.А
Группа:	М8О-208Б-18
Преподаватель:	Журалвев А.А.
Оценка:	
Дата:	
Подпись:	

Москва 2019

1. Код программы на языке C++

vertex.h:

```
#ifndef VERTEX_H
#define VERTEX_H

#include <iostream>
#include <type_traits>
#include <cmath>


template<class T>
struct vertex {
    T x;
    T y;
    vertex<T>& operator=(const vertex<T>& A);
};

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, vertex<T> p) {
    os << '(' << p.x << ", " << p.y << ')';
    return os;
}

template<class T>
vertex<T> operator+(const vertex<T>& A, const vertex<T>& B) {
    vertex<T> res;
    res.x = A.x + B.x;
    res.y = A.y + B.y;
    return res;
}

template<class T>
vertex<T>& vertex<T>::operator=(const vertex<T>& A) {
    this->x = A.x;
    this->y = A.y;
    return *this;
}

template<class T>
vertex<T> operator+=(vertex<T> &A, const vertex<T> &B) {
    A.x += B.x;
    A.y += B.y;
```

```
    return A;
}

template<class T>
vertex<T> operator/=(vertex<T>& A, const double B) {
    A.x /= B;
    A.y /= B;
    return A;
}

template<class T>
double length(vertex<T>& A, vertex<T>& B) {
    double res = sqrt( pow(B.x - A.x, 2) + pow(B.y - A.y, 2) );
    return res;
}

template<class T>
struct is_vertex : std::false_type {};

template<class T>
struct is_vertex<vertex<T>> : std::true_type {};


#endif //VERTEX_H
```

octagon.h:

```
#ifndef OCTAGON_H_
#define OCTAGON_H_

#include "vertex.h"
#include <iostream>
#include <type_traits>

template <class T>
class Octagon {
public:
    vertex<T> points[8];
    int size = 8;

    Octagon<T>() = default;
    explicit Octagon<T>(std::istream& is) {
        for (auto & point : points) {
            is >> point;
        }
    }

    double area() {
        double result = 0;
        for(int i = 0; i < 7; ++i) {
```

```
            result += (points[i].x * points[i+1].y) - (points[i+1].x * points[i].y);
        }

        result = (result + (points[7].x * points[0].y) - (points[0].x * points[7].y))/2;
        return std::abs(result);
    }

    void print(std::ostream& os) {
        for(int i = 0; i < 8; ++i) {
            os << this->points[i];
            if(i != size - 1) os << ", ";
        }
        os << '\n';
    }

    void operator<< (std::ostream& os) {
        for(int i = 0; i < 8; ++i) {
            os << this->points[i];
            if(i != size - 1) os << ", ";
        }
    }
};

#endif // OCTAGON_H_
```

stack.h:

```
#ifndef STACK_H_
#define STACK_H_

#include <iterator>
#include <memory>
#include <algorithm>

namespace containers {


    template<class T, class Allocator = std::allocator<T>>
    class stack {
    private:
        struct element;
        size_t size = 0;
    public:
        stack() = default;

        class forward_iterator {
        public:
            using value_type = T;
            using reference = T&;
```

```
        using pointer = T*;
        using difference_type = std::ptrdiff_t;
        using iterator_category = std::forward_iterator_tag;
        explicit forward_iterator(element* ptr);
        T& operator*();
        forward_iterator& operator++();
        forward_iterator operator++(int);
        bool operator== (const forward_iterator& other) const;
        bool operator!= (const forward_iterator& other) const;
    private:
        element* iterator_ptr;
        friend stack;
    };

    forward_iterator begin();
    forward_iterator end();
    void push(const T& value);
    T& top();
    void pop();
    void delete_by_it(forward_iterator d_it);
    void delete_by_number(size_t N);
    void insert_by_iterator(forward_iterator ins_it, T& value);
    void insert_by_number(size_t N, T& value);
    size_t Size();
private:

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

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

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

    private:
        allocator_type* allocator_;
    };

    struct element {
        T value;
        std::unique_ptr<element, deleter> next_element {nullptr, deleter{nullptr}};
        element(const T& value_): value(value_) {}
        forward_iterator next();
    };
    allocator_type allocator_{};
    std::unique_ptr<element, deleter> first{nullptr, deleter{nullptr}};
};
```

```
template<class T, class Allocator>
typename stack<T, Allocator>::forward_iterator stack<T, Allocator>::begin() {
    return forward_iterator(first.get());
}

template<class T, class Allocator>
typename stack<T, Allocator>::forward_iterator stack<T, Allocator>::end() {
    return forward_iterator(nullptr);
}

template<class T, class Allocator>
T& stack<T, Allocator>::top() {
    if (size == 0) {
        throw std::logic_error ("Stack empty");
    }
    return first->value;
}

template<class T, class Allocator>
size_t stack<T, Allocator>::Size() {
    return size;
}

template<class T, class Allocator>
void stack<T, Allocator>::delete_by_it(containers::stack<T, Allocator>::forward_iterator d_it) {
    forward_iterator i = this->begin(), end = this->end();
    if (d_it == end) throw std::logic_error ("Out of limit");
    if (d_it == this->begin()) {
        this->pop();
        return;
    }
    while((i.iterator_ptr != nullptr) && (i.iterator_ptr->next() != d_it)) {
        ++i;
    }
    if (i.iterator_ptr == nullptr) throw std::logic_error ("Out of limit");
    i.iterator_ptr->next_element = std::move(d_it.iterator_ptr->next_element);
    size--;
}

template<class T, class Allocator>
void stack<T, Allocator>::delete_by_number(size_t N) {
    forward_iterator it = this->begin();
    for (size_t i = 0; i < N; ++i) {
        if (i == N) break;
        ++it;
    }
    this->delete_by_it(it);
}

template<class T, class Allocator>
void stack<T, Allocator>::insert_by_iterator(containers::stack<T, Allocator>::forward_iterator ins_it, T&
value) {
```

```
        element* tmp = this->allocator_.allocate(1);
        std::allocator_traits<allocator_type>::construct(this->allocator_, tmp, value);
        forward_iterator i = this->begin();
        if (ins_it == this->begin()) {
            tmp->next_element = std::move(first);
            first = std::move(std::unique_ptr<element, deleter> (tmp, deleter{&this->allocator_}));
            size++;
            return;
        }
        while((i.iterator_ptr != nullptr) && (i.iterator_ptr->next() != ins_it)) {
            i++;
        }
        if (i.iterator_ptr == nullptr) throw std::logic_error ("Out of limit");
        tmp->next_element = std::move(i.iterator_ptr->next_element);
        i.iterator_ptr->next_element = std::move(std::unique_ptr<element, deleter> (tmp, deleter{&this-
>allocator_}));
        size++;
    }
    template<class T, class Allocator>
    void stack<T, Allocator>::insert_by_number(size_t N, T& value) {
        forward_iterator it = this->begin();
        for (size_t i = 0; i < N; ++i) {
            if (i == N) break;
            ++it;
        }
        this->insert_by_iterator(it, value);
    }


    template<class T, class Allocator>
    void stack<T, Allocator>::push(const T& value) {
        element* tmp = this->allocator_.allocate(1);
        std::allocator_traits<allocator_type>::construct(this->allocator_, tmp, value);

        if (first == nullptr){
            first = std::unique_ptr<element, deleter> (tmp, deleter{&this->allocator_});
        } else {
            std::swap(tmp->next_element, first);
            first = std::move(std::unique_ptr<element, deleter> (tmp, deleter{&this->allocator_}));
        }
        size++;
    }

    template<class T, class Allocator>
    void stack<T, Allocator>::pop() {
        if (size == 0) {
            throw std::logic_error ("Stack empty");
        }
        auto tmp = std::unique_ptr<element, deleter>(std::move(first->next_element));
        first = std::move(tmp);
        size--;
    }
```

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

    template<class T, class Allocator>
    stack<T, Allocator>::forward_iterator::forward_iterator(containers::stack<T, Allocator>::element *temp)
{
        iterator_ptr = temp;
    }

    template<class T, class Allocator>
    T& stack<T, Allocator>::forward_iterator::operator*() {
        return this->iterator_ptr->value;
    }

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

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

    template<class T, class Allocator>
    bool stack<T, Allocator>::forward_iterator::operator==(const forward_iterator& temp) const {
        return iterator_ptr == temp.iterator_ptr;
    }

    template<class T, class Allocator>
    bool stack<T, Allocator>::forward_iterator::operator!=(const forward_iterator& temp) const {
        return iterator_ptr != temp.iterator_ptr;
    }
}


#endif //STACK_H_
```

my_allocator.h

```
#ifndef MY_ALLOCATOR_H_
#define MY_ALLOCATOR_H_

#include <cstdlib>
```

```
#include <iostream>
#include <type_traits>
#include "containers/stack.h"

namespace allocators {

    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 U>
        struct rebind {
            using other = my_allocator<U, 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;
        containers::stack<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("can`t allocate arrays");
        }
        if (size_t(pool_end - pool_tail) < sizeof(T)) {
            if (free_blocks.Size()) {
                auto it = free_blocks.begin();
                char* ptr = *it;
                free_blocks.pop();
                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("can`t allocate arrays");
    }
    if(ptr == nullptr){
        return;
    }
    free_blocks.push(reinterpret_cast<char*>(ptr));
}

}

#endif // MY_ALLOCATOR_H_
```

main.cpp:

```
#include <iostream>
#include <algorithm>
#include <map>
#include "octagon.h"
#include "containers/stack.h"
#include "my_allocator.h"

int main() {
    size_t n;
    int S;
    char option = 'a';
    containers::stack<Octagon<int>, allocators::my_allocator<Octagon<int>, 800>> s;
    Octagon<int> oct{};
    while (option != '0') {
        std::cout << "> Choose option" << std::endl;
        std:: cin >> option;
        switch (option) {

            case 'm':
                std::cout << "q. Exit\n"
                << "m. Manual\n"
                << "1. Push element in stack\n"
                << "2. Delete element from the stack\n"
                << "3. Delete element from the chosen position\n"
                << "4. Print out stack\n"
                << "5. Print out N of elem., which area < than current value" << std::endl;
```

```
            break;

        case '1': {
            std::cout << "Put your octagon: " << std::endl;
            oct = Octagon<int>(std::cin);
            s.push(oct);
            break;
        }

        case '2': {
            s.pop();
            break;
        }

        case '3': {
            std::cout << "enter position to delete: ";
            std::cin >> n;
            s.delete_by_number(n);
            break;
        }

        case '4': {
            std::for_each(s.begin(), s.end(), [](Octagon<int> &X) {
            X.print(std::cout);
            });
            break;
        }

        case '5': {
            std::cout << "Enter number of area for searching: ";
            std::cin >> S;
            std::cout <<"The number of elements with area < than " << S << ": " << std::count_if(s.begin(),
s.end(), [=](Octagon<int>& X){return X.area() < S;}) << "\n";
            break;
        }

        case '0':
            break;

        default:
            std::cout << "no such option. Try m for man" << std::endl;
            break;

        }
    }
    return 0;
}
```

2. Ссылка на репозиторий на Github

https://github.com/mmaxim2710/oop_exercise_06

3.Набор testcases

1)
1
0 2 1 3 2 3 3 2 3 1 2 0 1 0 0 1
1
1 3 2 5 3 5 4 4 4 2 3 1 2 0 1 2
4
5
2
5
100
3
0
4

2)
1
1 3 2 5 3 5 4 4 4 2 3 1 2 0 1 2
1
0 2 1 3 2 3 3 2 3 1 2 0 1 0 0 1
4
2
4

4. Результат выполнения тестов

1)
(0, 2), (1, 3), (2, 3), (3, 2), (3, 1), (2, 0), (1, 0), (0, 1)
(1, 3), (2, 5), (3, 5), (4, 4), (4, 2), (3, 1), (2, 0), (1, 2)
The number of elements with area < than 2: 0
The number of elements with area < than 100: 2
(1, 3), (2, 5), (3, 5), (4, 4), (4, 2), (3, 1), (2, 0), (1, 2)

2)
(0, 2), (1, 3), (2, 3), (3, 2), (3, 1), (2, 0), (1, 0), (0, 1)
(1, 3), (2, 5), (3, 5), (4, 4), (4, 2), (3, 1), (2, 0), (1, 2)
(1, 3), (2, 5), (3, 5), (4, 4), (4, 2), (3, 1), (2, 0), (1, 2)

5. Объяснение результатов программы

Аллокатор описан в my_allocator.h и используется для выделения памяти. Он совместим с стандартными функциями.

Вывод: Проделав данную работу я ознакомился с аллокаторами, Аллокатор умеет выделять и освобождать память в требуемых количествах определённым образом. Это необходимо для увеличения производительности программы.