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»

Москва, 2019

1. **Тема**: Основы работы с коллекциями: итераторы

2. **Цель работы**: Изучение основ работы с контейнерами, знакомство с концепцией аллокаторов памяти

3. **Задание** (*вариант № 7*):
Фигура — шестиугольник. Контейнер — стек. Аллокатор — стек.

4. **Адрес репозитория на GitHub**

5. **Код программы на С++**

main.cpp

```
#include <iostream>
#include <algorithm>
#include "hexagon.h"
#include "stack.h"
#include "allocator.h"

int main() {
    size_t N;
    float S;
    char option = '0';
    containers::stack<Hexagon<int>, allocators::my_allocator<Hexagon<int>, 800>> q;
    Hexagon<int> rect{};
    while (option != 'q') {
        std::cout << "choose option (m to open man, q to quit)" << std::endl;
        std::cin >> option;
        switch (option) {
        case 'q':
            break;
        case 'm':
            std::cout << "1) push new element into stack\n"
                << "2) insert element into chosen position\n"
                << "3) pop element from the stack\n"
                << "4) delete element from the chosen position\n"
                << "5) print stack\n"
                << "6) count elements with area less then chosen value\n" << std::endl;
            break;
        case '1': {
            std::cout << "enter hexagon (have to enter dots consequently): " << std::endl;
            rect = Hexagon<int>(std::cin);
            q.push(rect);
            break;
        }
        case '2': {
            std::cout << "enter position to insert to: ";
            std::cin >> N;
            std::cout << "enter hexagon: ";
            rect = Hexagon<int>(std::cin);
            q.insert_by_number(N + 1, rect);
            break;
        }
        case '3': {
            q.pop();
            break;
        }
        case '4': {
            std::cout << "enter position to delete: ";
```

```
            std::cin >> N;
            q.delete_by_number(N);
            break;
        }

        case '5': {
            std::for_each(q.begin(), q.end(), [](Hexagon<int>& X) {
X.Printout(std::cout); });
            break;
        }
        case '6': {
            std::cout << "enter max area to search to: ";
            std::cin >> S;
            std::cout << "number of elements with value less than " << S << " is " <<
std::count_if(q.begin(), q.end(), [=](Hexagon<int>& X) {return X.Area() < S; }) <<
std::endl;
            break;
        }
        default:
            std::cout << "no such option. Try m for man" << std::endl;
            break;
        }
    }
    return 0;
}
```

vertex.h

```
#ifndef OOP_LAB5_VERTEX_H
#define OOP_LAB5_VERTEX_H

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


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

template<class T>
double vert_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
```

stack.h

```
#ifndef OOP_EXERCISE_06_STACK_H
#define OOP_EXERCISE_06_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* it_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_it(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>
    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 is empty");
        }
        std::unique_ptr<element,deleter> tmp = std::move(first->next_element);
        first = std::move(tmp);
        size--;
    }

    template<class T, class Allocator>
    T& stack<T, Allocator>::top() {
        if (size == 0) {
            throw std::logic_error("stack is 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 borders");
        if (d_it == this->begin()) {
            this->pop();
            return;
        }
        while ((i.it_ptr != nullptr) && (i.it_ptr->next() != d_it)) {
            ++i;
        }
        if (i.it_ptr == nullptr) throw std::logic_error("out of borders");
        i.it_ptr->next_element = std::move(d_it.it_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 = 1; i <= N; ++i) {
            if (i == N) break;
            ++it;
        }
        this->delete_by_it(it);
    }

    template<class T, class Allocator>
    void stack<T, Allocator>::insert_by_it(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.it_ptr != nullptr) && (i.it_ptr->next() != ins_it)) {
            i++;
        }
        if (i.it_ptr == nullptr) throw std::logic_error("out of borders");
        tmp->next_element = std::move(i.it_ptr->next_element);
        i.it_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 = 1; i <= N; ++i) {
            if (i == N) break;
            ++it;
        }
        this->insert_by_it(it, value);
    }

    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* ptr) {
        it_ptr = ptr;
    }

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

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

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

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

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

```
#endif
```

hexagon.h

```
#ifndef OOP_LAB5_HEXAGON_H
#define OOP_LAB5_HEXAGON_H

#include "vertex.h"

template <class T>
class Hexagon {
public:
    vertex<T> dots[6];

    explicit Hexagon<T>(std::istream& is) {
        for (auto& dot : dots) {
            is >> dot;
        }
    }

    Hexagon<T>() = default;

    double Area() {
        return (0.5 * abs(dots[0].x * dots[1].y + dots[1].x * dots[2].y + dots[2].x *
dots[3].y + dots[3].x * dots[4].y + dots[4].x * dots[5].y + dots[5].x * dots[0].y
            - dots[1].x * dots[0].y - dots[2].x * dots[1].y - dots[3].x * dots[2].y -
dots[4].x * dots[3].y - dots[5].x * dots[4].y - dots[0].x * dots[5].y));
    }

    void Printout(std::ostream& os) {
        for (int i = 0; i < 6; ++i) {
            os << this->dots[i];
            if (i != 5) {
                os << ", ";
            }
        }
        os << std::endl;
    }

    void operator<< (std::ostream& os) {
        for (int i = 0; i < 6; ++i) {
            os << this->dots[i];
            if (i != 5) {
                os << ", ";
            }
        }
    }
};

#endif
```

allocator.h

```
#ifndef OOP_EXERCISE_05_ALLOCATOR_H_
#define OOP_EXERCISE_05_ALLOCATOR_H_

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

#include "stack.h"

namespace allocators {
```

```cpp
    template<class T, size_t a_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, a_size>;
        };

        my_allocator() :
            begin(new char[a_size]),
            end(begin + a_size),
            tail(begin)
        {}

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

        ~my_allocator() {
            delete[] begin;
        }

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

    private:
        char* begin;
        char* end;
        char* tail;
        containers::stack<char*> free_blocks;
    };

    template<class T, size_t a_size>
    T* my_allocator<T, a_size>::allocate(std::size_t n) {
        if (n != 1) {
            throw std::logic_error("can`t allocate arrays");
        }
        if (size_t(end - 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*>(tail);
        tail += sizeof(T);
        return result;
    }

    template<class T, size_t a_size>
    void my_allocator<T, a_size>::deallocate(T* ptr, std::size_t n) {
        if (n != 1) {
            throw std::logic_error("can`t deallocate arrays");
        }
        if (ptr == nullptr) {
            return;
        }
        free_blocks.push(reinterpret_cast<char*>(ptr));
    }
}
```

```
#endif
```

CmakeLists.txt

```
cmake_minimum_required(VERSION 3.10)
project(oop_exercise_06)

set(CMAKE_CXX_STANDARD 17)

add_executable(oop_exercise_06 main.cpp)
```

6. Набор testcases

test_01.txt

```
m
1
0 0     1 0     2 0     2 1    1 1    2 2
5
2
2
2 2     1 2     0 2     0 1    1 1    2 2
5
4
2
5
q
```

test_02.txt

```
m
1
0 0     1 0     2 0     2 1    1 3    2 2
5
1
2 2     1 2     0 2     0 1    1 4    2 2
5
4
1
4
0
5
q
```

7. Объяснение результатов работы программы - вывод

В ходе данной лабораторной работы были получены навыки работы с аллокаторами. Аллокаторы позволяют ускорить быстроействие программ, а так же усилить контроль над менеджментом памяти.