Московский Авиационный Институт
(Национальный исследовательский Университет)

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

Лабораторная работа № 6
по курсу «ООП»

Тема:
Основы работы с коллекциями: итераторы.

Студент:	Козлов А.Д.
Группа:	М80-206Б-18
Преподаватель:	Журавлев А.А.
Вариант:	7
Оценка:	
Дата:	

Москва
2019

1. Код:
stack.hpp:

```
#ifndef CONT_STACK_HPP
#define CONT_STACK_HPP

#include <memory>
#include <exception>

namespace cont {
    template<class T, typename Allocator = std::allocator<T>>
    class stack {
    private:
        struct stack_node;
        struct deleter;
        using allocator_type = typename Allocator::template
rebind<stack_node>::other;
        allocator_type allocator_;
        std::shared_ptr<stack_node> head;
        std::shared_ptr<stack_node> tail;
    public:
        class iterator;
        class const_iterator;
        stack();
        stack(const stack&) = delete;
        stack& operator =(const stack&) = delete;

        bool empty() const;
        void push(const T&);
        void pop();
        T& top();
        size_t size() const;

        iterator begin();
        iterator end();
        const_iterator begin() const;
        const_iterator end() const;

        void insert(iterator, const T&);
        void erase(iterator);
    };

    template<typename T, typename Allocator>
    struct stack<T, Allocator>::stack_node {
```

```
    stack_node() = default;
    stack_node(T new_value) : value(new_value) {}

    T value;
    std::shared_ptr<stack_node> next = nullptr;
    std::weak_ptr<stack_node> prev;
};

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

template<typename T, typename Allocator>
stack<T, Allocator>::stack() {
    stack_node* ptr = allocator_.allocate(1);
    std::allocator_traits<allocator_type>::construct(allocator_, ptr);
    std::shared_ptr<stack_node> new_elem(ptr, deleter(&allocator_));
    head = new_elem;
    tail = head;
}

template<typename T, typename Allocator>
bool stack<T, Allocator>::empty() const {
    return head == tail;
}

template<typename T, typename Allocator>
void stack<T, Allocator>::push(const T& value) {
    stack_node* ptr = allocator_.allocate(1);
    std::allocator_traits<allocator_type>::construct(allocator_, ptr, value);
    std::shared_ptr<stack_node> new_elem(ptr, deleter(&allocator_));
    if(empty()) {
        head = new_elem;
        head->next = tail;
```

```
            tail->prev = head;
        } else {
            new_elem->next = head;
            head->prev = new_elem;
            head = new_elem;
        }
    }
    template<typename T, typename Allocator>
    void stack<T, Allocator>::pop() {
        if(empty()) {
            throw std::out_of_range("Pop from empty stack");
        }
        head = head->next;
    }

    template<typename T, typename Allocator>
    T& stack<T, Allocator>::top() {
        return head->value;
    }

    template<typename T, typename Allocator>
    size_t stack<T, Allocator>::size() const {
        size_t size = 0;
        for(auto i : *this) {
            ++size;
        }
        return size;
    }

    template<typename T, typename Allocator>
    typename stack<T, Allocator>::iterator stack<T, Allocator>::begin() {
        return iterator(head, this);
    }
    template<typename T, typename Allocator>
    typename stack<T, Allocator>::iterator stack<T, Allocator>::end() {
        return iterator(tail, this);
    }
    template<typename T, typename Allocator>
    typename stack<T, Allocator>::const_iterator stack<T, Allocator>::begin() const
{
        return const_iterator(head, this);
    }
    template<typename T, typename Allocator>
```

```
typename stack<T, Allocator>::const_iterator stack<T, Allocator>::end() const {
    return const_iterator(tail, this);
}

template<typename T, typename Allocator>
void stack<T, Allocator>::insert(iterator it, const T& value) {
    if(it.collection != this) {
        throw std::runtime_error("Iterator does not belong to this collection");
    }
    std::shared_ptr<stack_node> it_ptr = it.node.lock();
    if(!it_ptr) {
        throw std::runtime_error("Iterator is corrupted");
    }
    if(it == begin()) {
        push(value);
        return;
    }
    stack_node* ptr = allocator_.allocate(1);
    std::allocator_traits<allocator_type>::construct(allocator_, ptr, value);
    std::shared_ptr<stack_node> new_elem(ptr, deleter(&allocator_));
    if(it == end()) {
        it_ptr->prev.lock()->next = new_elem;
        new_elem->prev = it_ptr->prev;
        new_elem->next = it_ptr;
        it_ptr->prev = new_elem;
    } else {
        std::shared_ptr<stack_node> next_ptr = it_ptr->next;
        std::weak_ptr<stack_node> prev_ptr = it_ptr;
        new_elem->next = next_ptr;
        next_ptr->prev = new_elem;
        new_elem->prev = prev_ptr;
        prev_ptr.lock()->next = new_elem;
    }
}

template<typename T, typename Allocator>
void stack<T, Allocator>::erase(iterator it) {
    if(it.collection != this) {
        throw std::runtime_error("Iterator does not belong to this collection");
    }
    std::shared_ptr<stack_node> it_ptr = it.node.lock();
    if(!it_ptr) {
        throw std::runtime_error("Iterator is corrupted");
```

```
        }
        if(it == end()) {
            throw std::runtime_error("Erase of end iterator");
        }
        if(it == begin()) {
            pop();
        } else {
            std::shared_ptr<stack_node> next_ptr = it_ptr->next;
            std::weak_ptr<stack_node> prev_ptr = it_ptr->prev;
            next_ptr->prev = prev_ptr;
            prev_ptr.lock()->next = next_ptr;
        }
    }

    template<typename T, typename Allocator>
    class stack<T, Allocator>::iterator {
        friend stack<T, Allocator>;
    private:
        std::weak_ptr<stack_node> node;
        const stack<T, Allocator>* collection;
    public:
        using value_type = T;
        using reference = T&;
        using pointer = T*;
        using difference_type = ptrdiff_t;
        using iterator_category = std::forward_iterator_tag;

        iterator(std::shared_ptr<stack_node> init_ptr, const stack<T, Allocator>*
ptr) : node(init_ptr), collection(ptr) {}
        iterator(const iterator& other) : node(other.node), collection(other.collection)
{}

        iterator& operator =(const iterator&);
        bool operator ==(const iterator&) const;
        bool operator !=(const iterator&) const;
        iterator& operator ++();
        iterator operator ++(int);
        T& operator *() const;
    };

    template<typename T, typename Allocator>
    typename stack<T, Allocator>::iterator& stack<T, Allocator>::iterator::operator
=(const iterator& other) {
```

```
        node = other.node;
        return *this;
    }
    template<typename T, typename Allocator>
    bool stack<T, Allocator>::iterator::operator ==(const iterator& other) const {
        auto lhs = node.lock();
        auto rhs = other.node.lock();
        if (lhs && rhs) {
            return lhs.get() == rhs.get();
        }
        return false;
    }
    template<typename T, typename Allocator>
    bool stack<T, Allocator>::iterator::operator !=(const iterator& other) const {
        return !(*this == other);
    }
    template<typename T, typename Allocator>
    typename stack<T, Allocator>::iterator& stack<T, Allocator>::iterator::operator
++() {
        std::shared_ptr<stack_node> tmp = node.lock();
        if(tmp) {
            if(tmp->next == nullptr) {
                throw std::out_of_range("Going out of container boundaries");
            }
            tmp = tmp->next;
            node = tmp;
            return *this;
        } else {
            throw std::runtime_error("Element pointed by this iterator doesnt exist
anymore");
        }
    }
    template<typename T, typename Allocator>
    typename stack<T, Allocator>::iterator stack<T, Allocator>::iterator::operator +
+(int) {
        iterator result(*this);
        ++(*this);
        return result;
    }

    template<typename T, typename Allocator>
    T& stack<T, Allocator>::iterator::operator *() const {
        std::shared_ptr<stack_node> tmp = node.lock();
```

```
        if(tmp) {
            if(tmp->next == nullptr) {
                throw std::runtime_error("Dereferencing of end iterator");
            }
            return tmp->value;
        } else {
            throw std::runtime_error("Element pointed by this iterator doesnt exist
anymore");
        }
    }

    template<typename T, typename Allocator>
    class stack<T, Allocator>::const_iterator {
        friend stack<T, Allocator>;
    private:
        std::weak_ptr<stack_node> node;
        const stack<T, Allocator>* collection;
    public:
        using value_type = T;
        using reference = T&;
        using pointer = T*;
        using difference_type = ptrdiff_t;
        using iterator_category = std::forward_iterator_tag;

        const_iterator(std::shared_ptr<stack_node> init_ptr, const stack<T,
Allocator>* ptr) : node(init_ptr), collection(ptr) {}
        const_iterator(const const_iterator& other) : node(other.node),
collection(other.collection) {}

        const_iterator& operator =(const const_iterator&);
        bool operator ==(const const_iterator&) const;
        bool operator !=(const const_iterator&) const;
        const_iterator& operator ++();
        const_iterator operator ++(int);
        T& operator *() const;
    };

    template<typename T, typename Allocator>
    typename stack<T, Allocator>::const_iterator& stack<T,
Allocator>::const_iterator::operator =(const const_iterator& other) {
        node = other.node;
        return *this;
    }
```

```
    template<typename T, typename Allocator>
    bool stack<T, Allocator>::const_iterator::operator ==(const const_iterator&
other) const {
        auto lhs = node.lock();
        auto rhs = other.node.lock();
        if (lhs && rhs) {
            return lhs.get() == rhs.get();
        }
        return false;
    }
    template<typename T, typename Allocator>
    bool stack<T, Allocator>::const_iterator::operator !=(const const_iterator&
other) const {
        return !(*this == other);
    }
    template<typename T, typename Allocator>
    typename stack<T, Allocator>::const_iterator& stack<T,
Allocator>::const_iterator::operator ++() {
        std::shared_ptr<stack_node> tmp = node.lock();
        if(tmp) {
            if(tmp->next == nullptr) {
                throw std::out_of_range("Going out of container boundaries");
            }
            tmp = tmp->next;
            node = tmp;
            return *this;
        } else {
            throw std::runtime_error("Element pointed by this iterator doesnt exist
anymore");
        }
    }
    template<typename T, typename Allocator>
    typename stack<T, Allocator>::const_iterator stack<T,
Allocator>::const_iterator::operator ++(int) {
        const_iterator result(*this);
        ++(*this);
        return result;
    }

    template<typename T, typename Allocator>
    T& stack<T, Allocator>::const_iterator::operator *() const {
        std::shared_ptr<stack_node> tmp = node.lock();
        if(tmp) {
```

```
            if(tmp->next == nullptr) {
                throw std::runtime_error("Dereferencing of end iterator");
            }
            return tmp->value;
        } else {
            throw std::runtime_error("Element pointed by this iterator doesnt exist
anymore");
        }
    }
}

#endif
```

allocator.hpp:

```
#ifndef CONT_ALLOCATOR_HPP
#define CONT_ALLOCATOR_HPP

#include <exception>
#include "stack.hpp"

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;

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

    allocator(const allocator&) = delete;
    allocator(allocator&&) = delete;

    allocator() {
        size_t object_count = ALLOC_SIZE / sizeof(T);
        memory = reinterpret_cast<char*>(operator new(sizeof(T) * object_count));
        for(size_t i = 0; i < object_count; ++i) {
            free_blocks.push(memory + sizeof(T) * i);
        }
    }
```

```
    ~allocator() {
        operator delete(memory);
    }

    T* allocate(size_t size) {
        if (size > 1) {
            throw std::logic_error("This allocator cant do that");
        }
        if (free_blocks.empty()) {
            throw std::bad_alloc();
        }
        T* temp = reinterpret_cast<T*>(free_blocks.top());
        free_blocks.pop();
        return temp;
    }

    void deallocate(T* ptr, size_t size) {
        if (size > 1) {
            throw std::logic_error("This allocator cant do that");
        }
        free_blocks.push(reinterpret_cast<char*>(ptr));
    }

private:
    cont::stack<char*> free_blocks;
    char* memory;
};

#endif
```

point.hpp:

```
#ifndef T_POINT_HPP
#define T_POINT_HPP

#include <iostream>

template<typename T>
struct point {
    T x;
    T y;
    point<T> operator +(point<T>&);
    point<T> operator -(point<T>&);
};
```

```
template<typename T>
point<T> point<T>::operator +(point<T>& other) {
    return {x + other.x, y + other.y};
}

template<typename T>
point<T> point<T>::operator -(point<T>& other) {
    return {x - other.x, y - other.y};
}

template<typename T>
std::ostream& operator <<(std::ostream& os, const point<T>& p) {
    os << p.x << " " << p.y;
    return os;
}

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

#endif
```

hexagon.hpp:

```
#ifndef T_HEXAGON_HPP
#define T_HEXAGON_HPP

#include <iostream>
#include <exception>

#include "point.hpp"

template<typename T>
class hexagon {
public:
    hexagon() = default;
    hexagon(point<T>&, point<T>&, point<T>&, point<T>&, point<T>&,
point<T>&);
    point<double> center() const;
    double area() const;
    void write(std::ostream&) const;
```

```
    void read(std::istream&);
private:
    point<T> p1, p2, p3, p4, p5, p6;
};

template<typename T>
hexagon<T>::hexagon(point<T>& p1_, point<T>& p2_, point<T>& p3_,
point<T>& p4_, point<T>& p5_, point<T>& p6_)
          : p1(p1_), p2(p2_), p3(p3_), p4(p4_), p5(p5_), p6(p6_) {};

template<typename T>
point<double> hexagon<T>::center() const {
    point<double> res;
    res.x = double(p1.x + p2.x + p3.x + p4.x + p5.x + p6.x) / 6;
    res.y = double(p1.y + p2.y + p3.y + p4.y + p5.y + p6.y) / 6;
    return res;
}

template<typename T>
double hexagon<T>::area() const {
    double A = (p1.x * p2.y + p2.x * p3.y + p3.x * p4.y + p4.x * p5.y + p5.x * p6.y
+ p6.x * p1.y)
        - (p2.x * p1.y + p3.x * p2.y + p4.x * p3.y + p5.x * p4.y + p6.x * p5.y) - (p1.x
* p6.y);
    return A >= 0 ? (A * 0.5) : (-A * 0.5);
}

template<typename T>
void hexagon<T>::write(std::ostream& os) const {
    os << "Hexagon p1: " << p1 << ", p2: "<< p2 << ", p3: "<< p3 << ", p4: "<< p4
<< ", p5: "<< p5 << ", p6: "<< p6;
}

template<typename T>
void hexagon<T>::read(std::istream& is) {
    point<T> p1_, p2_, p3_, p4_, p5_, p6_;
    is >> p1_ >> p2_ >> p3_ >> p4_ >> p5_ >> p6_;
    *this = hexagon(p1_, p2_, p3_, p4_, p5_, p6_);
}

template<typename T>
std::ostream& operator <<(std::ostream& os, const hexagon<T>& hex) {
    hex.write(os);
```

```
    return os;
}

template<typename T>
std::istream& operator >>(std::istream& is, hexagon<T>& hex) {
    hex.read(is);
    return is;
}

#endif
```

main.cpp:

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

#include "../include/hexagon.hpp"
#include "../include/stack.hpp"
#include "../include/allocator.hpp"

int main() {
    std::string command;
    cont::stack<hexagon<int>, allocator<hexagon<int>,1000>> figures;
    while (std::cin >> command) {
        if (command == "menu") {
            std::cout << "1) add\n";
            std::cout << "2) erase\n";
            std::cout << "3) size\n";
            std::cout << "4) print\n";
            std::cout << "5) count\n";
            std::cout << "6) exit\n";
        } else if (command == "help") {
            std::cout << "1) add - add a figure by index\n \t example: add 0 1 1 1 1 1 1
1 1 1 1 1 1\n    It means: add hexagon {{1,1}, {1,1}, {1,1}, {1,1}, {1,1}, {1,1}} to
position 0\n";
            std::cout << "2) erase - erase a figure by index\n\t example: erase 0\n    It
means: erase shape from position 0\n";
            std::cout << "3) size - print size of stack\n";
            std::cout << "4) print - print all shapes in a stack and their area\n";
            std::cout << "5) count - print the number of figures with a given area\n";
        } else if (command == "add") {
            size_t position;
            std::cin >> position;
```

```
            auto it = figures.begin();
            try {
                it = std::next(it, position);
            } catch(std::exception& e) {
                std::cout << "Position is too big\n";
                continue;
            }
            hexagon<int> new_figure;
            try {
                std::cin >> new_figure;
                figures.insert(it, new_figure);
                std::cout << new_figure << "\n";
            } catch (std::exception& ex) {
                std::cout << ex.what() << "\n";
            }

        } else if (command == "erase") {
            size_t index;
            std::cin >> index;
            try {
                auto it = std::next(figures.begin(), index);
                figures.erase(it);
            } catch (...) {
                std::cout << "Index is too big\n";
                continue;
            }
        } else if (command == "size") {
          std::cout << figures.size() << "\n";
        } else if (command == "print") {
            std::for_each(figures.begin(), figures.end(), [] (const hexagon<int>& fig) {
                std::cout << fig << " ";
                std::cout << "Center: " << fig.center() << "\n";
                std::cout << "Area: " << fig.area() << "\n";
            });
        } else if (command == "count") {
            size_t required_area;
            std::cin >> required_area;
            std::cout << std::count_if(figures.begin(), figures.end(), [&required_area]
(const hexagon<int>& fig) {
                return fig.area() < required_area;
            });
            std::cout << "\n";
        } else if (command == "exit") {
```

```
            break;
        } else {
            std::cout << "Incorrect command" << "\n";
            std::cin.ignore(32767, '\n');
        }
    }
    return 0;
}
```

CmakeLists.txt:

```
cmake_minimum_required(VERSION 3.0)
project(oop_exercise_06)
set(CMAKE_CXX_STANDART 17)

set(MAIN ./source/main.cpp)

add_executable(oop_exercise_06 ${MAIN})
```

2. Ссылка на репозиторий на GitHub.

https://github.com/ArtemKD/oop_exercise_06

3. Набор testcases.

test_01.txt:

```
add 0
0 0 0 0 0 0 0 0 0 0 0
add 1
1 1 1 1 1 1 1 1 1 1 1
add 0
2 2 2 2 2 2 2 2 2 2 2
print
size
erase
1
print
```

test_02.txt:

```
add 0
0 0 0 0 0 0 0 0 0 0 0
add 1
1 1 1 1 1 1 1 1 1 1 1
add 2
2 2 2 2 2 2 2 2 2 2 2
add 3
```

3 3 3 3 3 3 3 3 3 3 3 3
add 4
4 4 4 4 4 4 4 4 4 4 4 4
add 2
5 5 5 5 5 5 5 5 5 5 5 5
print
size
erase 3
print

4. Результаты выполнения тестов.

test_01.txt:

Hexagon p1: 0 0, p2: 0 0, p3: 0 0, p4: 0 0, p5: 0 0, p6: 0 0
Hexagon p1: 1 1, p2: 1 1, p3: 1 1, p4: 1 1, p5: 1 1, p6: 1 1
Hexagon p1: 2 2, p2: 2 2, p3: 2 2, p4: 2 2, p5: 2 2, p6: 2 2
Hexagon p1: 2 2, p2: 2 2, p3: 2 2, p4: 2 2, p5: 2 2, p6: 2 2 Center: 2 2 Area: 0
Hexagon p1: 0 0, p2: 0 0, p3: 0 0, p4: 0 0, p5: 0 0, p6: 0 0 Center: 0 0 Area: 0
Hexagon p1: 1 1, p2: 1 1, p3: 1 1, p4: 1 1, p5: 1 1, p6: 1 1 Center: 1 1 Area: 0
3
Hexagon p1: 2 2, p2: 2 2, p3: 2 2, p4: 2 2, p5: 2 2, p6: 2 2 Center: 2 2 Area: 0
Hexagon p1: 1 1, p2: 1 1, p3: 1 1, p4: 1 1, p5: 1 1, p6: 1 1 Center: 1 1 Area: 0

test_02.txt:

Hexagon p1: 0 0, p2: 0 0, p3: 0 0, p4: 0 0, p5: 0 0, p6: 0 0
Hexagon p1: 1 1, p2: 1 1, p3: 1 1, p4: 1 1, p5: 1 1, p6: 1 1
Hexagon p1: 2 2, p2: 2 2, p3: 2 2, p4: 2 2, p5: 2 2, p6: 2 2
Hexagon p1: 3 3, p2: 3 3, p3: 3 3, p4: 3 3, p5: 3 3, p6: 3 3
Hexagon p1: 4 4, p2: 4 4, p3: 4 4, p4: 4 4, p5: 4 4, p6: 4 4
Hexagon p1: 5 5, p2: 5 5, p3: 5 5, p4: 5 5, p5: 5 5, p6: 5 5
Hexagon p1: 0 0, p2: 0 0, p3: 0 0, p4: 0 0, p5: 0 0, p6: 0 0 Center: 0 0 Area: 0
Hexagon p1: 1 1, p2: 1 1, p3: 1 1, p4: 1 1, p5: 1 1, p6: 1 1 Center: 1 1 Area: 0
Hexagon p1: 2 2, p2: 2 2, p3: 2 2, p4: 2 2, p5: 2 2, p6: 2 2 Center: 2 2 Area: 0
Hexagon p1: 5 5, p2: 5 5, p3: 5 5, p4: 5 5, p5: 5 5, p6: 5 5 Center: 5 5 Area: 0
Hexagon p1: 3 3, p2: 3 3, p3: 3 3, p4: 3 3, p5: 3 3, p6: 3 3 Center: 3 3

Area: 0
Hexagon p1: 4 4, p2: 4 4, p3: 4 4, p4: 4 4, p5: 4 4, p6: 4 4 Center: 4 4
Area: 0
6
Hexagon p1: 0 0, p2: 0 0, p3: 0 0, p4: 0 0, p5: 0 0, p6: 0 0 Center: 0 0
Area: 0
Hexagon p1: 1 1, p2: 1 1, p3: 1 1, p4: 1 1, p5: 1 1, p6: 1 1 Center: 1 1
Area: 0
Hexagon p1: 2 2, p2: 2 2, p3: 2 2, p4: 2 2, p5: 2 2, p6: 2 2 Center: 2 2
Area: 0
Hexagon p1: 3 3, p2: 3 3, p3: 3 3, p4: 3 3, p5: 3 3, p6: 3 3 Center: 3 3
Area: 0
Hexagon p1: 4 4, p2: 4 4, p3: 4 4, p4: 4 4, p5: 4 4, p6: 4 4 Center: 4 4
Area: 0

5. Объяснение результатов работы программы.

В программе реализовано меню с пунктами:

1) add - добавление элемента в стек фигур по индексу
2) erase - удаление фигуры по индексу
3) size — размер стека
4) print — вывод всех фигур стека, середины и площади
5) count — вывод кол-ва фигур с заданной площадью, которые лежат в стеке

6. Вывод.

Выполняя данную лабораторную я получил опыт работы с аллокаторами. Создал шаблонный класс stack, hexagon, point и allocator. Реализовал выделение памяти для узлов стека с помощью аллокатора. Реализовал сохранение введенных фигур в стек по индексу, удаление по индексу вывод фигур и их площадей. Также реализовал подсчет кол-ва фигур с заданной площадью.