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: «Основные работы с коллекциями: итераторы»

Группа:	М8О-206Б-18, №14
Студент:	Орозбакиев Э.Д.
Преподаватель:	Журавлёв Андрей Андреевич
Оценка:	
Дата:	

Москва, 2019

1. Задание

Разработать программу на языке C++ согласно варианту задания. Программа на C++ должна собираться с помощью системы сборки CMake. Программа должна получать данные из стандартного ввода и выводить данные в стандартный вывод.

Необходимо настроить сборку лабораторной работы с помощью CMake. Собранная программа должна называться oop_exercise_06 (в случае использования Windows oop_exercise_06.exe)

Необходимо зарегистрироваться на GitHub (если студент уже имеет регистрацию на GitHub то можно использовать ее) и создать репозитарий для задания лабораторной работы.

Преподавателю необходимо предъявить ссылку на публичный репозиторий на Github. Имя репозитория должно быть https://github.com/login/oop_exercise_06

Создать шаблон динамической коллекцию, согласно варианту задания:

1. Коллекция должна быть реализована с помощью умных указателей (std::shared_ptr, std::weak_ptr).

Опционально использование std::unique_ptr;

2. В качестве параметра шаблона коллекция должна принимать тип данных;

3. Коллекция должна содержать метод доступа:

Стек – pop, push, top;

Очередь – pop, push, top;

Список,Динамический массив – доступ к элементу по оператору [];

Реализовать аллокатор, который выделяет фиксированный размер памяти (количество блоков памяти – является параметром шаблона аллокатора). Внутри аллокатор должен хранить указатель на используемый блок памяти и динамическую коллекцию указателей на свободные блоки. Динамическая коллекция должна соответствовать варианту задания (Динамический массив, Список, Стек, Очередь);

Коллекция должна использовать аллокатор для выделеления и освобождения памяти для своих элементов.

Аллокатор должен быть совместим с контейнерами std::map и std::list (опционально – vector).

Реализовать программу, которая: Позволяет вводить с клавиатуры фигуры (с типом int в качестве параметра шаблона фигуры) и добавлять в коллекцию использующую аллокатор; Позволяет удалять элемент из кол-

лекции по номеру элемента; Выводит на экран введенные фигуры с помощью std::for_each;

2. Адрес репозитория на GitHub

https://github.com/p0kemo4ik/oop_exercise_06

3. Код программы на C++

main.cpp

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

#include "list.hpp"
#include "allocator.hpp"
#include "pentagon.hpp"

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

    while(true) {
        std::cout << std::endl;
        std::cout << "1. Добавить фигуру в список"<< std::endl;
        std::cout << "2. Удалить фигуру" << std::endl;
        std::cout << "3. Вывести все фигуры" << std::endl;
        std::cout << "4. Вывести кол-во фигур чья площаль больше чем ..." << std::endl;
        std::cout << "5. Вывести фигуру" << std::endl << std::endl;
        std::cin >> command;

        if (command == 0) {
            break;

        } else if(command == 1) {
            std::cout << "Введите координаты" << std::endl;
            Pentagon<double> pentagon(std::cin);

            std::cout << "1. Добавить фигуру в начало списка" << std::endl;
            std::cout << "2. Добавить фигуру по индексу" << std::endl;
            std::cin >> command;
            if(command == 1) {
                list.push(pentagon);
                continue;
            } else if(command == 2) {
                std::cout << "Введите индекс" << std::endl;
                std::cin >> pos;
                list.insert(pos, pentagon);
                continue;
            } else {
                std::cout << "Неправильная команда" << std::endl;
                std::cin >> command;
```

```
            continue;
        }

    } else if(command == 2) {
        std::cout << "1. Удалить фигуру из списка по индексу" << std::endl;
        std::cout << "2. Удалить по итератору" << std::endl;
        std::cout << "3. Удалить фигуру из начала списка" << std::endl;
        std::cin >> command;
        if (command == 1) {
            std::cout << "Введите индекс" << std::endl;
            std::cin >> pos;
            list.erase(pos);
            continue;
        } else if (command == 2) {
            std::cout << "Введите индекс" << 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 << "Неправильная команда" << 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.square() << std::endl;
            continue;
        }

    } else if(command == 4) {
        std::cout << "Введите площадь" << std::endl;
        std::cin >> pos;
        std::cout << "Количество пятиугольнков площадь, которых меньше заданной "
<< pos;
        std::cout << std::count_if(list.begin(), list.end(), [pos](Pentagon<double> square)
{return square.square() < pos;}) << std::endl;
        continue;
```

```
        } else if (command == 5) {
            std::cout << "Введите номер элемента" << std::endl;
            std::cin >> pos;
            try {
                list[pos].print(std::cout);
                std::cout << "Center: [" << list[pos].center() << "]" << std::endl;
                std::cout << "Area: " << list[pos].square() << std::endl;
            } catch(std::exception& e) {
                std::cout << e.what() << std::endl;
                continue;
            }
            continue;

        } else {
            std::cout << "Неправильная команда" << std::endl;
            continue;
        }
    }

    return 0;
}
```

list.hpp

```
#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_;
    }
}
```

allocator.h

```
#ifndef D_ALLOCATOR_H_
#define D_ALLOCATOR_H_ 1

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

#include "list.hpp"

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_
```

4. Результаты выполнения тестов

1. Добавить фигуру в список

2. Удалить фигуру
3. Вывести все фигуры
4. Вывести кол-во фигур чья площаль больше чем ...
5. Вывести фигуру

1
Введите координаты
0 0 0 0 0 0 0 0 0 0
1. Добавить фигуру в начало списка
2. Добавить фигуру по индексу
1

1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести все фигуры
4. Вывести кол-во фигур чья площаль больше чем ...
5. Вывести фигуру

1
Введите координаты
1 1 1 1 1 1 1 1 1 1
1. Добавить фигуру в начало списка
2. Добавить фигуру по индексу
2
Введите индекс
0

1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести все фигуры
4. Вывести кол-во фигур чья площаль больше чем ...
5. Вывести фигуру

1
Введите координаты
2 2 2 2 2 2 2 2 2 2
1. Добавить фигуру в начало списка
2. Добавить фигуру по индексу
2
Введите индекс
1

1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести все фигуры

4. Вывести кол-во фигур чья площаль больше чем ...
5. Вывести фигуру

1
Введите координаты
3 3 3 3 3 3 3 3 3 3
1. Добавить фигуру в начало списка
2. Добавить фигуру по индексу
1

1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести все фигуры
4. Вывести кол-во фигур чья площаль больше чем ...
5. Вывести фигуру

3
Pentagon: [3, 3] [3, 3] [3, 3] [3, 3] [3, 3]
Center: [[3, 3]]
Area: 0
Pentagon: [1, 1] [1, 1] [1, 1] [1, 1] [1, 1]
Center: [[1, 1]]
Area: 0
Pentagon: [2, 2] [2, 2] [2, 2] [2, 2] [2, 2]
Center: [[2, 2]]
Area: 0
Pentagon: [0, 0] [0, 0] [0, 0] [0, 0] [0, 0]
Center: [[0, 0]]
Area: 0

1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести все фигуры
4. Вывести кол-во фигур чья площаль больше чем ...
5. Вывести фигуру

2
1. Удалить фигуру из списка по индексу
2. Удалить по итератору
3. Удалить фигуру из начала списка
3

1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести все фигуры

4. Вывести кол-во фигур чья площаль больше чем ...
5. Вывести фигуру

3
Pentagon: [1, 1] [1, 1] [1, 1] [1, 1] [1, 1]
Center: [[1, 1]]
Area: 0
Pentagon: [2, 2] [2, 2] [2, 2] [2, 2] [2, 2]
Center: [[2, 2]]
Area: 0
Pentagon: [0, 0] [0, 0] [0, 0] [0, 0] [0, 0]
Center: [[0, 0]]
Area: 0

1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести все фигуры
4. Вывести кол-во фигур чья площаль больше чем ...
5. Вывести фигуру

0

5. Объяснение результатов работы программы

Программа выводит меню, в котором описываются все применимые к фигурам функции – вставка, удаление и вывод фигур из трех различных мест, а также подсчет фигур с площадью большей чем заданное число.

6. Вывод

С помощью пользовательских аллокаторов программист может более эффективно распоряжаться отданной для хранения фигур памятью, сам следить за процессом выделения и очистки памяти, конструирования и деконструирования объектов.