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 4: «Основы метапрограммирования»

Группа:	М8О-206Б-18, №12
Студент:	Кузьмичев Александр Николаевич
Преподаватель:	Журавлёв Андрей Андреевич
Оценка:	
Дата:	20.12.2019

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_04 (в случае использования Windows oop_exercise_04.exe)

Репозиторий должен содержать файлы:

• main.cpp //файл с заданием работы

• CMakeLists.txt // файл с конфигураций CMake

• test_xx.txt // файл с тестовыми данными. Где xx – номер тестового набора 01, 02 , … Тестовых наборов

• report.doc // отчет о лабораторной работе

Разработать шаблоны классов согласно варианту задания. Параметром шаблона должен являться скалярный тип

данных задающий тип данных для оси координат. Классы должны иметь публичные поля. Фигуры являются

Создать набор шаблонов, создающих функции, реализующие:

1. Вычисление геометрического центра фигуры;

2. Вывод в стандартный поток вывода std::cout координат вершин фигуры;

3. Вычисление площади фигуры;

Параметром шаблона должен являться тип класса фигуры (например Square<int>). Помимо самого класса фигуры, шаблонная функция должна уметь работать с tuple. Например, std::tuple<std::pair<int,int>, std::pair<int,int>, std::pair<int,int>> должен интерпретироваться как треугольник. std::tuple<std::pair<int,int>, std::pair<int,int>, std::pair<int,int>, std::pair<int,int>> - как квадрат. Каждый std::pair<int,int> - соответствует координатам вершины фигуры вращения.

Создать программу, которая позволяет:

• Вводить из стандартного ввода std::cin фигуры, согласно варианту задания (как в виде класса, так и в виде

std::tuple).

• Вызывать для нее шаблонные функции (1-3).

При реализации шаблонных функций допускается использование вспомогательных шаблонов std::enable_if, std::tuple_size, std::is_same.

2. Адрес репозитория на GitHub

https://github.com/poisoned-monkey/OOP_labs4_6_7

3. Код программы на C++

main.cpp

```
#include <iostream>
#include <tuple>

#include "vertex.h"
#include "trapeze.h"
#include "rhombus.h"
#include "pentagon.h"
#include "templates.h"

template<class T>
void process() {
    T object;
    read(std::cin, object);
    print(std::cout, object);
    std::cout << square(object) << std::endl;
    std::cout << center(object) << std::endl;
}

int main() {
    std::cout << "Как вы хотите ввести фигуру: " << std::endl;
    std::cout << "1. Кортеж(Tuple)" << std::endl;
    std::cout << "2. Класс" << std::endl;
    int menu, angles, figure;
    std::cin >> menu;
    std::cout << "Сколько углов у фигуры (4, 5): " << std::endl;
    std::cin >> angles;
    switch (menu) {
        case 1 :
            switch (angles) {
                case 4:
                    process<std::tuple<Vertex<double>, Vertex<double>,
Vertex<double>, Vertex<double>>>();
                    break;
                case 5:
                    process<std::tuple<Vertex<double>, Vertex<double>,
Vertex<double>, Vertex<double>, Vertex<double>>>();
                    break;
            }
            break;
        case 2:
            switch (angles) {
                case 4:
                    std::cout << "Введите фигуру: " << std::endl;
                    std::cout << "1. Трапеция" << std::endl;
                    std::cout << "2. Ромб" << std::endl;
                    std::cin >> figure;
                    switch (figure) {
                        case 1:
                            process<Trapeze<double>>();
                            break;
                        case 2:
                            process<Rhombus<double>>();
                            break;
                    }
                    break;
```

```
                case 5:
                    process<Pentagon<double>>();
                    break;
            }
            break;
    }
    system("pause");
    return 0;
}
```

pentagon.h

```
#pragma once
#include<iostream>
#include"vertex.h"
template <class T> class Pentagon{
private:
    Vertex<T> Vertexs[5];
public:
    using vertex_type = Vertex<T>;
    Pentagon();
    Pentagon(std::istream& in);
    Vertex<T> center() const;
    double square() const;
    void read(std::istream& in);
    void print(std::ostream& os) const;
};

template<class T>
Pentagon<T>::Pentagon() {}

template<class T> Pentagon<T>::Pentagon(std::istream& in) {
    for (int i = 0; i < 5; i++)
        in >> Vertexs[i];
}
template<class T> double Pentagon<T>::square() const {
    double Area = 0;
    for (int i = 0; i < 5; i++) {
        Area += (Vertexs[i].x) * (Vertexs[(i + 1) % 5].y) - (Vertexs[(i + 1)
% 5].x)*(Vertexs[i].y);
    }
    Area *= 0.5;
    return abs(Area);
}

template<class T> void Pentagon<T>::print(std::ostream& os) const {
    std::cout << "Pentagon: ";
    for (int i = 0; i < 5; i++)
        std::cout << Vertexs[i] << ' ';
    std::cout << '\n';
}

template<class T> Vertex<T> Pentagon<T>::center() const {
    Vertex<T> res = Vertex<T>();
    for (int i = 0; i < 5; i++)
        res += Vertexs[i];
    return res / 5;
}

template <class T> void Pentagon<T>::read(std::istream& in) {
    Pentagon<T> res = Pentagon(in);
```

```
    *this = res;
}
```

rhombus.h

```
#pragma once
#include<iostream>
#include"vertex.h"
template <class T> class Rhombus {
private:
    Vertex<T> Vertexs[4];
public:
    using vertex_type = Vertex<T>;
    Rhombus();
    Rhombus(std::istream& in);
    Vertex<T> center() const ;
    double square() const;
    void read(std::istream& in);
    void print(std::ostream& os) const;
};

template<class T> Rhombus<T>::Rhombus() {}

template<class T> Rhombus<T>::Rhombus(std::istream& in) {
    for (int i = 0; i < 4; i++)
        in >> Vertexs[i];
}
template<class T> double Rhombus<T>::square() const {
    double Area = 0;
    for (int i = 0; i < 4; i++) {
        Area += (Vertexs[i].x) * (Vertexs[(i + 1) % 4].y) - (Vertexs[(i + 1)
% 4].x)*(Vertexs[i].y);
    }
    Area *= 0.5;
    return abs(Area);
}

template<class T> void Rhombus<T>::print(std::ostream& os) const {
    std::cout << "Rhombus: ";
    for (int i = 0; i < 4; i++)
        std::cout << Vertexs[i] << ' ';
    std::cout << '\n';
}

template<class T> Vertex<T> Rhombus<T>::center() const {
    Vertex<T> res = Vertex<T>();
    for (int i = 0; i < 4; i++)
        res += Vertexs[i];
    return res / 4;
}

template <class T> void Rhombus<T>::read(std::istream& in) {
    Rhombus<T> res = Rhombus<T>(in);
    *this = res;
}
```

templates.h

```
#include<tuple>
#include<type_traits>
```

```cpp
#include"vertex.h"

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

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

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

template<class Head, class... Tail>
struct is_figurelike_tuple<std::tuple<Head, Tail...>> :
std::conjunction<is_vertex<Head>, std::is_same<Head, Tail>...>{};



template<class T>
inline constexpr bool is_figurelike_tuple_v = is_figurelike_tuple<T>::value;

template<class T, class = void>
struct has_method_square : std::false_type {};

template<class T>
struct has_method_square<T,    std::void_t<decltype(std::declval<const
T&>().square())>> :    std::true_type {};

template<class T>
inline constexpr bool has_method_square_v = has_method_square<T>::value;

template<class T>
std::enable_if_t<has_method_square_v<T>, double> square(const T& object) {
    return object.square();
}


template< class T>


double compute_square(const T& tuple) {
    if constexpr(std::tuple_size_v<T> == 4){
        double Area = 0;
        double a1 = (std::get<0>(tuple).x) * (std::get<1>(tuple).y) -
(std::get<1>(tuple).x)*(std::get<0>(tuple).y);
        double a2 = (std::get<1>(tuple).x) * (std::get<2>(tuple).y) -
(std::get<2>(tuple).x)*(std::get<1>(tuple).y);
        double a3 = (std::get<2>(tuple).x) * (std::get<3>(tuple).y) -
(std::get<3>(tuple).x)*(std::get<2>(tuple).y);
        double a4 = (std::get<3>(tuple).x) * (std::get<0>(tuple).y) -
(std::get<0>(tuple).x)*(std::get<3>(tuple).y);
        double res = 0.5 * (a1 + a2 + a3 + a4);
        return res;
    }
    else if constexpr (std::tuple_size_v<T> == 5) {
        double a1 = (std::get<0>(tuple).x) * (std::get<1>(tuple).y) -
(std::get<1>(tuple).x)*(std::get<0>(tuple).y);
        double a2 = (std::get<1>(tuple).x) * (std::get<2>(tuple).y) -
(std::get<2>(tuple).x)*(std::get<1>(tuple).y);
        double a3 = (std::get<2>(tuple).x) * (std::get<3>(tuple).y) -
(std::get<3>(tuple).x)*(std::get<2>(tuple).y);
        double a4 = (std::get<3>(tuple).x) * (std::get<4>(tuple).y) -
```

```
(std::get<4>(tuple).x)*(std::get<3>(tuple).y);
        double res = 0.5 * (a1 + a2 + a3 + a4);
        return res;
    }
    return 0;
}

template<class T>
std::enable_if_t<is_figurelike_tuple_v<T>, double> square(const T& object) {
    if constexpr (std::tuple_size_v<T> < 4 || std::tuple_size_v<T> > 5) {
        return 0;
    }
    else {
        return compute_square(object);
    }
}

//------------------
template<class T, class = void>
struct has_method_center : std::false_type {};

template<class T>
struct has_method_center<T, std::void_t<decltype(std::declval<const
T&>().center())>> : std::true_type {};

template<class T>
inline constexpr bool has_method_center_v = has_method_center<T>::value;

template<class T>
std::enable_if_t<has_method_center_v<T>,
std::void_t<decltype(std::declval<const T&>().Vertexs[0])>> center(const T&
object) {
    return object.center();
}

template<class T>
std::enable_if_t<has_method_center_v<T>, typename T::vertex_type>
center(const T& object) {
    return object.center();
}



template<size_t Id, class T>
std::tuple_element_t<0,T> compute_center(const T& tuple) {
    using vertex_type = std::tuple_element_t<0, T>;
    vertex_type res{};
    if constexpr (std::tuple_size_v<T> == 4) {
        res += std::get<0>(tuple) + std::get<1>(tuple) + std::get<2>(tuple) +
std::get<3>(tuple);
        return res / 4;
    }
    else if (std::tuple_size_v<T> == 5) {
        res = std::get<0>(tuple) + std::get<1>(tuple) + std::get<2>(tuple) +
std::get<3>(tuple) + std::get<4>(tuple);
        return res / 5;
    }
    else
        return res;
}

template<class T>
```

```
std::enable_if_t<is_figurelike_tuple_v<T>, std::tuple_element_t<0,T>>
center(const T& object) {
    using vertex_type = std::tuple_element_t<0, T>;
    vertex_type res{};
    if constexpr (std::tuple_size_v<T> < 4 || std::tuple_size_v<T> > 5) {
        return res;
    }
    else {
        return std::tuple_size_v<T> == 4 ? compute_center<3>(object) :
compute_center<4>(object);
    }
}


template<class T, class = void>
struct has_method_print : std::false_type {};

template<class T>
struct has_method_print<T, std::void_t<decltype(std::declval<const
T&>().print(std::cout))>> : std::true_type {};

template<class T>
inline constexpr bool has_method_print_v = has_method_print<T>::value;

template<class T>
std::enable_if_t<has_method_print_v<T>, void> print(std::ostream& on, const
T& object) {
    object.print(on);
}



template<size_t Id, class T>
void t_print(std::ostream& on, T& tuple) {
    if constexpr (Id >= std::tuple_size_v<T>) {
        return;
    }
    else {
        if constexpr (std::tuple_size_v<T> == 4)
            if(vector_product(std::get<0>(tuple) - std::get<3>(tuple),
std::get<1>(tuple) - std::get<2>(tuple)) == 0) {
                on << "Trapeze: " << std::get<0>(tuple) << ' ' <<
std::get<1>(tuple) << ' ' << std::get<2>(tuple) << ' ' << std::get<3>(tuple)
<< ' ';
            }
            else if constexpr(std::tuple_size_v<T> == 4)
                if(distance(std::get<0>(tuple), std::get<3>(tuple)) ==
distance(std::get<0>(tuple), std::get<1>(tuple)) &&
distance(std::get<0>(tuple), std::get<3>(tuple)) ==
distance(std::get<1>(tuple), std::get<2>(tuple)) &&
distance(std::get<0>(tuple), std::get<3>(tuple)) ==
distance(std::get<2>(tuple), std::get<3>(tuple))) {
                    on << "Rhombus: " << std::get<0>(tuple) << ' ' <<
std::get<1>(tuple) << ' ' << std::get<2>(tuple) << ' ' << std::get<3>(tuple)
<< ' ';
                }
                else if constexpr (std::tuple_size_v<T> == 5) {
                    on << "Pentagon: " << std::get<0>(tuple) << ' ' <<
std::get<1>(tuple)<< ' ' << std::get<2>(tuple) << ' ' << std::get<3>(tuple)
<< ' ' << std::get<4>(tuple);
                }
        return;
```

```
    }
}

template<class T>
std::enable_if_t<is_figurelike_tuple_v<T>, void>
print(std::ostream& in, T& object) {
    if constexpr (std::tuple_size_v<T> < 4 || std::tuple_size_v<T> > 5) {
        return;
    }
    else {
        std::tuple_size_v<T> == 4 ? t_print<3>(std::cout, object) :
t_print<4>(std::cout, object);
    }
}


//------------------
template<class T, class = void>
struct has_method_read : std::false_type {};

template<class T>
struct has_method_read<T,
std::void_t<decltype(std::declval<T&>().read(std::cin))>> : std::true_type
{};

template<class T>
inline constexpr bool has_method_read_v = has_method_print<T>::value;

template<class T>
std::enable_if_t<has_method_read_v<T>, void> read(std::istream& in, T&
object) {
    object.read(in);
}



template<size_t Id, class T>
void t_read(std::istream& in, T& tuple) {
    if constexpr (Id >= std::tuple_size_v<T>) {
        return;
    }
    else {
        if constexpr (std::tuple_size_v<T> == 4) {
            in >> std::get<0>(tuple) >> std::get<1>(tuple) >>
std::get<2>(tuple) >> std::get<3>(tuple);
        }
        else if constexpr (std::tuple_size_v<T> == 5) {
            in >> std::get<0>(tuple) >> std::get<1>(tuple) >>
std::get<2>(tuple) >> std::get<3>(tuple) >> std::get<4>(tuple);
        }
        return;
    }
}

template<class T>
std::enable_if_t<is_figurelike_tuple_v<T>, void>
read(std::istream& in, T& object) {
    if constexpr ((std::tuple_size_v<T> < 4) || (std::tuple_size_v<T> > 5)) {
        return;
    }
    else if constexpr ((std::tuple_size_v<T>) == 4)
        t_read<3>(std::cin, object);
```

```
    else if constexpr ((std::tuple_size_v<T>) == 5)
        t_read<4>(std::cin, object);
}
```

trapeze.h

```
#pragma once
#include<iostream>
#include"vertex.h"
template <class T>
class Trapeze {
private:
    Vertex<T> Vertexs[4];
public:
    using vertex_type = Vertex<T>;
    Trapeze();
    Trapeze(std::istream& in);
    void read(std::istream& in);
    Vertex<T> center() const ;
    double square() const;
    void print(std::ostream& os) const;
};

template<class T> Trapeze<T>::Trapeze() {}

template<class T> Trapeze<T>::Trapeze(std::istream& in) {
    for (int i = 0; i < 4; i++)
        in >> Vertexs[i];
}
template<class T> double Trapeze<T>::square() const {
    double Area = 0;
    for (int i = 0; i < 4; i++) {
        Area += (Vertexs[i].x) * (Vertexs[(i + 1) % 4].y) - (Vertexs[(i + 1)
% 4].x)*(Vertexs[i].y);
    }
    Area *= 0.5;
    return abs(Area);
}

template<class T> void Trapeze<T>::print(std::ostream& os) const {
    std::cout << "Trapeze: ";
    for (int i = 0; i < 4; i++)
        std::cout << Vertexs[i] << ' ';
    std::cout << '\n';
}

template<class T> Vertex<T> Trapeze<T>::center() const {
    Vertex<T> res = Vertex<T>();
    for (int i = 0; i < 4; i++)
        res += Vertexs[i];
    return res / 4;
}

template <class T> void Trapeze<T>::read(std::istream& in) {
    Trapeze<T> res = Trapeze(in);
    *this = res;
}
```

vertex.h

```cpp
#pragma once
#include <cmath>
#include<iostream>
template <class T> class Vertex {
public:
    T x, y;
    Vertex() : x(0), y(0) {};
    Vertex(T _x, T _y) :x(_x), y(_y) {};
    Vertex& operator+=(const Vertex& b) {
        x += b.x;
        y += b.y;
        return *this;
    }
    Vertex& operator-=(const Vertex& b) {
        x -= b.x;
        y -= b.y;
        return *this;
    }
};

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;
}

template<class T> Vertex<T> operator+(const Vertex<T>&a, const Vertex<T>& b)
{
    return Vertex<T>(a.x + b.x, a.y + b.y);
}

template<class T> Vertex<T> operator-(const Vertex<T>& a, const Vertex<T>& b)
{
    return Vertex<T>(a.x - b.x, a.y - b.y);
}

template<class T> Vertex<T> operator/(const Vertex<T>& a, const int b) {
    return Vertex<T>(a.x / b, a.y / b);
}

template <class T> T distance(const Vertex<T>& a, const Vertex<T>& b) {
    return sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2));
}

template <class T> T vector_product(const Vertex<T>& a, const Vertex<T>& b) {
    return a.x*b.y - b.x*a.y;
}
```

4. Результаты выполнения тестов

Test 1:

```
Как вы хотите ввести фигуру:
1. Кортеж(Tuple)
```

```
2. Класс
1
Сколько углов у фигуры (4, 5):
4
0 0
3 2
4 2
5 0
Trapeze: 0 0 3 2 4 2 5 0 -6
3 1
```

Test 2:

```
Как вы хотите ввести фигуру:
1. Кортеж(Tuple)
2. Класс
2
Сколько углов у фигуры (4, 5):
5
1 2
2 4
4 6
6 7
8 9
Pentagon: 1 2 2 4 4 6 6 7 8 9
3.5
4.2 5.6
```

5. Объяснение результатов работы программы

Программа спрашивает, каким образом хранить фигуру, заданную несколькими вершинами, после выводится меню для вычисления площадей фигур. Особенностью программы является то, что функции, которые конструируют, выводят или считают площади могут одновременно работать как с обычными фигурами, заданными через экземпляры класса, так и tuple различной длины.

6. Вывод

Навыки, полученные в ходе выполнения данной лабораторной работы приодятся в реализации различных шаблонных функций с особенными спецификациями, которые будут способны работать с большим кол-вом параметров и аргументов. Также, при выполнении данной лабораторной работы я получил опыт работы с tuple.

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, №12
Студент:	Кузьмичев Александр Николаевич
Преподаватель:	Журавлёв Андрей Андреевич
Оценка:	
Дата:	22.12.2019

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

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

2. Адрес репозитория на GitHub

https://github.com/poisoned-monkey/OOP_labs4_6_7

3. Код программы на C++

main.cpp

```
#include<iostream>
#include<algorithm>
#include<locale.h>
#include"trapeze.h"
#include"containers.h"
#include"allocators.h"

void Menu1() {
   std::cout << "1. Добавить фигуру в список\n";
   std::cout << "2. Удалить фигуру\n";
   std::cout << "3. Вывести фигуру\n";
   std::cout << "4. Вывести все фигуры\n";
   std::cout << "5. Вывести кол-во фигур чья площаль больше чем ...\n";
}

void Push() {
   std::cout << "1. Добавить фигуру в начало списка\n";
   std::cout << "2. Добавить фигуру в конец списка\n";
   std::cout << "3. Добавить фигуру по индексу\n";
}

void Delete() {
   std::cout << "1. Удалить фигуру в начале списка\n";
   std::cout << "2. Удалить фигуру в конце списка\n";
   std::cout << "3. Удалить фигуру по индексу\n";
}

void Print() {
   std::cout << "1. Вывести первую фигуру в списке\n";
   std::cout << "2. Вывести последнюю фигуру в списке\n";
   std::cout << "3. Вывести фигуру по индексу\n";
}

int main() {
   std::cout<<sizeof(Trapeze<int>)<<std::endl;
   setlocale(LC_ALL, "rus");
   containers::list<Trapeze<int>, allocators::my_allocator<Trapeze<int>,
500>> MyList;
   Trapeze<int> TempTrapeze;
   while (true) {
      Menu1();
      int n, m, ind;
      double s;
      std::cin >> n;
      switch (n) {
      case 1:
         TempTrapeze.read(std::cin);
         Push();
         std::cin >> m;
```

```
         switch (m) {
         case 1:
            MyList.push_front(TempTrapeze);
            break;
         case 2:
            MyList.push_back(TempTrapeze);
            break;
         case 3:
            std::cin >> ind;
            MyList.insert_by_number(ind, TempTrapeze);
         default:
            break;
         }
         break;
      case 2:
         Delete();
         std::cin >> m;
         switch (m) {
         case 1:
            MyList.pop_front();
            break;
         case 2:
            MyList.pop_back();
            break;
         case 3:
            std::cin >> ind;
            MyList.delete_by_number(ind);
            break;
         default:
            break;
         }
         break;
      case 3:
         Print();
         std::cin >> m;
         switch (m) {
         case 1:
            MyList.front().print(std::cout);
            std::cout << std::endl;
            break;
         case 2:
            MyList.back().print(std::cout);
            std::cout  << std::endl;
            break;
         case 3:
            std::cin >> ind;
            MyList[ind].print(std::cout);
            std::cout  << std::endl;
            break;
         default:
            break;
         }
         break;
      case 4:
         std::for_each(MyList.begin(), MyList.end(), [](Trapeze<int> &X) {
X.print(std::cout); std::cout << std::endl; });
         break;
      case 5:
         std::cin >> s;
         std::cout << std::count_if(MyList.begin(), MyList.end(),
[=](Trapeze<int>& X) {return X.square() > s; }) << std::endl;
         break;
```

```
      default:
         return 0;
      }
   }
   system("pause");
   return 0;
}
```

vertex.h

```
#pragma once
#include<iostream>
#include<cmath>
template<class T>
class Vertex {
public:
    T x, y;
};

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

template<class T>
std::ostream& operator<<(std::ostream& os, Vertex<T> point) {
    os <<'['<< point.x << ",  " << point.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> operator+=(Vertex<T> &a, const Vertex<T> &b) {
    a.x += b.x;
    a.y += b.y;
    return a;
}

template<class T>
double distance(const Vertex<T> &a, const Vertex<T>& b) {
    return sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2));
}
```

trapeze.h

```
#pragma once
#include"vertex.h"
template <class T>
class Trapeze {
private:
    Vertex<T> Vertexs[4];
public:
```

```
    using vertex_type = Vertex<T>;
    Trapeze();
    Trapeze(std::istream& in);
    void read(std::istream& in);
    Vertex<T> center() const;
    double square() const;
    void print(std::ostream& os) const;
    friend std::ostream& operator<< (std::ostream &out, const Trapeze<T>
&point);
    friend std::ostream& operator>> (std::istream &in, const Trapeze<T>
&point);
};

template<class T> Trapeze<T>::Trapeze() {}

template<class T> Trapeze<T>::Trapeze(std::istream& in) {
    for (int i = 0; i < 4; i++)
        in >> Vertexs[i];
}
template<class T> double Trapeze<T>::square() const {
    double Area = 0;
    for (int i = 0; i < 4; i++) {
        Area += (Vertexs[i].x) * (Vertexs[(i + 1) % 4].y) - (Vertexs[(i + 1)
% 4].x)*(Vertexs[i].y);
    }
    Area *= 0.5;
    return abs(Area);
}

template<class T> void Trapeze<T>::print(std::ostream& os) const {
    os << "Trapeze: ";
    for (int i = 0; i < 4; i++)
        os << Vertexs[i] << ' ';
    os << '\n';
}


template<class T> Vertex<T> Trapeze<T>::center() const {
    Vertex<T> res = Vertex<T>();
    for (int i = 0; i < 4; i++)
        res += Vertexs[i];
    return res / 4;
}

template <class T> void Trapeze<T>::read(std::istream& in) {
    Trapeze<T> res = Trapeze(in);
    *this = res;
}

template<class T>
std::ostream& operator<< (std::ostream &out, const Trapeze<T> &point) {
    out << "Trapeze: ";
    for (int i = 0; i < 4; i++)
        out << point.Vertexs[i] << ' ';
    out << '\n';
}

template<class T>
std::ostream& operator>> (std::istream &in, const Trapeze<T> &point){
    for (int i = 0; i < 4; i++)
        in >> point.Vertexs[i];
}
```

list.h

```
#pragma once
#include <iterator>
#include <memory>


namespace containers {

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

      class forward_iterator {
      public:
         using value_type = T;
         using reference = value_type& ;
         using pointer = value_type* ;
         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 list;
      };

      forward_iterator begin();
      forward_iterator end();
      void push_back(const T& value);
      void push_front(const T& value);
      T& front();
      T& back();
      void pop_back();
      void pop_front();
      size_t length();
      bool empty();
      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);
      list& operator=(list& other);
      T& operator[](size_t index);
   private:
      using allocator_type = typename Allocator::template
rebind<element>::other;

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

     };

     using unique_ptr = std::unique_ptr<element, deleter>;
     struct element {
        T value;
        unique_ptr next_element = { nullptr, deleter{nullptr} };
        element* prev_element = nullptr;
        element(const T& value_) : value(value_) {}
        forward_iterator next();
     };

     allocator_type allocator_{};
     unique_ptr first{ nullptr, deleter{nullptr} };
     element* tail = nullptr;
  };

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

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

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

   template<class T, class Allocator>
   void list<T, Allocator>::pop_back() {
      if (size == 0) {
         throw std::logic_error("can`t pop from empty list");
      }
      if (tail->prev_element){
         element* tmp = tail->prev_element;
         tail->prev_element->next_element = nullptr;
         tail = tmp;
      }
      else{
         first = nullptr;
         tail = nullptr;
      }
      size--;
   }


   template<class T, class Allocator>
   T& list<T, Allocator>::front() {
      if (size == 0) {
         throw std::logic_error("list is empty");
      }
      return first->value;
   }
```

```
   template<class T, class Allocator>
   T& list<T, Allocator>::back() {
      if (size == 0) {
         throw std::logic_error("list is empty");
      }
      forward_iterator i = this->begin();
      while ( i.it_ptr->next() != this->end()) {
         i++;
      }
      return *i;
   }
   template<class T, class Allocator>
   list<T,Allocator>& list<T, Allocator>::operator=(list<T, Allocator>&
other) {
      size = other.size;
      first = std::move(other.first);
   }

   template<class T, class Allocator>
   void list<T, Allocator>::delete_by_it(containers::list<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_front();
         return;
      }
      if (d_it.it_ptr == tail) {
         this->pop_back();
         return;
      }

      if (d_it.it_ptr == nullptr) throw std::logic_error("out of broders");
      auto temp = d_it.it_ptr->prev_element;
      unique_ptr temp1 = std::move(d_it.it_ptr->next_element);
      d_it.it_ptr->prev_element->next_element = std::move(temp1);
      d_it.it_ptr = d_it.it_ptr->prev_element;
      d_it.it_ptr->next_element->prev_element = temp;

      size--;
   }

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

   template<class T, class Allocator>
   void list<T, Allocator>::insert_by_it(containers::list<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()) {
         this->push_front(value);
         return;
```

```
      }
      if(ins_it.it_ptr == nullptr){
         this->push_back(value);
         return;
      }

      tmp->prev_element = ins_it.it_ptr->prev_element;
      ins_it.it_ptr->prev_element = tmp;
      tmp->next_element = unique_ptr(ins_it.it_ptr, deleter{ &this-
>allocator_ });
      tmp->prev_element->next_element = unique_ptr(tmp, deleter{ &this-
>allocator_ });

      size++;
   }

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

   template<class T, class Allocator>
   list<T, Allocator>::forward_iterator::forward_iterator(containers::list<T,
Allocator>::element *ptr) {
      it_ptr = ptr;
   }

   template<class T, class Allocator>
   T& list<T, Allocator>::forward_iterator::operator*() {
      return this->it_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's borders");
      }
      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 (it_ptr == nullptr) throw std::logic_error("out of list borders");
      *this = it_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 it_ptr == other.it_ptr;
    }

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

allocator.h

```
#pragma once

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

namespace allocators {

    template<class T, size_t ALLOC_SIZE>//ALLOC_SIZE – размер, который
требуется выделить
    struct my_allocator {

    private:
        char* pool_begin; //указатель на начало хранилища
        char* pool_end;//указатель на конец хранилища
        char* pool_tail;//указатель на конец заполненного пространства
        std::queue<char*> free_blocks;
    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 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);

    };

    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()) {//ищем свободное место в райне отданном
пространстве
                char* ptr = free_blocks.front();
                free_blocks.pop();
                return reinterpret_cast<T*>(ptr);
            }
            std::cout<<"Bad Alloc"<<std::endl;
            throw std::bad_alloc();
        }
        T* result = reinterpret_cast<T*>(pool_tail);//приведение к типу
        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, thus can`t
deallocate them too");
        }
        if (ptr == nullptr) {
            return;
        }
        free_blocks.push(reinterpret_cast<char*>(ptr));
    }

}
```

quenue.h

```
#pragma once

#include <memory>

#include "allocators.h"

namespace oop
{
    template<typename Q>
    class queue_forward_iterator
    {

    };

    template <typename T, typename TBaseAllocator = std::allocator<T>>
```

```
    class queue
    {

        struct node;

        using allocator = typename
std::allocator_traits<TBaseAllocator>::template rebind_alloc<node>;


        struct deleter
        {
            explicit deleter(allocator& al) noexcept
                : al_(al)
            {}

            void operator()(node* ptr)
            {
                std::allocator_traits<allocator>::destroy(al_, ptr);
                al_.deallocate(ptr, 1);
            }

        private:
            allocator& al_;
        };


        struct node
        {
            T                                value;
            std::shared_ptr<node> next;

            explicit node(const T& v, deleter& d) noexcept
                : value(v)
                , next(nullptr, d)
            {}
        };

    public:
        queue()
            : deleter_(al_)
            , first_(nullptr, deleter_)
            , last_(nullptr)
            , size_(0)
        {}

        void pop()
        {
            if (first_)
            {
                first_ = std::move(first_->next);
            }

            --size_;
        }

        void push(const T& v)
        {
            node* obj = al_.allocate(1);
            std::allocator_traits<allocator>::construct(al_, obj, v,
deleter_);
            if (last_)
            {
```

```
                last_->next.reset(obj);
            }
            else
            {
                first_.reset(obj);
            }
            last_ = obj;

            ++size_;
        }

        [[nodiscard]] T& top()
        {
            return first_->value;
        }

        [[nodiscard]] size_t size() const noexcept
        {
            return size_;
        }

    private:
        allocator al_;
        deleter   deleter_;

        std::shared_ptr<node> first_;
        node*                 last_;
        size_t                size_;

        friend struct node;
    };
}
```

4. Результаты выполнения тестов

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

1
1 0
0 0
0 0
0 0
1. Добавить фигуру в начало списка
2. Добавить фигуру в конец списка
3. Добавить фигуру по индексу
1
1. Добавить фигуру в список
2. Удалить фигуру

3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...
1
2 0
0 0
0 0
0 0
1. Добавить фигуру в начало списка
2. Добавить фигуру в конец списка
3. Добавить фигуру по индексу
2
1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...
1
1 2
2 3
3 4
5 0
1. Добавить фигуру в начало списка
2. Добавить фигуру в конец списка
3. Добавить фигуру по индексу
3
1
1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...
4
Trapeze: [1, 0] [0, 0] [0, 0] [0, 0]

Trapeze: [1, 2] [2, 3] [3, 4] [5, 0]

Trapeze: [2, 0] [0, 0] [0, 0] [0, 0]

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

3 0
0 0
0 0
0 0
1. Добавить фигуру в начало списка
2. Добавить фигуру в конец списка
3. Добавить фигуру по индексу
2
1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...
4
Trapeze: [1, 0] [0, 0] [0, 0] [0, 0]

Trapeze: [1, 2] [2, 3] [3, 4] [5, 0]

Trapeze: [2, 0] [0, 0] [0, 0] [0, 0]

Trapeze: [3, 0] [0, 0] [0, 0] [0, 0]

1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...
5
0
1
1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...
2
1. Удалить фигуру в начале списка
2. Удалить фигуру в конце списка
3. Удалить фигуру по индексу
1
1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...

2
1. Удалить фигуру в начале списка
2. Удалить фигуру в конце списка
3. Удалить фигуру по индексу
2
1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...
2
1. Удалить фигуру в начале списка
2. Удалить фигуру в конце списка
3. Удалить фигуру по индексу
31
1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...
4
Trapeze: [1, 2] [2, 3] [3, 4] [5, 0]

Trapeze: [2, 0] [0, 0] [0, 0] [0, 0]

Trapeze: [3, 0] [0, 0] [0, 0] [0, 0]

1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...
2
1. Удалить фигуру в начале списка
2. Удалить фигуру в конце списка
3. Удалить фигуру по индексу
3
1
1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...
4
Trapeze: [1, 2] [2, 3] [3, 4] [5, 0]

Trapeze: [2, 0] [0, 0] [0, 0] [0, 0]

Trapeze: [3, 0] [0, 0] [0, 0] [0, 0]

1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...
2
1. Удалить фигуру в начале списка
2. Удалить фигуру в конце списка
3. Удалить фигуру по индексу
1
1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...
4
Trapeze: [2, 0] [0, 0] [0, 0] [0, 0]

Trapeze: [3, 0] [0, 0] [0, 0] [0, 0]

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

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

4
Trapeze: [3, 0] [0, 0] [0, 0] [0, 0]

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

5. Объяснение результатов работы программы

Программа выводит меню со всеми применимыми к фигурам функциями – вставкой, удалением и выводом фигур из трех различных мест, а также подсчетом фигур с площадью большей чем заданное число. Функционально программа не изменилась, однако для реализованного ранее списка был написан аллокатор, который более грамотно распоряжается памятью, отведенной для хранения списка фигур.

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.

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 7: «Проектирование структуры классов»

Группа:	М8О-206Б-18, №12
Студент:	Кузьмичев Александр Николаевич
Преподаватель:	Журавлёв Андрей Андреевич
Оценка:	
Дата:	22.02.2020

Москва, 2020

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_07 (в случае использования Windows oop_exercise_07.exe)

Спроектировать простейший графический векторный редактор.

Требование к функционалу редактора:

• создание нового документа

• импорт документа из файла

• экспорт документа в файл

• создание графического примитива (согласно варианту задания)

• удаление графического примитива

• отображение документа на экране (печать перечня графических объектов и их характеристик)

• реализовать операцию undo, отменяющую последнее сделанное действие. Должно действовать для операций добавления/удаления фигур.

Требования к реализации:

• Создание графических примитивов необходимо вынести в отдельный класс – Factory.

• Сделать упор на использовании полиморфизма при работе с фигурами;

• Взаимодействие с пользователем (ввод команд) реализовать в функции main;

2. Адрес репозитория на GitHub

https://github.com/poisoned-monkey/OOP_labs4_6_7

3. Код программы на C++

Figure.h

```
#pragma once
#include"sdl.h"
#include<iostream>


struct color {
```

```
    color(): r(255), g(0), b(0) {}
    int32_t r, g, b;
    color(int r_, int g_, int b_) :r(r_), g(g_), b(b_) {}
    void set_color(int r_, int g_, int b_) { r = r_, g = g_, b = b_; }
};
struct figure {
    virtual void render(const sdl::renderer& renderer) const = 0;
    virtual void save(std::ostream& os) const = 0;
    virtual ~figure() = default;


    color color_{};
    virtual void set_color(int r, int g, int b) {
        color_.r = r;
        color_.g = g;
        color_.b = b;
    }
    virtual void set_color(color c) {
        color_.r = c.r;
        color_.g = c.g;
        color_.b = c.b;
    }
};


struct vertex {
    int32_t x, y;
};
```

Figures.h

```
#pragma once
#include"pentagon.h"
#include"rhombus.h"
#include"trapeze.h"
#include"line.h"
#include"curve_line.h"
```

Main.cpp

```
#include <array>
#include <fstream>
#include <iostream>
#include <memory>
#include <vector>

#include "sdl.h"
#include "lib/imgui/imgui.h"
#include "tools.h"
#include "loader.h"
#include "canvas.h"


int main() {
    canvas canv;
    color figure_color{};
    sdl::renderer renderer("Editor");
    bool quit = false;
    std::unique_ptr<builder> active_builder = nullptr;
    const int32_t file_name_length = 128;
    char file_name[file_name_length] = "";
    int32_t remove_id = 0;
    while(!quit){
```

```
        renderer.set_color(0, 0, 0);
        renderer.clear();

        sdl::event event;

        while(sdl::event::poll(event)){
            sdl::quit_event quit_event;
            sdl::mouse_button_event mouse_button_event;
            if(event.extract(quit_event)){
                quit = true;
                break;
            }else if(event.extract(mouse_button_event)){
                if(active_builder && mouse_button_event.button() ==
sdl::mouse_button_event::left &&
                   mouse_button_event.type() ==
sdl::mouse_button_event::down){
                    std::unique_ptr<figure> figure =
                            active_builder-
>add_vertex(vertex{mouse_button_event.x(), mouse_button_event.y()});
                    if(figure){
                        (*figure).set_color(figure_color);
                        canv.add_figure(std::move(figure));
                        active_builder = nullptr;
                    }
                }
            }
        }

        for (int i = 0; i < canv.figures.size(); ++i) {
            if(canv.figures[i])
                canv.figures[i]->render(renderer);
        }

        ImGui::Begin("Menu");
        if (ImGui::Button("New canvas")) {
            canv.figures.clear();
        }
        ImGui::InputText("File name", file_name, file_name_length - 1);
        if(ImGui::Button("Save")){
            std::ofstream os(file_name);
            if(os){
                for (int i = 0; i < canv.figures.size();++i) {
                    canv.figures[i]->save(os);
                }
            }
        }
        ImGui::SameLine();
        if (ImGui::Button("Load")) {
            std::ifstream is(file_name);
            if (is) {
                loader loader;
                canv.figures = loader.load(is);
            }
        }

        ImGui::InputInt("R", &figure_color.r);
        ImGui::InputInt("G", &figure_color.g);
        ImGui::InputInt("B", &figure_color.b);
        if (ImGui::Button("Red")) {
            figure_color.set_color(255, 0, 0);
        }
        ImGui::SameLine();
```

```
        if (ImGui::Button("Green")) {
            figure_color.set_color(0, 255, 0);
        }
        ImGui::SameLine();
        if (ImGui::Button("Blue")) {
            figure_color.set_color(0, 0, 255);
        }

        if (ImGui::Button("Line")) {
            active_builder = std::make_unique<line_builder>();
        }
        ImGui::SameLine();
        if(ImGui::Button("Trapeze")){
            active_builder = std::make_unique<trapeze_builder>();
        }

        ImGui::SameLine();
        if (ImGui::Button("Rhombus")) {
            active_builder = std::make_unique<rhombus_builder>();
        }
        ImGui::SameLine();
        if (ImGui::Button("Pentagon")) {
            active_builder = std::make_unique<pentagon_builder>();
        }
        ImGui::SameLine();
        if (ImGui::Button("Curve Line")) {
            active_builder = std::make_unique<curve_line_builder>();
        }


        ImGui::InputInt("Remove id", &remove_id);
        if(ImGui::Button("Remove")){
            if (remove_id < canv.figures.size()) {
                if(canv.figures[remove_id])
                    canv.remove_figure(remove_id);
            }
        }

        if (ImGui::Button("Undo")) {
            canv.undo();
        }
        ImGui::End();

        renderer.present();
    }
}
```

Pentagon.h

```
#pragma once
#include"tools.h"
#include"figure.h"
#include"journal.h"
#include<memory>
struct  pentagon : figure {
    pentagon() {}
    pentagon(const std::array<vertex, 5>& vertices) : vertices_(vertices) {}

    void render(const sdl::renderer& renderer) const override {
        renderer.set_color(color_.r, color_.g, color_.b);
        for (int32_t i = 0; i < 5; ++i) {
            renderer.draw_line(vertices_[i].x, vertices_[i].y,
```

```
                                vertices_[(i + 1) % 5].x, vertices_[(i + 1) %
5].y);
        }
    }

    void save(std::ostream& os) const override {
        os << "pentagon" << std::endl;
        for (int32_t i = 0; i < 5; ++i) {
            os << vertices_[i].x << ' ' << vertices_[i].y << '\n';
        }

        os << this->color_.r << ' ' << this->color_.g << ' ' << this-
>color_.b << std::endl;
    }

private:
    std::array<vertex, 5> vertices_;

};


struct  pentagon_builder : builder {
    std::unique_ptr<figure> add_vertex(const vertex& v) {
        vertices_[n_] = v;
        n_ += 1;
        if (n_ != 5) {
            return nullptr;
        }
        jl.push(1, nullptr);
        return std::make_unique<pentagon>(vertices_);
    }
private:
    int32_t n_ = 0;
    std::array<vertex, 5> vertices_;
};
```

Rhombus.h

```
#pragma once
#include"tools.h"
#include"figure.h"
#include<memory>
struct  rhombus : figure {
    rhombus(const std::array<vertex, 4>& vertices) : vertices_(vertices) {}

    void render(const sdl::renderer& renderer) const override {
        renderer.set_color(color_.r, color_.g, color_.b);
        for (int32_t i = 0; i < 4; ++i) {
            renderer.draw_line(vertices_[i].x, vertices_[i].y,
                               vertices_[(i + 1) % 4].x, vertices_[(i + 1) %
4].y);
        }
    }

    void save(std::ostream& os) const override {
        os << "rhombus" << std::endl;
        for (int32_t i = 0; i < 4; ++i) {
            os << vertices_[i].x << ' ' << vertices_[i].y << '\n';
        }

        os << this->color_.r << ' ' << this->color_.g << ' ' << this-
```

```
>color_.b << std::endl;
    }

private:
    std::array<vertex, 4> vertices_;

};


struct  rhombus_builder : builder {
    std::unique_ptr<figure> add_vertex(const vertex& v) {
        vertices_[n_] = v;
        n_ += 1;
        if (n_ != 4) {
            return nullptr;
        }
        jl.push(1, nullptr);
        return std::make_unique<rhombus>(vertices_);
    }

private:
    int32_t n_ = 0;
    std::array<vertex, 4> vertices_;

};
```

Tools.h

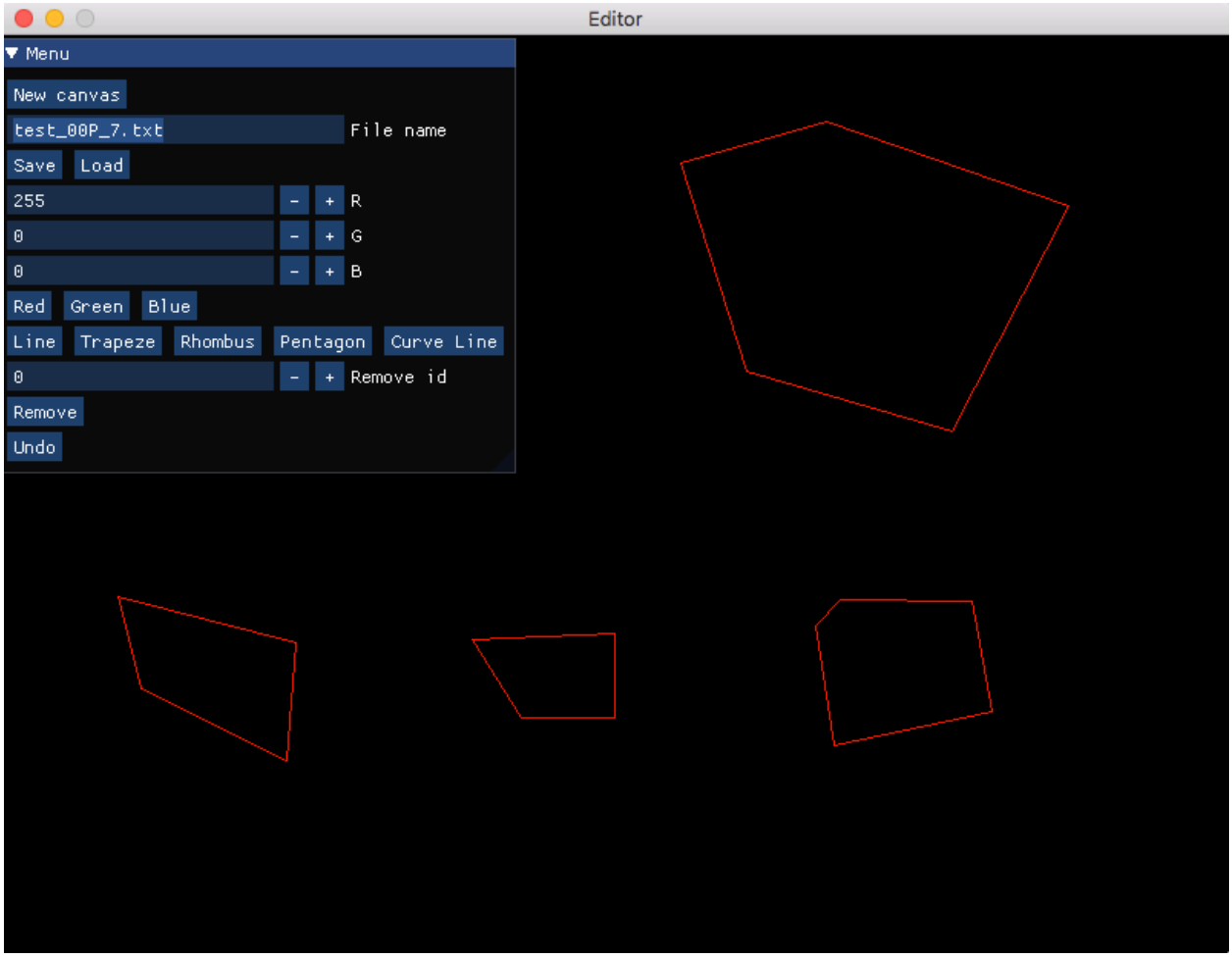
```
#pragma once
#include<cmath>
#include<memory>
#include<stack>
#include<string>
#include<iostream>
#include<vector>
#include<array>
#include"sdl.h"
#include"figures/figure.h"


struct builder {
    virtual std::unique_ptr<figure> add_vertex(const vertex& v) = 0;

    virtual ~builder() = default;

};
```

Trapeze.h

```
#pragma once
#include"tools.h"
#include"figure.h"
#include<memory>
struct  trapeze : figure {
    trapeze(const std::array<vertex, 4>& vertices) : vertices_(vertices) {}

    void render(const sdl::renderer& renderer) const override {
        renderer.set_color(color_.r, color_.g, color_.b);
        for (int32_t i = 0; i < 4; ++i) {
            renderer.draw_line(vertices_[i].x, vertices_[i].y,
                               vertices_[(i + 1) % 4].x, vertices_[(i + 1) %
4].y);
```

```
        }
    }

    void save(std::ostream& os) const override {
        os << "trapeze" << std::endl;
        for (int32_t i = 0; i < 4; ++i) {
            os << vertices_[i].x << ' ' << vertices_[i].y << '\n';
        }

        os << this->color_.r << ' ' << this->color_.g << ' ' << this-
>color_.b << std::endl;
    }

private:
    std::array<vertex, 4> vertices_;

};



struct  trapeze_builder : builder {
    std::unique_ptr<figure> add_vertex(const vertex& v) {
        vertices_[n_] = v;
        n_ += 1;
        if (n_ != 4) {
            return nullptr;
        }
        jl.push(1, nullptr);
        return std::make_unique<trapeze>(vertices_);
    }

private:
    int32_t n_ = 0;
    std::array<vertex, 4> vertices_;

};
```

4. Результаты выполнения тестов



Test_OOP_7.txt

trapeze
75 366
90 426
185 473
191 396
255 0 0
rhombus
306 394
338 445
399 445
399 390
255 0 0
pentagon
530 385
542 463
645 441
632 369
546 368
255 0 0

pentagon
485 219
619 258
695 111
537 56
442 83
255 0 0

5. Объяснение результатов работы программы

Программа запускается с появлением черного полотна и меню с функциями для рисования и удаления различных фигур. Пользователь может нарисовать сразу несколько видов фигур а затем удалить, либо сохранить их. Также есть возможность отмены прошлых действий при помощи кнопки undo. Внутри программы было реализовано несколько классов для построения, рендеринга и сохранения фигур, журнал изменения состояния полотна.

6. Вывод

Выполнив данную лабораторную работу студент может улучшить свои навыки в проектировании более сложных программ. Умение проектировать структуру классов позволяет сделать дальнейшую разработку более гибкой и простой, повысить читаемость кода. Кроме того, в ходе выполнения работы студент может познакомится с графическими библиотеками и написать свой пользовательский интерфейс. Мне было крайне интересно работать с SDL.