Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа**

**по курсу «Объектно-ориентированное программирование»**

**III Семестр**

**Задание 4  
Вариант 8**

**Основы метапрограммирования**

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

# 1. Код программы на языке С++

**vertex.h:**

#ifndef VERTEX\_H

#define VERTEX\_H

#include <iostream>

#include <type\_traits>

#include <cmath>

template<class T>

struct vertex {

T x;

T y;

vertex<T>& operator=(vertex<T> A);

};

template<class T>

std::istream& operator>>(std::istream& is, vertex<T>& p) {

is >> p.x >> p.y;

return is;

}

template<class T>

std::ostream& operator<<(std::ostream& os, vertex<T> p) {

os << '(' << p.x << ' ' << p.y << ')';

return os;

}

template<class T>

vertex<T> operator+(const vertex<T>& A, const vertex<T>& B) {

vertex<T> res;

res.x = A.x + B.x;

res.y = A.y + B.y;

return res;

}

template<class T>

vertex<T>& vertex<T>::operator=(const vertex<T> A) {

this->x = A.x;

this->y = A.y;

return \*this;

}

template<class T>

vertex<T> operator+=(vertex<T> &A, const vertex<T> &B) {

A.x += B.x;

A.y += B.y;

return A;

}

template<class T>

vertex<T> operator/=(vertex<T>& A, const double B) {

A.x /= B;

A.y /= B;

return A;

}

template<class T>

double length(vertex<T>& A, vertex<T>& B) {

double res = sqrt( pow(B.x - A.x, 2) + pow(B.y - A.y, 2) );

return res;

}

template<class T>

struct is\_vertex : std::false\_type {};

template<class T>

struct is\_vertex<vertex<T>> : std::true\_type {};

#endif //VERTEX\_H

**figures.h:**

#ifndef FIGURES\_H

#define FIGURES\_H

#include "vertex.h"

#include <type\_traits>

#include <iostream>

template <class T>

class Octagon {

public:

vertex<T> points[8];

int size = 8;

explicit Octagon<T>(std::istream& is) {

for (auto & point : points) {

is >> point;

}

}

};

template <class T>

class Triangle {

public:

vertex<T> points[3];

int size = 3;

explicit Triangle<T>(std::istream& is) {

for (auto & point : points) {

is >> point;

}

}

};

template <class T>

class Square {

public:

vertex<T> points[4];

int size = 4;

explicit Square<T>(std::istream& is) {

for (auto & point : points) {

is >> point;

}

if (!is\_square(points)) {

throw std::logic\_error("square is not squarish enouth");

}

}

};

template<class T>

bool is\_square (vertex<T> points[4]) {

bool check = ((length(points[0], points[1]) == length(points[1], points[2])) && (length(points[2], points[3]) == length(points[3], points[0])) && (length(points[0], points[1]) == length(points[3], points[0])) ) && (((points[0].x - points[1].x) \* (points[2].x - points[1].x) + (points[0].y - points[1].y) \* (points[2].y - points[1].y)) == 0);

return check;

}

#endif // FIGURES\_H

**templates.h:**

#ifndef TEMPLATES\_H

#define TEMPLATES\_H

#include <tuple>

#include <type\_traits>

#include <cassert>

#include "vertex.h"

#include "figures.h"

template<class T, class = void>

struct has\_points : std::false\_type {};

template<class T>

struct has\_points<T, std::void\_t<decltype(std::declval<const T&>().points)>> : 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<size\_t Id, class T>

void print\_tuple(const T& figure, std::ostream& os) {

if constexpr (Id >= std::tuple\_size<T>::value) {

} else {

os << std::get<Id>(figure) << " ";

print\_tuple<Id + 1>(figure, os);

}

}

template <class T>

void print(const T& figure, std::ostream& os) {

if constexpr (has\_points<T>::value) {

for (auto point : figure.points) {

os << point << " ";

}

} else if constexpr (is\_figurelike\_tuple<T>::value) {

print\_tuple<0>(figure, os);

} else throw std::logic\_error("error");

}

template<size\_t Id, class T>

vertex<double> tuple\_center(const T& figure) {

if constexpr (Id >= std::tuple\_size<T>::value) {

return vertex<double> {0, 0};

} else {

vertex<double> res = std::get<Id>(figure);

return res + tuple\_center<Id+1>(figure);

}

}

template <class T>

vertex<double> center(const T& figure) {

vertex<double> res{0.0, 0.0};

int i = 0;

if constexpr (has\_points<T>::value) {

for (auto point : figure.points) {

res += point;

++i;

}

res /= i;

return res;

} else if constexpr (is\_figurelike\_tuple<T>::value) {

res = tuple\_center<0>(figure);

res /= std::tuple\_size\_v<T>;

return res;

} else throw std::logic\_error ("error");

}

template<size\_t Id, class T>

double tuple\_area(const T& figure) {

if constexpr (Id >= std::tuple\_size<T>::value - 1) {

return 0.0;

} else {

double res = (std::get<Id>(figure).x \* std::get<Id+1>(figure).y) - (std::get<Id+1>(figure).x \* std::get<Id>(figure).y);

return res + tuple\_area<Id+1>(figure);

}

}

template <class T>

double area(const T& figure) {

double res = 0.0;

if constexpr (has\_points<T>::value) {

for (int i = 0; i < figure.size-1; ++i) {

res += (figure.points[i].x \* figure.points[i+1].y) - (figure.points[i+1].x \* figure.points[i].y);

}

res += (figure.points[figure.size-1].x \* figure.points[0].y) - (figure.points[0].x \* figure.points[figure.size-1].y);

return std::abs(res) / 2;

} else if constexpr (is\_figurelike\_tuple<T>::value) {

res = tuple\_area<0>(figure);

res += (std::get<std::tuple\_size<T>::value - 1>(figure).x \* std::get<0>(figure).y) - (std::get<0>(figure).x \* std::get<std::tuple\_size<T>::value - 1>(figure).y);

return std::abs(res) / 2;

} else {

throw std::logic\_error ("error");

}

}

#endif //TEMPLATES\_H

**process.h:**

#ifndef PROCESS\_H\_

#define PROCESS\_H\_

#include <tuple>

#include <iostream>

#include "vertex.h"

#include "figures.h"

#include "templates.h"

template<class T>

void process(std::istream& is, std::ostream& os) {

if constexpr (has\_points<T>::value) {

T figure(is);

print(figure, os);

os << std::endl;

os << area(figure) << std::endl;

os << center(figure) << std::endl;

} else if constexpr (is\_figurelike\_tuple<T>::value) {

size\_t number;

os << "Please, write number of angles: ";

is >> number;

switch(number) {

case 3: {

vertex<double> points[3];

for (auto & i : points) {

is >> i;

}

auto[p1, p2, p3] = points;

auto figure = std::make\_tuple(p1, p2, p3);

print(figure, os);

os << std::endl;

os << area(figure) << std::endl;

os << center(figure) << std::endl;

break;

}

case 4: {

vertex<double> points[4];

for (auto & i : points) {

is >> i;

}

if (!is\_square(points)) {

throw std::logic\_error ("this is not square");

}

auto[p1, p2, p3, p4] = points;

auto figure = std::make\_tuple(p1, p2, p3, p4);

print(figure, os);

os << std::endl;

os << area(figure) << std::endl;

os << center(figure) << std::endl;

break;

}

case 8: {

vertex<double> points[8];

for (auto & i : points) {

is >> i;

}

auto[p1, p2, p3, p4, p5, p6, p7, p8] = points;

auto figure = std::make\_tuple(p1, p2, p3, p4, p5, p6, p7, p8);

print(figure, os);

os << std::endl;

os << area(figure) << std::endl;

os << center(figure) << std::endl;

break;

}

default:

throw std::logic\_error("number of angles must be 3, 4, 8");

}

} else {

throw std::logic\_error ("trying to process something wrong");

}

}

#endif //PROCESS\_H\_

**main.cpp:**

#include <iostream>

#include "templates.h"

#include "vertex.h"

#include "process.h"

int main() {

char key = 'a';

while (key != '0') {

std::cout << "1. triangle" << '\n' << "2. square" << '\n' << "3. octagon" << '\n' << "4. tuple" << '\n' << "0 for exit" << std::endl;

std::cin >> key;

switch (key) {

case '0':

break;

case '1': {

process<Triangle<double>>(std::cin, std::cout);

break;

}

case '2': {

process<Square<double>>(std::cin, std::cout);

break;

}

case '3': {

process<Octagon<double>>(std::cin, std::cout);

break;

}

case '4': {

process<std::tuple<vertex<double>>>(std::cin, std::cout);

break;

}

default: {

std::cout << "you write the wrong key!" << std::endl;

break;

}

}

}

return 0;

}

**2. Ссылка на репозиторий на Github**

[https://github.com/mmaxim2710/oop\_exercise\_0](https://github.com/mmaxim2710/oop_exercise_01)4

**3.Набор testcases**

1)

1

0 0 0 2 2 0

2

0 0 0 5 5 5 5 0

3

0 4 3 5 5 5 6 3 6 1 4 0 2 0 0 2

4

3

1 1 5 5 6 2

2)

2

1 1 2 2 3 3 4 4

**4. Результат выполнения тестов**

1)

(0 0) (0 2) (2 0)

2

(0.666667 0.666667)

(0 0) (0 5) (5 5) (5 0)

25

(2.5 2.5)

(0 4) (3 5) (5 5) (6 3) (6 1) (4 0) (2 0) (0 2)

24.5

(3.25 2.5)

(1 1) (5 5) (6 2)

8

(4 2.66667)

2) terminate called after throwing an instance of 'std::logic\_error'

what(): this is not square

**5. Объяснение результатов программы**

Данная программа состоит из логически нескольких частей.

vertex.h: заголовочный файл с описанием точки, перегрузками некоторых операторов, таких как ввод, вывод, +, =, +=, /=, функции вычисления длинны отрезка по 2-м точкам.

figures.h: заголовочный файл с описанием фигур по варианту задания: треугольник, квадрат, восьмиугольник.

templates.h: заголовочный файл с вспомогательными шаблонами, использующими type\_traits, определяющими, является ли данный объект тьюплом определённой фигуры, определенным по структуре, или классом фигуры; вводом выводом для класса фигуры и тьюпла, центром и площадью.

process.h: заголовочный файл, который принимает для обработки шаблон фигуры. Для тьюпла запрашивает количество углов (= количеству точек).

Вывод:

Проделав данную работу я приобрел навыки работы с шаблонами, библиотекой <type\_traits>, выявил плюсы шаблонов: используя шаблоны мы можем описать одну функцию, которая будет принимать разные типы/шаблоны/… и корректно обрабатывать их, вместо описания нескольких функций для разных типов. Закрепил навыки работы по обработке ошибок с помощью throw.