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-208Б-18, |
| Студент: | Рыженко И.А. |
| Преподаватель: | Поповкин А. В. |
| Оценка: |  |
| Дата: | 23.03.2021 |

Москва, 2021

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.

Вариант 22: 5-угольник, 6-угольник, 8-угольник.

1. **TestCase**

|  |
| --- |
| **Test 1**//добавление элементов в коллекцию  Input figure type:  1 - Pentagon  2 - Octagon  3 - Hexagon  4 - Tuple  'q' to close application  1  Input Vertex of the Pentagon:  Input coordinate 'x': 12  Input coordinate 'y': 213  Input coordinate 'x': 21  Input coordinate 'y': 231  Input coordinate 'x': 123  Input coordinate 'y': 123  Input coordinate 'x': 123  Input coordinate 'y': 123  Input coordinate 'x': 12  Input coordinate 'y': 12  [12 213] [21 231] [123 123] [123 123] [12 12]  Area of object: 12559  Center of object: [58.2 140.4]  Input figure type:  1 - Pentagon  2 - Octagon  3 - Hexagon  4 - Tuple  'q' to close application  4  Input tuple:  Enter the number of vertices  12  Input figure type:  1 - Pentagon  2 - Octagon  3 - Hexagon  4 - Tuple  'q' to close application  4  Input tuple:  Enter the number of vertices  112  Input figure type:  1 - Pentagon  2 - Octagon  3 - Hexagon  4 - Tuple  'q' to close application  3  Input Vertex of the hexagon:  Input coordinate 'x': 12  Input coordinate 'y': 124  Input coordinate 'x': 12  Input coordinate 'y': 214  Input coordinate 'x': 214  Input coordinate 'y': 124  Input coordinate 'x': 124  Input coordinate 'y': 12  Input coordinate 'x': 4214  Input coordinate 'y': 124  Input coordinate 'x': 124  Input coordinate 'y': 12  [12 124] [12 214] [214 124] [124 12] [4214 124] [124 12]  Area of object: 20402  Center of object: [783.333 101.667]  Input figure type:  1 - Pentagon  2 - Octagon  3 - Hexagon  4 - Tuple  'q' to close application  2  Input Vertex of the octagon:  Input coordinate 'x': 124  Input coordinate 'y': 12  Input coordinate 'x': 124  Input coordinate 'y': 214  Input coordinate 'x': 124  Input coordinate 'y': 12  Input coordinate 'x': 41  Input coordinate 'y': 421  Input coordinate 'x': 412  Input coordinate 'y': 4124  Input coordinate 'x': 124  Input coordinate 'y': 12  Input coordinate 'x': 124  Input coordinate 'y': 124  Input coordinate 'x': 124  Input coordinate 'y': 124  [124 12] [124 214] [124 12] [41 421] [412 4124] [124 12] [124 124] [124 124]  Area of object: 229544  Center of object: [149.625 630.375] |
| **Test 2//**  Input figure type:  1 - Pentagon  2 - Octagon  3 - Hexagon  4 - Tuple  'q' to close application  4  Input tuple:  Enter the number of vertices  5  Coordinate 'x': 12  Coordinate 'y': 214  Coordinate 'x': 124  Coordinate 'y': 21  Coordinate 'x': 4124  Coordinate 'y': 14  Coordinate 'x': 124  Coordinate 'y': 124  Coordinate 'x': 124  Coordinate 'y': 12  [12 214] [124 21] [4124 14] [124 124] [124 12]  Area of object: 565720  Center of object: [901.6 77] |

1. **Адрес репозитория на GitHub**

[https://github.com/THEproVANO/oop\_exercise\_04](https://github.com/vasilisavasileva/oop_exercise_04)

1. **Код программы на С++**

**vertex.h**

#pragma once

#include <iostream>

template<class T>

struct **Vertex** {

using vertex = std::pair<T,T>;

vertex coordinates;

};

template<class T>

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

Vertex<T> res;

res.coordinates.first = A.coordinates.first + B.coordinates.first;

res.coordinates.second = A.coordinates.second + B.coordinates.second;

return res;

}

template<class T>

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

A.coordinates.first /= B;

A.coordinates.second /= B;

return A;

}

template<class T>

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

{

std::cout << "Coordinate 'x': ";

is >> p.coordinates.first;

std::cout << "Coordinate 'y': ";

is >> p.coordinates.second;

return is;

}

template<class T>

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

os << '[' << p.coordinates.first << ' ' << p.coordinates.second << ']';

return os;

}

**templates.h**

#pragma once

#include <tuple>

#include <type\_traits>

#include <iostream>

#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\_area** : std::false\_type {};

template<class T>

struct **has\_method\_area**<T, std::void\_t<decltype(std::declval<const T&>().area())>> : std::true\_type {};

template<class T>

inline constexpr bool has\_method\_area\_v = has\_method\_area<T>::value;

template<class T>

std::enable\_if\_t<has\_method\_area\_v<T>, double> **area**(const T& object) {

return object.area();

}

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>, Vertex<double>> **center**(const T& object) {

return object.center();

}

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& os, const T& object) {

object.print(os);

}

template<size\_t Id, class T>

double **compute\_area**(const T& tuple) {

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

return 0;

else {

const auto x1 = std::get<Id - 0>(tuple).coordinates.first - std::get<0>(tuple).coordinates.first;

const auto y1 = std::get<Id - 0>(tuple).coordinates.second - std::get<0>(tuple).coordinates.second;

const auto x2 = std::get<Id - 1>(tuple).coordinates.first - std::get<0>(tuple).coordinates.first;

const auto y2 = std::get<Id - 1>(tuple).coordinates.second - std::get<0>(tuple).coordinates.second;

const double local\_area = std::abs(x1 \* y2 - y1 \* x2) \* 0.5;

return local\_area + compute\_area<Id + 1>(tuple);

}

}

template<class T>

std::enable\_if\_t<is\_figurelike\_tuple\_v<T>, double>

**area**(const T& object) {

if constexpr (std::tuple\_size\_v<T> < 3)

throw std::logic\_error("It`s not a figure");

else

return compute\_area<2>(object);

}

template<size\_t Id, class T>

Vertex<double> **tuple\_center**(const T& object) {

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

return Vertex<double> { std::make\_pair(0,0) };

else

{

Vertex<double> res = std::get<Id>(object);

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

}

}

template<class T>

Vertex<double> **compute\_center**(const T& tuple)

{

Vertex<double> res{ std::make\_pair(0,0) };

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

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

return res;

}

template<class T>

std::enable\_if\_t<is\_figurelike\_tuple\_v<T>, Vertex<double>>

**center**(const T& object) {

if constexpr (std::tuple\_size\_v<T> < 3)

throw std::logic\_error("It`s not a figure");

else

return compute\_center(object);

}

template<size\_t Id, class T>

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

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

std::cout << "\n";

else

{

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

print<Id + 1>(object, os);

}

}

template<class T>

std::enable\_if\_t<is\_figurelike\_tuple\_v<T>, void>

**print**(std::ostream& os, const T& object) {

if constexpr (std::tuple\_size\_v<T> < 3)

throw std::logic\_error("It`s not a figure");

else

print<0>(object, os);

}

**pentagon.h**

#pragma once

#include <iostream>

#include <cmath>

#include "vertex.h"

//Класс "Пятиугольника", методы аналогичны пятиугольнику

template<class T>

struct **Pentagon**

{

Vertex<T> vertices[5];

**Pentagon**(std::istream& is);

double **area**() const;

Vertex<T> **center**() const;

void **print**(std::ostream& os) const;

};

template<class T>

Pentagon<T>::**Pentagon**(std::istream& is)

{

for (int i = 0; i < 5; ++i)

{

std::cout << "Input coordinate 'x': ";

is >> vertices[i].coordinates.first;

std::cout << "Input coordinate 'y': ";

is >> vertices[i].coordinates.second;

}

}

template<class T>

double Pentagon<T>::**area**() const

{

double area = 0;

for (int i = 0; i < 5; i++) {

area += (vertices[i].coordinates.first) \* (vertices[(i + 1) % 5].coordinates.second) - (vertices[(i + 1) % 5].coordinates.first) \* (vertices[i].coordinates.second);

}

area \*= 0.5;

return abs(area);

}

template<class T>

Vertex<T> Pentagon<T>::**center**() const

{

Vertex<T> res;

res.coordinates.first = (vertices[0].coordinates.first + vertices[1].coordinates.first + vertices[2].coordinates.first + vertices[3].coordinates.first + vertices[4].coordinates.first) / 5;

res.coordinates.second = (vertices[0].coordinates.second + vertices[1].coordinates.second + vertices[2].coordinates.second + vertices[3].coordinates.second + vertices[4].coordinates.second) / 5;

return res;

}

template<class T>

void Pentagon<T>::**print**(std::ostream& os) const

{

for (int i = 0; i < 5; ++i) {

os << vertices[i];

if (i + 1 != 5) {

os << ' ';

}

}

}

**hexagon.h**

#pragma once

#include <iostream>

#include <cmath>

#include "vertex.h"

//Класс "Шестиугольника"

template<class T>

struct **Hexagon**

{

Vertex<T> vertices[6];

**Hexagon**(std::istream& is);

double **area**() const;

Vertex<T> **center**() const;

void **print**(std::ostream& os) const;

};

template<class T>

Hexagon<T>::**Hexagon**(std::istream& is)

{

for (int i = 0; i < 6; ++i)

{

std::cout << "Input coordinate 'x': ";

is >> vertices[i].coordinates.first;

std::cout << "Input coordinate 'y': ";

is >> vertices[i].coordinates.second;

}

}

template<class T>

double Hexagon<T>::**area**() const //вывод площади фигуры

{

double area = 0;

for (int i = 0; i < 6; i++) {

area += (vertices[i].coordinates.first) \* (vertices[(i + 1) % 6].coordinates.second) - (vertices[(i + 1) % 6].coordinates.first) \* (vertices[i].coordinates.second);

}

area \*= 0.5;

return abs(area);

}

template<class T>

Vertex<T> Hexagon<T>::**center**() const //вывод центра фигуры

{

Vertex<T> res;

res.coordinates.first = (vertices[0].coordinates.first + vertices[1].coordinates.first + vertices[2].coordinates.first + vertices[3].coordinates.first + vertices[4].coordinates.first + vertices[5].coordinates.first) / 6;

res.coordinates.second = (vertices[0].coordinates.second + vertices[1].coordinates.second + vertices[2].coordinates.second + vertices[3].coordinates.second + vertices[4].coordinates.second + vertices[5].coordinates.second) / 6;

return res;

}

template<class T>

void Hexagon<T>::**print**(std::ostream& os) const //вывод координат фигуры

{

for (int i = 0; i < 6; ++i)

{

os << vertices[i];

if (i + 1 != 6)

os << ' ';

}

}

**octagon.h**

#pragma once

#include <iostream>

#include <cmath>

#include "vertex.h"

//Класс "Восьмиугольника", методы аналогичны пятиугольнику

template<class T>

struct **Octagon** {

Vertex<T> vertices[8];

**Octagon**(std::istream& is);

double **area**() const;

Vertex<T> **center**() const;

void **print**(std::ostream& os) const;

};

template<class T>

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

for (int i = 0; i < 8; ++i)

{

std::cout << "Input coordinate 'x': ";

is >> vertices[i].coordinates.first;

std::cout << "Input coordinate 'y': ";

is >> vertices[i].coordinates.second;

}

}

template<class T>

double Octagon<T>::**area**() const {

double area = 0;

for (int i = 0; i < 8; i++) {

area += (vertices[i].coordinates.first) \* (vertices[(i + 1) % 8].coordinates.second) - (vertices[(i + 1) % 8].coordinates.first) \* (vertices[i].coordinates.second);

}

area \*= 0.5;

return abs(area);

}

template<class T>

Vertex<T> Octagon<T>::**center**() const {

Vertex<T> res;

res.coordinates.first = (vertices[0].coordinates.first + vertices[1].coordinates.first + vertices[2].coordinates.first + vertices[3].coordinates.first + vertices[4].coordinates.first + vertices[5].coordinates.first + vertices[6].coordinates.first + vertices[7].coordinates.first) / 8;

res.coordinates.second = (vertices[0].coordinates.second + vertices[1].coordinates.second + vertices[2].coordinates.second + vertices[3].coordinates.second + vertices[4].coordinates.second + vertices[5].coordinates.second + vertices[6].coordinates.first + vertices[7].coordinates.second) / 8;

return res;

}

template<class T>

void Octagon<T>::**print**(std::ostream& os) const {

for (int i = 0; i < 8; ++i) {

os << vertices[i];

if (i + 1 != 8) {

os << ' ';

}

}

}

**main.cpp**

#include <iostream>

#include <tuple>

#include "Vertex.h"

#include "pentagon.h"

#include "hexagon.h"

#include "octagon.h"

#include "templates.h"

template<class T>

void **processing**(std::istream& is, std::ostream& os)

{

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

{

int vert;

std::cout << "Enter the number of vertices" << std::endl;

std::cin >> vert;

if (vert == 5)

{

Vertex<double> A, B, C, D, E;

is >> A >> B >> C >> D >> E;

auto object = std::make\_tuple(*A*, *B*, *C*, *D*, *E*);

print(*os*, object);

os << "Area of object: " << area(object) << std::endl;

os << "Center of object: " <<center(object) << std::endl;

return;

}

else if (vert == 6)

{

Vertex<double> A, B, C, D, E, F;

is >> A >> B >> C >> D >> E >> F;

auto object = std::make\_tuple(*A*, *B*, *C*, *D*, *E*, *F*);

print(*os*, object);

os << "Area of object: " << area(object) << std::endl;

os << "Center of object: " <<center(object) << std::endl;

return;

}

else if (vert == 8)

{

Vertex<double> A, B, C, D, E, F, G, I;

is >> A >> B >> C >> D >> E >> F >> G >> I;

auto object = std::make\_tuple(*A*, *B*, *C*, *D*, *E*, *F*, *G*, *I*);

print(*os*, object);

os << "Area of object: " << area(object) << std::endl;

os << "Center of object: " <<center(object) << std::endl;

return;

}

else if (vert < 3)

{

std::cout << "The tuple is not a figure!\n";

}

}

else

{

T object(is);

print(os, object);

os << '\n' << "Area of object: "<< area(object) << std::endl;

os << "Center of object: " << center(object) << "\n\n";

return;

}

}

void **PrintMenu**() {

std::cout << "Input figure type:" << std::endl;

std::cout << "1 - Pentagon" << std::endl;

std::cout << "2 - Octagon" << std::endl;

std::cout << "3 - Hexagon" << std::endl;

std::cout << "4 - Tuple" << std::endl;

std::cout << "'q' to close application" << std::endl;

}

int **main**() {

char obj\_type;

while (true) {

PrintMenu();

std::cin >> obj\_type;

switch (obj\_type)

{

case'4':

std::cout << "Input tuple:\n";

processing<std::tuple<Vertex<double>>>(std*::cin*, std*::cout*);

break;

case'1':

std::cout << "Input Vertex of the Pentagon:\n";

processing<Pentagon<double>>(std*::cin*, std*::cout*);

break;

case'2':

std::cout << "Input Vertex of the octagon:\n";

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

break;

case'3':

std::cout << "Input Vertex of the hexagon:\n";

processing<Hexagon<double>>(std*::cin*, std*::cout*);

break;

case'q':

return 0;

default:

std::cout << "Incorrect input.\n\n" << std::endl;

break;

}

}

return 0;

}

1. **Объяснение результатов работы программы**

Начальное меню предлагает пользователю выбрать тип фигуры, координаты которой он собирается вводить. На выбор предоставляются pentagon(пятиугольник), hexagon(шестиугольник), octagon(восьмиугольник) и tuple(кортёж). После выбора фигуры основная программа направляет выполнение в running, где он разбивается на конкретные фигуры. Если фигура обыкновенная, то от нее просто вызываются шаблонные методы и выводятся результаты. Если вызывается tuple(кортёж), то программа предлагает ввести количество вершин и уже от этого начинает выполнение. Формируется tuple, и от него вызываются методы, описанные в templates.

1. **Вывод**

Частичная специализация шаблона позволяет писать код более сжато и обще, приспосабливая его под конкретные задачи. Это позволяет экономить время и значительно сокращать количество кода, в целом. Так, мы, в некотором роде, получаем доступ к инструментам логического программирования, и расширяем круг потенциально выполняемых задач.