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: | 21.12.2019 |

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.

1. **Адрес репозитория на GitHub**

https://github.com/poisoned-monkey/oop\_exercise\_04

1. **Код программы на С++**

*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<**int**>>();  
 **break**;  
 **case** 2:  
 process<Rhombus<**int**>>();  
 **break**;  
 }  
 **break**;  
 **case** 5:  
 process<Pentagon<**int**>>();  
 **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>  
  
#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*

#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;  
}

1. **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.

1. **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.