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-108Б-18, №6
Студент:	Васильева Василиса Евгеньевна
Преподаватель:	Журавлёв Андрей Андреевич
Оценка:	
Дата:	09.01.2020

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.

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

2. Адрес репозитория на GitHub

https://github.com/vasilisavasileva/oop_exercise_04

3. Код программы на C++

vertex.h

```
#pragma once
#include <iostream>

template<class T>
struct vertex {
	T x;
	T y;
};

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 double B) {
	A.x /= B;
	A.y /= B;
	return 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, const vertex<T>& p) {
	os << '[' << p.x << ' ' << p.y << ']';
	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).x - std::get<0>(tuple).x;
		const auto y1 = std::get<Id - 0>(tuple).y - std::get<0>(tuple).y;
		const auto x2 = std::get<Id - 1>(tuple).x - std::get<0>(tuple).x;
		const auto y2 = std::get<Id - 1>(tuple).y - std::get<0>(tuple).y;
		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> {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{ 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"


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) {
		is >> vertices[i];
	}
}

template<class T>
double pentagon<T>::area() const {
	double area = 0;
	for (int i = 0; i < 5; i++) {
		area += (vertices[i].x) * (vertices[(i + 1) % 5].y) - (vertices[(i + 1) %
5].x) * (vertices[i].y);
	}
	area *= 0.5;
	return abs(area);
}


template<class T>
vertex<T> pentagon<T>::center() const {
	vertex<T> res;
	res.x = (vertices[0].x + vertices[1].x + vertices[2].x + vertices[3].x +
vertices[4].x) / 5;
	res.y = (vertices[0].y + vertices[1].y + vertices[2].y + vertices[3].y +
vertices[4].y) / 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"


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) {
		is >> vertices[i];
	}
}

template<class T>
double hexagon<T>::area() const {
	double area = 0;
	for (int i = 0; i < 6; i++) {
		area += (vertices[i].x) * (vertices[(i + 1) % 6].y) - (vertices[(i + 1) %
6].x) * (vertices[i].y);
	}
	area *= 0.5;
	return abs(area);
}


template<class T>
vertex<T> hexagon<T>::center() const {
	vertex<T> res;
	res.x = (vertices[0].x + vertices[1].x + vertices[2].x + vertices[3].x +
vertices[4].x + vertices[5].x) / 6;
	res.y = (vertices[0].y + vertices[1].y + vertices[2].y + vertices[3].y +
vertices[4].y + vertices[5].y) / 6;
	return res;

}

template<class T>
void hexagon<T>::print(std::ostream& os) const {
	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"
```

```
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) {
		is >> vertices[i];
	}
}

template<class T>
double octagon<T>::area() const {
	double area = 0;
	for (int i = 0; i < 8; i++) {
		area += (vertices[i].x) * (vertices[(i + 1) % 8].y) - (vertices[(i + 1) %
8].x) * (vertices[i].y);
	}
	area *= 0.5;
	return abs(area);
}


template<class T>
vertex<T> octagon<T>::center() const {
	vertex<T> res;
	res.x = (vertices[0].x + vertices[1].x + vertices[2].x + vertices[3].x +
vertices[4].x + vertices[5].x + vertices[6].x + vertices[7].x) / 8;
	res.y = (vertices[0].y + vertices[1].y + vertices[2].y + vertices[3].y +
vertices[4].y + vertices[5].y + vertices[6].x + vertices[7].y) / 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(object) << std::endl;
			os << center(object) << std::endl;
			return;
		}
		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(object) << std::endl;
			os << center(object) << std::endl;
			return;
		}
		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(object) << std::endl;
			os << center(object) << std::endl;
			return;
		}
	}
	else {
		T object(is);
		print(os, object);
		os << '\n' << area(object) << std::endl;
		os << center(object) << std::endl;
		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 quit" << std::endl;
}

int main() {
	char obj_type;
	while (true) {
		PrintMenu();
		std::cin >> obj_type;
		switch (obj_type) {
		case'4':
			processing<std::tuple<vertex<double>>>(std::cin, std::cout);
			break;
		case'1':
			processing<pentagon<double>>(std::cin, std::cout);
			break;
		case'2':
			processing<octagon<double>>(std::cin, std::cout);
			break;
```

```
            case'3':
                processing<hexagon<double>>(std::cin, std::cout);
                break;

            case'q':
                return 0;

            default:
                std::cout << "Smth is wrong. Try another one." << std::endl;
                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 quit" << std::endl;

            }
        }
}
```

4. Объяснение результатов работы программы

Начальное меню предлагает пользователю выбрать тип фигуры, координаты которой он собирается вводить. На выбор предоставляются пентагон, гексагон, октагон и тьюпл, в соответствии с вариантом задания. 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 тьюпл, то программа предлагает ввести количество вершин и уже от этого начинает выполнение. Формируется тьюпл, и от него вызываются методы, описанные в темплейтах.

5. 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.