**Московский авиационный институт**

**(Национальный исследовательский университет)**

Институт: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Объектно-ориентированное программирование»

**Лабораторная работа № 4**

Тема: Основы метапрограммирования

Студент: Шавандрин Федор Михайлович

Группа: 80-208

Преподаватель: Чернышов Л.Н.

Дата:

Оценка:

Москва, 2020

1. **Постановка задачи**
2. Ознакомиться с теоретическим материалом по шаблонам и метапрограммированию в языке С++.
3. Разработать шаблоны классов “Rectangle”, “Rhombus”, “Trapezoid” для работы с прямоугольниками, ромбами и трапециями соответственно. Параметром шаблона должен являться скалярный тип данных, задающий тип данных для оси координат. Классы должны иметь только публичные поля. В классах не должно быть методов. Все фигуры являются правильными фигурами. Для хранения координат фигур необходимо использовать шаблон std::pair.
4. Необходимо реализовать две шаблонных функции:
5. Функция **print** для печати фигур на экран std::cout (печататься должны координаты вершин фигур). Функция должна принимать на вход std::tuple с фигурами, согласно варианту задания (минимум по одной каждого класса).
6. Функция **square** вычисления суммарной площади фигур. Функция должна принимать на вход std::tuple с фигурами, согласно варианту задания (минимум по одной каждого класса).
7. Создать программу, которая создает набор фигур согласно варианту задания (как минимум по одной фигуре каждого типа с координатами типа int и координатами типа double), сохраняет фигуры в std::tuple, печатает на экран содержимое std::tuple с помощью шаблонной функции print, вычисляет суммарную площадь фигур в std::tuple и выводит значение на экран.
8. Настроить CMake файл для сборки программы.
9. Подготовить наборы тестовых данных.
10. Загрузить файлы лабораторной работы в репозиторий GitHub.
11. **Описание программы**

Программа состоит из нескольких файлов.

Rectangle.h

В файле Rectangle.h описан шаблон класса Rectangle для работы с прямоугольниками. Параметром шаблона является тип данных для координат вершин прямоугольника. Прямоугольник задается левой нижней вершиной и стороной.

Rhombus.h

В файле Rhombus.h описан шаблон класса Rhombus для работы с ромбами. Параметром шаблона является тип данных для координат вершин ромба. Ромб задается нижней вершиной и двумя диагоналями.

Trapezoid.h

В файле Trapezoid.h описан шаблон класса Trapezoid для работы с трапециями. Параметром шаблона является тип данных для координат вершин трапеции. Трапеция задается левой нижней вершиной, основаниями трапециями и высотой. Основания параллельны оси абсцисс.

main.cpp

В начале функции *main* описываются фигуры, которые будут обрабатываться программой. Для хранения фигур используется std::tuple.

Для печати фигур на экран используется шаблонная рекурсивная функция *Tuple\_print*, принимающая в качестве аргумента tuple с фигурами. При помощи шаблона std::enable\_if проверяется, лежит ли данный индекс в кортеже или нет. Если лежит, то вызывается шаблонная функция *Print,* вычисляющая и выводящая на экран координаты фигуры, и рекурсивно вызывается *Tuple\_print* с инкрементированным индексом. Если индекс лежит за пределами кортежа, то печатается перевод строки.

Для каждой фигуры реализованы функции *Print.* Компилятор определяет нужную функцию по наличию специальных полей в классах фигур: если у фигуры есть поле *side*, то это прямоугольник, если есть *diag1* - ромб, если есть *height* - трапеция.

Аналогичным образом реализована рекурсивная шаблонная функция *TotalSquare*, вычисляющая общую площадь фигур, и функции square, вычисляющие площади определенных фигур. Для прямоугольника площадь вычисляется как произведение сторон, для ромба - полупроизведение диагоналей, для трапеции - полусумма оснований на высоту.

В функции *main* последовательно вызываются функции *Tuple\_print* и *TotalSquare*.

1. **Набор тестов**

Так как программа выполняет большинство вычислений на этапе компиляции, то итеративный ввод фигур не обязателен. Фигуры и их параметры описаны в главной функции. Всего в std::tuple помещены 6 фигур: 2 прямоугольника, 2 ромба, 2 трапециями.

Тест 1: Rectangle<int> r1; r1.lower\_left\_vertex = {1,5}; r1.side = 2;

Rectangle<double> r2 r2.lower\_left\_vertex = {0.3,2.4}; r2.side = 3;

Rhombus<int> rh1; rh1.center = {0,2}; rh1.diag1 = 2; rh1.diag2 = 3;

Rhombus<double> rh2; rh2.center = {0.5,0.6}; rh2.diag1 = 2.5; rh2.diag2 = 1.7;

Trapezoid<int> t1; t1.lower\_left\_vertex = {0,0}; t1.height = 2; t1.side1 = 3; t1.side2 = 5;

Trapezoid<double> t2; t2.lower\_left\_vertex = {1.2,2.5}; t2.height = 1.6; t2.side1 = 2.5; t2.side2 = 0.6;

-----------------------------------------------------------------------------------------

Тест 2:

Rectangle<int> r1; r1.lower\_left\_vertex = {0,0}; r1.side = 4;

Rectangle<double> r2; r2.lower\_left\_vertex = {1.5,3.5}; r2.side = 2.7;

Rhombus<int> rh1; rh1.center = {1,1}; rh1.diag1 = 1; rh1.diag2 = 5;

Rhombus<double> rh2; rh2.center = {0.7,1.2}; rh2.diag1 = 1.1; rh2.diag2 = 0.8;

Trapezoid<int> t1; t1.lower\_left\_vertex = {1,2}; t1.height = 7; t1.side1 = 5; t1.side2 = 6;

Trapezoid<double> t2; t2.lower\_left\_vertex = {0.9,1.1}; t2.height = 3.4; t2.side1 = 1.3; t2.side2 = 0.8;

1. **Результаты выполнения тестов**

Для теста 1: Rectangle: {(1, 5), (1, 7), (3, 7), (3, 5)} Rhombus: {(0, 1), (-1.5, 2), (0, 3), (1.5, 2)} Trapezoid: {(0, 0), (-1, 2), (4, 2), (3, 0)} Rectangle: {(0.3, 2.4), (0.3, 5.4), (3.3, 5.4), (3.3, 2.4)} Rhombus: {(0.5, -0.65), (-0.35, 0.6), (0.5, 1.9), (1.4, 0.6)} Trapezoid: {(1.2, 2.5), (2.1, 4.1), (2.8, 4.1), (3.7, 2.5)}

Total square is 28.61

-----------------------------------------------------------------------------------------

Для теста 2

Rectangle: {(0, 0), (0, 4), (4, 4), (4, 0)} Rhombus: {(1, 0.5), (-1.5, 1), (1, 1.5), (3.5, 1)} Trapezoid: {(1, 2), (0.5, 9), (6.5, 9), (6, 2)} Rectangle: {(1.5, 3.5), (1.5, 6.2), (4.2, 6.2), (4.2, 3.5)} Rhombus: {(0.7, 0.65), (0.3, 1.2), (0.7, 1.8), (1.1, 1.2)} Trapezoid: {(0.9, 1.1), (1.1, 4.5), (2, 4.5), (2.2, 1.1)}

Total square is 68.30

1. **Листинг программы**

Rectangle.h

#ifndef OOP\_EXERCISE\_004\_RECTANGLE\_H

#define OOP\_EXERCISE\_004\_RECTANGLE\_H

template<class T>

class Rectangle{

public:

std::pair<T,T> lower\_left\_vertex;

double side;

};

#endif //OOP\_EXERCISE\_004\_RECTANGLE\_H

Rhombus.h

#ifndef OOP\_EXERCISE\_004\_RHOMBUS\_H

#define OOP\_EXERCISE\_004\_RHOMBUS\_H

template<class T>

class Rhombus{

public:

std::pair<T,T> center;

double diag1;

double diag2;

};

#endif //OOP\_EXERCISE\_004\_RHOMBUS\_H

Trapezoid.h

#ifndef OOP\_EXERCISE\_004\_TRAPEZOID\_H

#define OOP\_EXERCISE\_004\_TRAPEZOID\_H

template<class T>

class Trapezoid{

public:

std::pair<T,T> lower\_left\_vertex;

double ‘side1;

double side2;

double height;

};

#endif //OOP\_EXERCISE\_004\_TRAPEZOID\_H

main.cpp

/\*

Created by Шавандрин Фёдор

Ссылка на репозиторий https://github.com/Adi6e/oop\_exercise\_04

Разработать шаблоны классов “Rectangle”, “Rhombus”, “Trapezoid” для работы с прямоугольниками, ромбами и трапециями соответственно.

Параметром шаблона должен являться скалярный тип данных, задающий тип данных для оси координат.

Классы должны иметь только публичные поля. В классах не должно быть методов.

Все фигуры являются правильными фигурами.

Для хранения координат фигур необходимо использовать шаблон std::pair.

\*/

#include <iostream>

#include <tuple>

#include <cmath>

#include "Rectangle.h"

#include "Rhombus.h"

#include "Trapezoid.h"

//prints a tuple

template<class T, int index>

typename std::enable\_if<index >= std::tuple\_size<T>::value,void>::type Tuple\_print(T &t){

std::cout << std::endl;

}

template<class T, int index>

typename std::enable\_if<index < std::tuple\_size<T>::value,void>::type Tuple\_print(T &t){

auto f = std::get<index>(t);

Print(f);

Tuple\_print<T, index + 1>(t);

}

//prints a rectangle

template<class T>

typename std::enable\_if<sizeof(T::side > 0),void>::type Print(T& rectangle){

std::cout.precision(2);

std::cout << "Rectangle: {(" << rectangle.lower\_left\_vertex.first << ", " << rectangle.lower\_left\_vertex.second << "), (";

std::cout << rectangle.lower\_left\_vertex.first << ", " << rectangle.lower\_left\_vertex.second + rectangle.side << "), (";

std::cout << rectangle.lower\_left\_vertex.first + rectangle.side << ", " << rectangle.lower\_left\_vertex.second + rectangle.side << "), (";

std::cout << rectangle.lower\_left\_vertex.first + rectangle.side << ", " << rectangle.lower\_left\_vertex.second << ")}" << std::endl;

}

//prints a rhombus

template<class T>

typename std::enable\_if<sizeof(T::diag1 > 0),void>::type Print(T& rh){

std::cout.precision(2);

std::cout << "Rhombus: {(" << rh.center.first << ", " << rh.center.second - rh.diag1 / 2 << "), (";

std::cout << rh.center.first - rh.diag2 / 2 << ", " << rh.center.second << "), (";

std::cout << rh.center.first << ", " << rh.center.second + rh.diag1 / 2 << "), (";

std::cout << rh.center.first + rh.diag2 / 2 << ", " << rh.center.second << ")}" << std::endl;

}

//prints a trapezoid

template<class T>

typename std::enable\_if<sizeof(T::height > 0),void>::type Print(T& t){

std::cout.precision(2);

if(t.side1 > t.side2){

std::cout << "Trapezoid: {(" << t.lower\_left\_vertex.first << ", " << t.lower\_left\_vertex.second << "), (";

std::cout << t.lower\_left\_vertex.first + (double)(t.side1 - t.side2)/2 << ", " << t.lower\_left\_vertex.second + t.height << "), (";

std::cout << t.lower\_left\_vertex.first + (double)(t.side1 - t.side2)/2 + t.side2 << ", " << t.lower\_left\_vertex.second + t.height << "), (";

std::cout << t.lower\_left\_vertex.first + t.side1 << ", " << t.lower\_left\_vertex.second << ")}" << std::endl;

}

else{

std::cout << "Trapezoid: {(" << t.lower\_left\_vertex.first << ", " << t.lower\_left\_vertex.second << "), (";

std::cout << t.lower\_left\_vertex.first - (double)(t.side2 - t.side1)/2 << ", " << t.lower\_left\_vertex.second + t.height << "), (";

std::cout << t.lower\_left\_vertex.first - (double)(t.side2 - t.side1)/2 + t.side2 << ", " << t.lower\_left\_vertex.second + t.height << "), (";

std::cout << t.lower\_left\_vertex.first + t.side1 << ", " << t.lower\_left\_vertex.second << ")}" << std::endl;

}

}

//total square of figures in tuple

template<class T, int index>

typename std::enable\_if<index >= std::tuple\_size<T>::value,double>::type TotalSquare(T& t){

return 0;

}

template<class T, int index>

typename std::enable\_if<index < std::tuple\_size<T>::value,double>::type TotalSquare(T& t){

auto f = std::get<index>(t);

double cur\_square = Square(f);

return cur\_square + TotalSquare<T, index + 1>(t);

}

//square of rectangle

template<class T>

typename std::enable\_if<sizeof(T::side > 0),double>::type Square(T& r){

return r.side \* r.side;

}

//square of rhombus

template<class T>

typename std::enable\_if<sizeof(T::diag1 >0 ),double>::type Square(T& rh){

return rh.diag1 \* rh.diag2 / 2;

}

//square of trapezoid

template<class T>

typename std::enable\_if<sizeof(T::side1 > 0),double>::type Square(T& t){

return (t.side1 + t.side2) / 2 \* t.height;

}

int main(){

//creating figures

Rectangle<int> r1;

r1.lower\_left\_vertex = {1,5};

r1.side = 2;

Rectangle<double> r2;

r2.lower\_left\_vertex = {0.3,2.4};

r2.side = 3;

Rhombus<int> rh1;

rh1.center = {0,2};

rh1.diag1 = 2;

rh1.diag2 = 3;

Rhombus<double> rh2;

rh2.center = {0.5,0.6};

rh2.diag1 = 2.5;

rh2.diag2 = 1.7;

Trapezoid<int> t1;

t1.lower\_left\_vertex = {0,0};

t1.height = 2;

t1.side1 = 3;

t1.side2 = 5;

Trapezoid<double> t2;

t2.lower\_left\_vertex = {1.2,2.5};

t2.height = 1.6;

t2.side1 = 2.5;

t2.side2 = 0.6;

//creating tuple

std::tuple<decltype(r1),decltype(rh1),decltype(t1),decltype(r2),decltype(rh2),decltype(t2)>

tuple {r1, rh1, t1, r2, rh2, t2};

Tuple\_print<decltype(tuple),0>(tuple);

std::cout << std::fixed << "Total square is " << TotalSquare<decltype(tuple),0>(tuple);

}

1. **Выводы**

Данная лабораторная работа направлена на изучение основ метапрограммирования. Были приобретены навыки работы с кортежами (std::tuple), с шаблонными функциями и классами.

Метапрограммирование позволяет компилятору самому генерировать необходимый код, а также производит некоторые вычисления на этапе компиляции.

SFINAE, или Substitution Error Is Not An Error, необходим для того, чтобы компилятор, найдя ошибку в конкретной специализации шаблона класса, не выводил ошибку, а анализировал возможные варианты.

**Список литературы**

1. Руководство по языку С++ [Электронный ресурс] URL: <https://qosys.info/tuple-cpp> (дата обращения 02.11.2020).
2. Шаблоны функций в С++ [Электронный ресурс] URL: <https://ravesli.com/urok-173-shablony-funktsij/> (дата обращения 01.11.2020).
3. Шаблоны классов в С++ [Электронный ресурс] URL: <http://cppstudio.com/post/5188/> (дата обращения 01.11.2020).