Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет: «Информационных технологий и прикладной математики»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Объектно-ориентированное программирование»

**Лабораторная работа № 3**

**Тема: Основы метапрограммирования**

|  |  |
| --- | --- |
| Cтудент: | Королев И.М. |
| Группа: | 8О-208Б |
| Преподаватель: | Чернышов Л.Н. |
| Дата: |  |
| Оценка: |  |

Москва 2020

Оглавление

[1. Постановка задачи 2](#_Toc55733520)

[2. Описание программы 3](#_Toc55733521)

[3. Тесты и результаты их выполнения 6](#_Toc55733526)

[5. Листинг программы 9](#_Toc55733527)

[6. Выводы 14](#_Toc55733528)

[Список используемых источников 15](#_Toc55733529)

# 1. Постановка задачи

Разработать шаблоны классов согласно варианту задания. Классы должны иметь только публичные поля. В классах не должно быть методов, они должны иметь только поля. Фигуры являются фигурами вращения (равносторонними), за исключением трапеции и прямоугольника. Для хранения координат фигур необходимо использовать шаблон std::pair.

Необходимо реализовать две шаблонные функции:

1. Функция **print** 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 для каждого класса);
2. Функция **square** вычисления суммарной площади фигур. Функция должна принимать на вход std::tuple с фигурами, согласно варианту задания (минимум по одной для каждого класса).
3. Вычисление площади фигуры.

Создать программу, которая позволяет:

* Создаёт 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.

При реализации шаблонных функций допускается использование вспомогательных шаблонов std::enable\_if, std::tuple\_size, std::is\_same.

Таблица 1 – вариант задания

|  |  |  |  |
| --- | --- | --- | --- |
| **Вариант** | **Фигура № 1** | **Фигура № 2** | **Фигура № 3** |
| 10 | Квадрат | Прямоугольник | Трапеция |

# 2. Описание программы

Так как std::tuple содержит неизменяемое количество элементов, и эти элементы должны быть заданы изначально, то для данной лабораторной работы меню не понадобилось.

В программе были реализованы структуры Quadrate, Rectangle, Trapeze, каждая из которых хранит координаты геометрического центра фигуры с помощью std::pair и дополнительные параметры фигуры. Эти фигуры сохраняются в кортеж std::tuple. Он может хранить фиксированное количество элементов разного типа.

Были реализованы шаблонные функции для вывода координат фигур и вывода общей площади фигур из std::tuple. Координаты фигур вычисляются с помощью сложения или вычитания из координат центра фигуры половины стороны или половины высоты (в зависимости от фигуры).

# Переменные структуры Quadrate

*std::pair<T,T> Center*  – координаты геометрического центра квадрата.

*T Side* – сторона фигуры.

# Переменные структуры Rectangle

std::pair<T,T> Center – координаты геометрического центра прямоугольника.

*T Length* – длина прямоугольника.

*T Width* – ширина прямоугольника.

# Переменные структуры Trapeze

*std::pair<T,T> Center*  – координаты геометрического центра квадрата.

*T UpperSide* – верхнее основание трапеции.

T DownSide – нижнее основание трапеции.

T Height – высота трапеции.

# Описание шаблонных функций

Чтобы не повторять функции с разными структурами в качестве параметров, имена этих структур бы ли заменены на *Figure*.

*void check\_figure(Figure<T> figure)* – функция проверки параметров фигуры на правильность. Стороны фигуры не должны быть меньше либо равны нулю. *T* – тип членов структуры.

*void print(Figure<T>& figure)* – функция вывода координат фигуры.

*double square(const Figure<T> &figure)* – функция получения площади фигуры.

*void print\_tuple(T& tuple)* – функция вывода координат каждой фигуры, записанной в std::tuple. Функция задает тип фигуры auto, чтобы компилятор мог сам решить к какого типа является определённая фигура.

*double tuple\_square(T& tuple)* – функция подсчёта и вывода общей площади фигур, записанных в std::tuple. Функция выполняется рекурсивно. К площади первого элемента из std::tuple прибавляется остальная площадь фигур из него, которая вызывается с помощью tuple\_square(T& tuple). И так повторяется до тех пор, пока индекс элемента не становится равным размеру std::tuple, после чего возвращается нуль, который прибавляется к площади последнего элемента. Их сумма прибавляется к площади предпоследнего элемента и так далее до первого элемента. После чего, будет выведен окончательный результат об общей площади фигур в std::tuple. Также в функции есть спецификатор constexpr. Он нужен для того, чтобы, если есть такая возможность, то компилятор мог бы посчитать результат вычисления на этапе компиляции.

# 3. Тесты и результаты их выполнения

Были проведены тесты. Тесты представляют собой фрагменты кода, который был изменён для показа работы программы при разных параметрах фигур.

Таблица 2 – test\_01.txt и результаты его выполнения

|  |  |
| --- | --- |
| **Введённые данные** | **Результат** |
| Quadrate<int> quadre1; quadre1.Center = {0, 0}; quadre1.Side = 4;  Quadrate<double> quadre2; quadre2.Center = {0.35, 10.5}; quadre2.Side = 8;  Rectangle<int> rectangle1; rectangle1.Center = {25, 100}; rectangle1.Length = 25; rectangle1.Width = 100;  Rectangle<double> rectangle2; rectangle2.Center = {45.85, 36.9}; rectangle2.Length = 56.7; rectangle2.Width = 1003.3;  Trapeze<int> trapeze1; trapeze1.Center = {1000, -86}; trapeze1.DownSide = 100; trapeze1.UpperSide = 5; trapeze1.Height = 56;   Trapeze<double> trapeze2; trapeze2.Center = {-9.85, -34.8}; trapeze2.DownSide = 89.5; trapeze2.UpperSide = 100.2; trapeze2.Height = 2.7; | Quadrate: {(2, 2), (-2, 2), (-2, -2), (2, -2)}  Quadrate: {(4.35, 14.5), (-3.65, 14.5),  (-3.65, 6.5), (4.35, 6.5)}  Rectangle: {(37.5, 150), (12.5, 150),  (12.5, 50), (37.5, 50)}  Rectangle: {(74.2, 538.55), (17.5, 538.55), (17.5, -464.75), (74.2, -464.75)}  Trapeze: {(1050, -114), (1002.5, -58),  (997.5, -58), (950, -114)}  Trapeze: {(34.9, -36.15), (40.25, -33.45), (-59.95, -33.45), (-54.6, -36.15)}  Total square of tuple: 62663.2050 |

Таблица 3 – test\_02.txt и результаты его выполнения

|  |  |
| --- | --- |
| **Введённые данные** | **Результат** |
| Quadrate<int> quadre1; quadre1.Center = {1, 54}; quadre1.Side = 87;  Quadrate<double> quadre2; quadre2.Center = {-493.43, 245.78}; quadre2.Side = 368.456;  Rectangle<int> rectangle1; rectangle1.Center = {53, 53}; rectangle1.Length = 100; rectangle1.Width = 11;  Rectangle<double> rectangle2; rectangle2.Center = {98.54, 33.556}; rectangle2.Length = 120.43; rectangle2.Width = 321.343;  Trapeze<int> trapeze1; trapeze1.Center = {90,323}; trapeze1.DownSide = 200; trapeze1.UpperSide = 45; trapeze1.Height = 3283;   Trapeze<double> trapeze2; trapeze2.Center = {22.321, 12.23}; trapeze2.DownSide = 12; trapeze2.UpperSide = 55.45; trapeze2.Height = 2.6; | Quadrate: {(44.5, 97.5), (-42.5, 97.5), (-42.5, 10.5), (44.5, 10.5)}  Quadrate: {(-309.202, 430.008),  (-677.658, 430.008), (-677.658, 61.552),  (-309.202, 61.552)}  Rectangle: {(103, 58.5), (3, 58.5), (3, 47.5), (103, 47.5)}  Rectangle: {(158.755, 194.228),  (38.325, 194.228), (38.325, -127.116),  (158.755, -127.116)}  Trapeze: {(112.5, -1318.5), (190, 1964.5),  (-10, 1964.5), (67.5, -1318.5)}  Trapeze: {(50.046, 10.93), (28.321, 13.53), (16.321, 13.53), (-5.404, 10.93)}  Total square of tuple: 585383.3464 |

Таблица 4 – test\_03.txt и результаты его выполнения

|  |  |
| --- | --- |
| **Введённые данные** | **Результат** |
| Quadrate<int> quadre1; quadre1.Center = {100003, 3231}; quadre1.Side = 50;  Quadrate<double> quadre2; quadre2.Center = {3231.34, -9043.5}; quadre2.Side = 78;  Rectangle<int> rectangle1; rectangle1.Center = {-32, 90}; rectangle1.Length = 34; rectangle1.Width = 5;  Rectangle<double> rectangle2; rectangle2.Center = {-98.32, -1023.2}; rectangle2.Length = 32; rectangle2.Width = 10.2;  Trapeze<int> trapeze1; trapeze1.Center = {1, 53}; trapeze1.DownSide = 5; trapeze1.UpperSide = 6; trapeze1.Height = 2;   Trapeze<double> trapeze2; trapeze2.Center = {-23.1, -32.432}; trapeze2.DownSide = 10.321; trapeze2.UpperSide = 11.232; trapeze2.Height = 1.1; | Quadrate: {(3270.34, -9004.5),  (3192.34, -9004.5), (3192.34, -9082.5),  (3270.34, -9082.5)}  Rectangle: {(-15, 92.5), (-49, 92.5),  (-49, 87.5), (-15, 87.5)}  Rectangle: {(-82.32, -1018.1),  (-114.32, -1018.1), (-114.32, -1028.3),  (-82.32, -1028.3)}  Trapeze: {(4, 52), (3.5, 54), (-1.5, 54),  (-2, 52)}  Trapeze: {(-17.484, -32.982),  (-17.9395, -31.882), (-28.2605, -31.882),  (-28.716, -32.982)}  Total square of tuple: 9103.2541 |

# 5. Листинг программы

**CMakeLists.txt**

cmake\_minimum\_required (VERSION 3.2)

project (OOP)

add\_executable (oop\_exercise\_04 main.cpp)

set\_target\_properties(oop\_exercise\_04 PROPERTIES CXX\_STANDART 17

CXX\_STANDART\_REQUIRED ON)

**quadrate.hpp**

#pragma once

#include <iostream>

template <class T>

struct Quadrate {

    Quadrate() {}

    Quadrate(std::pair<T,T> center, T side): Center(center), Side(side) {}

    std::pair<T,T> Center;

    T Side;

};

template<class T>

void check\_figure(Quadrate<T> figure) {

    if(figure.Side <= 0) {

        std::cout << "Quadrate parameters were entered incorrectly!\n";

        std::cout << "The program has been completed ahead of schedule!\n";

        exit(1);

    }

}

// Quadrate print

template <class T>

void print(Quadrate<T>& figure) {

    check\_figure(figure);

    std::cout << "Quadrate: ";

    std::cout << "{(" << figure.Center.first + figure.Side / 2.0 << ", " <<

figure.Center.second + figure.Side / 2.0 << "), ";

    std::cout << "(" << figure.Center.first - figure.Side / 2.0 << ", " << figure.Center.second + figure.Side / 2.0 << "), ";

    std::cout << "(" << figure.Center.first - figure.Side / 2.0 << ", " << figure.Center.second - figure.Side / 2.0 << "), ";

    std::cout << "(" << figure.Center.first + figure.Side / 2.0 << ", " <<

figure.Center.second - figure.Side / 2.0 << ")}\n";

    std::cout << std::endl;

}

// Square of quadrate

template <class T>

double square(const Quadrate<T> &figure) {

    check\_figure(figure);

    return (figure.Side \* figure.Side);

}

**rectangle.hpp**

#pragma once

#include <iostream>

template<class T>

struct Rectangle {

    Rectangle() {}

    Rectangle(std::pair<T,T> center, T length, T width): Center(center),

Length(length), Width(width) {}

    std::pair<T,T> Center;

    T Length;

    T Width;

};

template<class T>

void check\_figure(Rectangle<T> figure) {

    if(figure.Length <= 0 || figure.Width <= 0) {

        std::cout << "Rectangle parameters were entered incorrectly!\n";

        std::cout << "The program has been completed ahead of schedule!\n";

        exit(1);

    }

}

// Rectangle print

template <class T>

void print(Rectangle<T>& figure) {

    check\_figure(figure);

    std::cout << "Rectangle: ";

    std::cout << "{(" << figure.Center.first + figure.Length / 2.0 << ", " <<

figure.Center.second + figure.Width / 2.0 <<"), ";

    std::cout << "(" << figure.Center.first - figure.Length / 2.0 << ", " << figure.Center.second + figure.Width / 2.0 <<

"), ";

    std::cout << "(" << figure.Center.first - figure.Length / 2.0 << ", " << figure.Center.second - figure.Width / 2.0 <<"), ";

    std::cout << "(" << figure.Center.first + figure.Length / 2.0 << ", " <<

figure.Center.second - figure.Width / 2.0 <<")}\n";

    std::cout << std::endl;

}

// Square of rectangle

template <class T>

double square(const Rectangle<T> &figure) {

    check\_figure(figure);

    return (figure.Length \* figure.Width);

}

**trapeze.hpp**

#pragma once

#include <iostream>

template<class T>

struct Trapeze {

    Trapeze() {}

    Trapeze(std::pair<T,T> center, T upper\_side, T down\_side, T height):

Center(center), UpperSide(upper\_side), DownSide(down\_side), Height(height) {}

    std::pair<T,T> Center;

    T UpperSide;

    T DownSide;

    T Height;

};

template<class T>

void check\_figure(Trapeze<T> figure) {

    if(figure.UpperSide <= 0 || figure.DownSide <= 0 || figure.Height <=0) {

        std::cout << "Trapeze parameters were entered incorrectly!\n";

        std::cout << "The program has been completed ahead of schedule!\n";

        exit(1);

    }

}

// Trapeze print

template <class T>

void print(Trapeze<T>& figure) {

    check\_figure(figure);

    std::cout << "Trapeze: ";

    std::cout << "{(" << figure.Center.first + figure.DownSide / 2.0 << ", " << figure.Center.second - figure.Height / 2.0 <<"), ";

    std::cout << "(" << figure.Center.first + figure.UpperSide / 2.0 << ", " << figure.Center.second + figure.Height / 2.0 <<"), ";

    std::cout << "(" << figure.Center.first - figure.UpperSide / 2.0 << ", " << figure.Center.second + figure.Height / 2.0 <<"), ";

    std::cout << "(" << figure.Center.first - figure.DownSide / 2.0 << ", " << figure.Center.second - figure.Height / 2.0 <<")}\n";

    std::cout << std::endl;

}

// Square of trapeze

template <class T>

double square(const Trapeze<T> &figure) {

    check\_figure(figure);

    return (figure.UpperSide + figure.DownSide) / 2.0 \* figure.Height;

}

**tuple.hpp**

#pragma once

#include <iostream>

#include <tuple>

#include <utility>

#include <type\_traits>

// Tuple print

template<class T, size\_t index>

typename std::enable\_if<index == std::tuple\_size<T>::value, void>::type

print\_tuple(T&) {}

template <class T, size\_t index>

typename std::enable\_if<(index < std::tuple\_size<T>::value), void>::type

print\_tuple(T& tuple) {

    auto figure = std::get<index>(tuple);

    print(figure);

    print\_tuple<T,index+1>(tuple);

}

// The whole square of tuple

template <class T, size\_t index = 0>

long double tuple\_square(T& tuple) {

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

        return (long double)square(std::get<index>(tuple)) +

tuple\_square<T, index + 1>(tuple);

    }

    else {

        return 0;

    }

}

**main.cpp**

#include <iostream>

#include <iomanip>

#include <tuple>

#include "quadrate.hpp"

#include "rectangle.hpp"

#include "trapeze.hpp"

#include "tuple.hpp"

int main() {

    Quadrate<int> quadre1({100003, 3231}, 50);

    Quadrate<double> quadre2({3231.34, -9043.5}, 78);

    Rectangle<int> rectangle1({-32, 90}, 34, 5);

    Rectangle<double> rectangle2({-98.32, -1023.2}, 32, 10.2);

    Trapeze<int> trapeze1({1, 53}, 5, 6, 2);

    Trapeze<double> trapeze2({-23.1, -32.432}, 10.321, 11.232, 1.1);

    // Filling std::tuple by figures

    std::tuple<decltype(quadre1), decltype(quadre2), decltype(rectangle1),

decltype(rectangle2), decltype(trapeze1), decltype(trapeze2)>

    tuple {quadre1, quadre2, rectangle1, rectangle2, trapeze1, trapeze2};

    // Print coordinates of figures in tuple

    print\_tuple<decltype(tuple),0>(tuple);

    std::cout << "Total square of tuple: " << std::fixed << std::setprecision(4) << tuple\_square<decltype(tuple),0>(tuple) << std::endl;

}

Ссылка на github репозиторий: <https://github.com/HarryLiker/oop_exercise_04>

# 6. Выводы

Была написана программа, в которой реализованы шаблоны классов квадрата, прямоугольника и трапеции. Она создаёт фигуры указанного типа и записывает их в std::tuple. Были реализованы функции вывода координат фигур, находящихся в кортеже, и функция нахождения общей площади этих фигур.

Выполняя лабораторную работу, были получены навыки работы с шаблонными функциями, шаблоном std::pair, кортежем, который хранит в себе фиксированное количество элементов разного типа. Были получены знания об использовании рекурсии в шаблонных функциях. Также были получены знания о том, как компилятор может понять тип переменной на этапе компиляции.

# Список используемых источников

1. Р. Лафоре «Объектно-ориентированное программирование в C++» – Москва: «Питер», 2004.

2. Информация о шаблонных классах и функциях – [Электронный ресурс]. – URL: <https://code-live.ru/post/cpp-template-functions/> (дата обращения: 04.11.2020).

3. Информация о шаблоне std::pair – [Электронный ресурс]. – URL: <https://ru.cppreference.com/w/cpp/utility/pair> (дата обращения: 04.11.2020).

4. Информация о шаблонном классе std::tuple – [Электронный ресурс]. – URL: <https://habr.com/ru/post/318236/> (дата обращения: 05.11.2020).

5. Информация о шаблонном классе std::tuple – [Электронный ресурс]. – URL: <https://ru.cppreference.com/w/cpp/utility/tuple> (дата обращения: 05.11.2020).

6. Информация о спецификаторе auto – [Электронный ресурс]. – URL: <https://ru.cppreference.com/w/cpp/language/auto> (дата обращения: 06.11.2020).

7. Информация о спецификаторе auto – [Электронный ресурс]. – URL: <https://ravesli.com/urok-62-klyuchevoe-slovo-auto-vyvod-tipov/> (дата обращения: 06.11.2020).

8. Информация о спецификаторе auto – [Электронный ресурс]. – URL: <https://docs.microsoft.com/ru-ru/cpp/cpp/auto-cpp?view=msvc-160>

(дата обращения: 07.11.2020).

9. Информация о спецификаторе constexpr – [Электронный ресурс]. – URL: <https://habr.com/ru/post/228181/> (дата обращения: 07.11.2020).