Московский Авиационный Институт
(Национальный Исследовательский Университет)

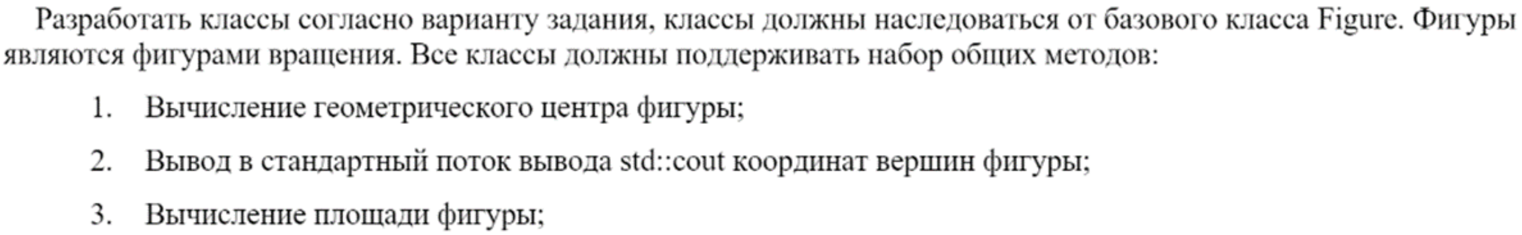
Кафедра 806 «Вычислительная информатика и программирование»
Факультет: «Информационные технологии и прикладная математика»

Лабораторная работа
Дисциплина: «Объектно-ориентированное программирование»
III семестр
Задание 4: «Основы метапрограммирования»

Группа:	М8О-206Б-18, №27
Студент:	Шорохов Алексей Павлович
Преподаватель:	Журавлёв Андрей Андреевич
Оценка:	
Дата:	

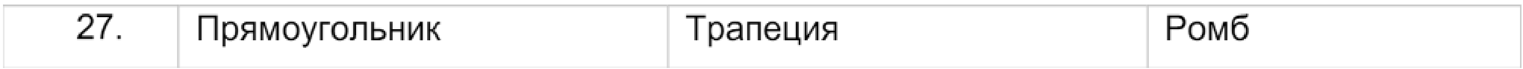
Москва, 2019

Задание



Разработать классы согласно варианту задания, классы должны наследоваться от базового класса Figure. Фигуры являются фигурами вращения. Все классы должны поддерживать набор общих методов:

1. Вычисление геометрического центра фигуры;
2. Вывод в стандартный поток вывода std::cout координат вершин фигуры;
3. Вычисление площади фигуры;



27.	Прямоугольник	Трапеция	Ромб

Адрес репозитория на GitHub

Код программы на C++

```
cmake_minimum_required(VERSION 3.2)

project(meta)

add_executable(lab4
        Source.cpp
        )

set_property(TARGET meta PROPERTY CXX_STANDART 11)
```

<u>vertex.h</u>

```
#ifndef D_VERTEX_H
#define D_VERTEX_H 1

#include <iostream>

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

};

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 << '\n';
        return os;
}

template<class T>
vertex<T> operator+(vertex<T> lhs,vertex<T> rhs){
   vertex<T> res;
   res.x = lhs.x + rhs.x;
   res.y = lhs.y + rhs.y;
   return res;
}

template<class T>
bool operator == (vertex<T> a, vertex<T> b) {
        return (a.x == b.x && a.y == b.y);
}

template<class T>
vertex<T>& operator/= (vertex<T>& vertex, int number) {
   vertex.x = vertex.x / number;
   vertex.y = vertex.y / number;
   return vertex;
}

#endif
```

Templates.h

```
#ifndef D_TEMPLATES_H_
#define D_TEMPLATES_H_ 1

#include <tuple>
#include <type_traits>

#include "rhombus.h"
#include "rectangle.h"
#include "trapezoid.h"
#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 Type, size_t SIZE>
struct is_figurelike_tuple<std::array<Type, SIZE>> :
    is_vertex<Type> {};

template<class T>
inline constexpr bool is_figurelike_tuple_v =
  is_figurelike_tuple<T>::value;

template<class T, class = void>
struct has_print_method : std::false_type {};

template<class T>
struct has_print_method<T,
  std::void_t<decltype(std::declval<const T>().print())>> :
    std::true_type {};

template<class T>
inline constexpr bool has_print_method_v =
  has_print_method<T>::value;

template<class T>
std::enable_if_t<has_print_method_v<T>, void>
    print(const T& figure) {
        figure.print();
}

template<size_t ID, class T>
void single_print(const T& t) {
    std::cout << std::get<ID>(t);
    return ;
}

template<size_t ID, class T>
void Recursiveprint(const T& t) {
    if constexpr (ID < std::tuple_size_v<T>){
        single_print<ID>(t);
        Recursiveprint<ID+1>(t);
        return ;
    }
    return;
}

template<class T>
std::enable_if_t<is_figurelike_tuple_v<T>, void>
    print(const T& fake) {
    return Recursiveprint<0>(fake);
}

template<class T, class = void>
struct has_center_method : std::false_type {};

template<class T>
struct has_center_method<T,
        std::void_t<decltype(std::declval<const T>().center())>> :
        std::true_type {};

template<class T>
inline constexpr bool has_center_method_v =
        has_center_method<T>::value;

template<class T>
std::enable_if_t<has_center_method_v<T>, vertex<double>>
center(const T& figure) {
    return figure.center();
}

template<class T>
inline constexpr const int tuple_size_v = std::tuple_size<T>::value;

template<size_t ID, class T>
vertex<double> single_center(const T& t) {
    vertex<double> v;
    v.x = std::get<ID>(t).x;
    v.y = std::get<ID>(t).y;
    v /= std::tuple_size_v<T>;
    return v;
}

template<size_t ID, class T>
vertex<double> Recursivecenter(const T& t) {
    if constexpr (ID < std::tuple_size_v<T>){
        return  single_center<ID>(t) + Recursivecenter<ID+1>(t);
    } else {
        vertex<double> v;
        v.x = 0;
        v.y = 0;
        return v;
    }
}

template<class T>
std::enable_if_t<is_figurelike_tuple_v<T>, vertex<double>>
center(const T& fake) {
    return Recursivecenter<0>(fake);
}

template<class T, class = void>
struct has_area_method : std::false_type {};

template<class T>
struct has_area_method<T,
        std::void_t<decltype(std::declval<const T>().area())>> :
        std::true_type {};

template<class T>
inline constexpr bool has_area_method_v =
        has_area_method<T>::value;

template<class T>
std::enable_if_t<has_area_method_v<T>, double>
area(const T& figure) {
    return figure.area();
}

template<size_t ID, class T>
double single_area(const T& t) {
    const auto& a = std::get<0>(t);
    const auto& b = std::get<ID - 1>(t);
    const auto& c = std::get<ID>(t);
    const double dx1 = b.x - a.x;
    const double dy1 = b.y - a.y;
    const double dx2 = c.x - a.x;
    const double dy2 = c.y - a.y;
    return std::abs(dx1 * dy2 - dy1 * dx2) * 0.5;
}

template<size_t ID, class T>
```

```
double Recursivearea(const T& t) {
    if constexpr (ID < std::tuple_size_v<T>){
        return single_area<ID>(t) + Recursivearea<ID + 1>(t);
    }
    return 0;
}

template<class T>
std::enable_if_t<is_figurelike_tuple_v<T>, double>
area(const T& fake) {
    return Recursivearea<2>(fake);
}

#endif // D_TEMPLATES_H_
```

Rectangle.h

```
#ifndef D_RECTANGLE_H_
#define D_RECTANGLE_H_ 1

#include <algorithm>
#include <iostream>

#include "vertex.h"
#include "vector.h"

template<class T>
struct rectangle {
        vertex<T> vertices[4];

        rectangle(std::istream& is);

        vertex<double> center() const;

        double area() const;
        void print() const;

};

template<class T>
rectangle<T>::rectangle(std::istream& is) {
        for(int i = 0; i < 4; ++i){
                is >> vertices[i];
        }
```

```
    if (isPerpendicular(Vector< vertex<T> >(vertices[0], vertices[1]), Vector<
vertex<T> >(vertices[0], vertices[3])) && isPerpendicular(Vector< vertex<T>
>(vertices[0], vertices[1]), Vector< vertex<T> >(vertices[1], vertices[2])) &&
        isPerpendicular(Vector< vertex<T> >(vertices[1], vertices[2]),
Vector< vertex<T> >(vertices[2], vertices[3])) && isPerpendicular(Vector<
vertex<T> >(vertices[2], vertices[3]), Vector< vertex<T> >(vertices[0],
vertices[3]))) {



    } else if (isPerpendicular(Vector< vertex<T> >(vertices[0], vertices[3]),
Vector< vertex<T> >(vertices[3], vertices[1])) && isPerpendicular(Vector<
vertex<T> >(vertices[3], vertices[1]), Vector< vertex<T> >(vertices[1],
vertices[2])) &&
        isPerpendicular(Vector< vertex<T> >(vertices[1], vertices[2]),
Vector< vertex<T> >(vertices[2], vertices[0])) && isPerpendicular(Vector<
vertex<T> >(vertices[0], vertices[2]), Vector< vertex<T> >(vertices[0],
vertices[3]))) {

            vertex<T> tmp;
            tmp = vertices[0];
            vertices[0] = vertices[3];
            vertices[3] = tmp;

    } else if (isPerpendicular(Vector< vertex<T> >(vertices[0], vertices[1]),
Vector< vertex<T> >(vertices[1], vertices[3])) && isPerpendicular(Vector<
vertex<T> >(vertices[1], vertices[3]), Vector< vertex<T> >(vertices[3],
vertices[2])) &&
        isPerpendicular(Vector< vertex<T> >(vertices[3], vertices[2]),
Vector< vertex<T> >(vertices[2], vertices[0])) && isPerpendicular(Vector<
vertex<T> >(vertices[2], vertices[0]), Vector< vertex<T> >(vertices[0],
vertices[1]))) {

            vertex<T> tmp;
            tmp = vertices[2];
            vertices[2] = vertices[3];
            vertices[3] = tmp;

    } else if (vertices[0] == vertices[1] || vertices[0] == vertices[2] || vertices[0]
== vertices[3] || vertices[1] == vertices[2] || vertices[1] == vertices[3] || vertices[2]
== vertices[3]) {
        throw std::logic_error("No points are able to be equal");
    } else {
        throw std::logic_error("That's not a Rectangle, sides are not
Perpendicular");
```

```
        }

        if (!(Vector< vertex<T> >(vertices[0], vertices[1]).length() == Vector<
vertex<T> >(vertices[2], vertices[3]).length() && Vector< vertex<T>
>(vertices[1], vertices[2]).length() == Vector< vertex<T> >(vertices[0],
vertices[3]).length())) {
                throw std::logic_error("That's not a Rectangle, sides are not equal");
        }
}

template<class T>
double rectangle<T>::area() const {
        return Vector< vertex<T> >(vertices[0], vertices[1]).length() * Vector<
vertex<T> >(vertices[1], vertices[2]).length();
}

template<class T>
void rectangle<T>::print() const {

        std::cout << vertices[0] << vertices[1] << vertices[2] << vertices[3] << '\n';

}

template<class T>
vertex<double> rectangle<T>::center() const {
        vertex<double> p;
        p.x = (vertices[0].x + vertices[1].x + vertices[2].x + vertices[3].x) / 4;
        p.y = (vertices[0].y + vertices[1].y + vertices[2].y + vertices[3].y) / 4;
        return p;
}

#endif
```

Rhombus.h

```
#ifndef D_RHOMBUS_H_
#define D_RHOMBUS_H_ 1

#include <algorithm>
#include <iostream>

#include "vertex.h"
#include "vector.h"

template<class T>
struct rhombus {
        vertex<T>  vertices[4];

        rhombus(std::istream& is);

        vertex<double>  center() const;

        double area() const;
        void print() const;

};

template<class T>
rhombus<T>::rhombus(std::istream& is) {
        for(int i = 0; i < 4; ++i){
                is >> vertices[i];
        }

        if (Vector< vertex<T> >(vertices[0], vertices[1]).length() == Vector<
vertex<T> >(vertices[1], vertices[2]).length() && Vector< vertex<T>
>(vertices[1], vertices[2]).length() == Vector< vertex<T> >(vertices[2],
vertices[3]).length()
         && Vector< vertex<T> >(vertices[0], vertices[1]).length() == Vector<
vertex<T> >(vertices[0], vertices[3]).length()) {

        } else if (Vector< vertex<T> >(vertices[0], vertices[1]).length() == Vector<
vertex<T> >(vertices[1], vertices[3]).length() && Vector< vertex<T>
>(vertices[1], vertices[3]).length() == Vector< vertex<T> >(vertices[2],
vertices[3]).length()
         && Vector< vertex<T> >(vertices[0], vertices[1]).length() == Vector<
vertex<T> >(vertices[0], vertices[2]).length()) {
                vertex<T> tmp = vertices[3];
                vertices[3] = vertices[2];
                vertices[2] = tmp;
        } else if (Vector< vertex<T> >(vertices[0], vertices[2]).length() == Vector<
vertex<T> >(vertices[3], vertices[2]).length() && Vector< vertex<T>
>(vertices[3], vertices[2]).length() == Vector< vertex<T> >(vertices[1],
vertices[3]).length()
         && Vector< vertex<T> >(vertices[0], vertices[1]).length() == Vector<
vertex<T> >(vertices[0], vertices[2]).length()) {
                vertex<T> tmp = vertices[3];
                vertices[3] = vertices[2];
                vertices[2] = tmp;
        } else if (vertices[0] == vertices[1] || vertices[0] == vertices[2] || vertices[0]
== vertices[3] || vertices[1] == vertices[2] || vertices[1] == vertices[3] || vertices[2]
== vertices[3]) {
                throw std::logic_error("No points are able to be equal");
        } else {
                throw std::logic_error("This is not a Rhombus, sides are not equal");
        }


        Vector< vertex<T> > v1(vertices[0], vertices[1]);
        Vector< vertex<T> > v2(vertices[1], vertices[2]);
        Vector< vertex<T> > v3(vertices[2], vertices[3]);
        Vector< vertex<T> > v4(vertices[3], vertices[0]);

        double cos1 = v1 * v2 / (v1.length() * v2.length());
        double cos2 = v2 * v3 / (v2.length() * v3.length());
        double cos3 = v3 * v4 / (v3.length() * v4.length());
        double cos4 = v1 * v4 / (v1.length() * v4.length());

        if (cos1 != cos3 || cos2 != cos4) {
                throw std::logic_error("This is not a Rhombus, opposite angles are not
equal");
        }
}

template<class T>
double rhombus<T>::area() const {
        return Vector< vertex<T> >(vertices[0], vertices[2]).length() * Vector<
vertex<T> >(vertices[1], vertices[3]).length() / 2;
}

template<class T>
void rhombus<T>::print() const {

        std::cout << vertices[0] << vertices[1] << vertices[2] << vertices[3] << '\n';
}

template<class T>
vertex<double> rhombus<T>::center() const {
        vertex<double> p;
        p.x = (vertices[0].x + vertices[1].x + vertices[2].x + vertices[3].x) / 4;
        p.y = (vertices[0].y + vertices[1].y + vertices[2].y + vertices[3].y) / 4;
        return p;
}

#endif // D_TRIANGLE_H_
```

Trapezoid.h

```
#ifndef D_TRAPEZOID_H_
#define D_TRAPEZOID_H_ 1

#include <algorithm>
#include <iostream>

#include "vertex.h"
#include "vector.h"

template<class T>
struct trapezoid {
	vertex<T> vertices[4];

	trapezoid(std::istream& is);

	vertex<double>  center() const;

	double area() const;
	void print() const;

};

template<class T>
trapezoid<T>::trapezoid(std::istream& is) {
	for(int i = 0; i < 4; ++i){
		is >> vertices[i];
	}

	if (isParallel(Vector< vertex<T> >(vertices[0], vertices[3]), Vector<
vertex<T> >(vertices[1], vertices[2]))) {

	} else if (isParallel(Vector< vertex<T> >(vertices[0], vertices[2]), Vector<
vertex<T> >(vertices[3], vertices[1]))) {

		vertex<T>  tmp;
		tmp = vertices[1];
		vertices[1] = vertices[3];
		vertices[3] = tmp;
		tmp = vertices[2];
		vertices[2] = vertices[3];
		vertices[3] = tmp;

	} else if (isParallel(Vector< vertex<T> >(vertices[0], vertices[2]), Vector<
vertex<T> >(vertices[1], vertices[3]))) {
        vertex<T>  tmp;
        tmp = vertices[2];
        vertices[2] = vertices[3];
        vertices[3] = tmp;

    } else if (vertices[0] == vertices[1] || vertices[0] == vertices[2] || vertices[0]
== vertices[3] || vertices[1] == vertices[2] || vertices[1] == vertices[3] || vertices[2]
== vertices[3]) {
        throw std::logic_error("No points are able to be equal");
    } else {
        throw std::logic_error("At least 2 sides of trapeze must be parallel");
    }
}

template<class T>
double trapezoid<T>::area() const {

    double a = vertices[1].y - vertices[2].y;
  double b = vertices[2].x - vertices[1].x;
  double c = vertices[1].x * vertices[2].y - vertices[2].x * vertices[1].y;
  double height = (std::abs(a * vertices[0].x + b * vertices[0].y + c) / sqrt(a * a + b
* b));

  Vector< vertex<T> > v1(vertices[0], vertices[1]);
  Vector< vertex<T> > v2(vertices[2], vertices[3]);

  return (v1.length() + v2.length()) * height / 2;

}

template<class T>
void trapezoid<T>::print() const {

    std::cout << vertices[0] << vertices[1] << vertices[2] << vertices[3] << '\n';

}


template<class T>
vertex<double> trapezoid<T>::center() const {
    vertex<double>  p;
    p.x = (vertices[0].x + vertices[1].x + vertices[2].x + vertices[3].x) / 4;
    p.y = (vertices[0].y + vertices[1].y + vertices[2].y + vertices[3].y) / 4;
    return p;
}
```

```
#endif
```

Vector.h

```
#ifndef VECTOR_H_
#define VECTOR_H_

#include "vertex.h"
#include <cmath>
#include <numeric>
#include <limits>

template<class T>
struct Vector {
	explicit Vector(T a, T b);
	double length() const;
	double x;
	double y;
	double operator* (Vector b) ;
	bool operator== (Vector b);
};

template<class T>
Vector<T>::Vector(T a, T b) {
	x = b.x - a.x;
	y = b.y - a.y;
}

template<class T>
double Vector<T>::length() const{
	return sqrt(x * x + y * y);
}

template<class T>
double Vector<T>::operator* (Vector<T> b) {
	return x * b.x + y * b.y;
}

template<class T>
bool Vector<T>::operator== (Vector<T> b) {
	return std::abs(x - b.x) < std::numeric_limits<double>::epsilon() * 100
	&& std::abs(y - b.y) < std::numeric_limits<double>::epsilon() * 100;
}
```

```
template<class T>
bool isParallel(const Vector<T> a, const Vector<T> b) {
    return (a.x * b.y - a.y * b.y) == 0;
}

template<class T>
bool isPerpendicular(const Vector<T> a, const Vector<T> b) {
    return (a.x * b.x + a.y * b.y) == 0;
}


#endif
```

File01.test

```
1
2
3
2
-20 0
20 0
20 10
-20 10
0
```

File02.test

```
1
2
3
1
-10 2
0 0
10 2
0 4
0
```

Source.cpp

```
#include "rhombus.h"
#include "rectangle.h"
#include "trapezoid.h"
#include "templates.h"
#include "vertex.h"

void menu() {
    std::cout << "______________________________\n";
  std::cout << "0: Exit\n";
```

```
    std::cout << "1: Fake figure\n";
    std::cout << "2: Array figure\n";
    std::cout << "3: Real figure\n";
}

void menuOf3() {
    std::cout << "______________________________\n";
    std::cout << "0: Exit\n";
    std::cout << "1: Rhombus\n";
    std::cout << "2: Rectangle\n";
    std::cout << "3: Trapezoid\n";
}

int main() {

        int cmd;

        while (true) {
            menu();
            std::cin >> cmd;
            if (cmd == 0) break;
            else if (cmd == 1) {
                std::cout << "Fake rhombus : float\n";
                std::tuple<vertex<float>, vertex<float>, vertex<float>, vertex<float>>
fakeRhombus{{0, 0}, {-1.5, 2}, {1.5, 2}, {0, 4}};

          std::cout << "Coordinates: \n";
          print(fakeRhombus);
          std::cout << '\n';
          std::cout << "Center: " << center(fakeRhombus) << '\n';
          std::cout << "Area: " << area(fakeRhombus) << '\n';
            } else if (cmd == 2) {
                std::cout << "Array rectangle : double\n";
                std::array<vertex<double>, 4> arrayRectangle{{{0, 0}, {10, 0}, {0, 8},
{10, 8}}};

                std::cout << "Coordinates: \n";
          print(arrayRectangle);
          std::cout << '\n';
          std::cout << "Center: " << center(arrayRectangle) << '\n';
          std::cout << "Area: " << area(arrayRectangle) << '\n';
            } else if (cmd == 3) {
                menuOf3();
                int cmdcmd;
                std::cin >> cmdcmd;

                if (cmdcmd == 0) break;
                else if (cmdcmd == 1) {
                    std::cout << "Input 4 coordinates of rhombus" << std::endl;
              rhombus<double> realRhombus(std::cin);
              std::cout << "Coordinates: \n";
              print(realRhombus);
              std::cout << '\n';
              std::cout << "Center: " << center(realRhombus) << '\n';
              std::cout << "Area: " << area(realRhombus) << '\n';
                } else if (cmdcmd == 2) {
                    std::cout << "Input 4 coordinates of rectangle" << std::endl;
              rectangle<double> realRectangle(std::cin);
              std::cout << "Coordinates: \n";
              print(realRectangle);
              std::cout << '\n';
              std::cout << "Center: " << center(realRectangle) << '\n';
              std::cout << "Area: " << area(realRectangle) << '\n';
                } else if (cmdcmd == 3) {
                    std::cout << "Input 4 coordinates of trapezoid" << std::endl;
              trapezoid<double> realTrapezoid(std::cin);
              std::cout << "Coordinates: \n";
              print(realTrapezoid);
              std::cout << '\n';
              std::cout << "Center: " << center(realTrapezoid) << '\n';
              std::cout << "Area: " << area(realTrapezoid) << '\n';
                } else {
                    std::cout << "Not a command\n";
                }

            } else {
                std::cout << "Not a command\n";
            }
        }

    return 0;
}
```

Результаты тестов

1:

```
0: Exit
1: Fake figure
2: Array figure
```

```
3: Real figure
1
Fake rhombus : float
Coordinates:
0 0
-1.5 2
1.5 2
0 4

Center: 0 2

Area: 6
______________________________
0: Exit
1: Fake figure
2: Array figure
3: Real figure
2
Array rectangle : double
Coordinates:
0 0
10 0
0 8
10 8

Center: 5 4

Area: 80
______________________________
0: Exit
1: Fake figure
2: Array figure
3: Real figure
3
______________________________
0: Exit
1: Rhombus
2: Rectangle
3: Trapezoid
2
Input 4 coordinates of rectangle
-20 04□ □
20 0
20 10
-20 10
```

```
Coordinates:
-20 0
20 0
20 10
-20 10


Center: 0 5

Area: 400
______________________________
0: Exit
1: Fake figure
2: Array figure
3: Real figure
0
```

2:

```
______________________________
0: Exit
1: Fake figure
2: Array figure
3: Real figure
1
Fake rhombus : float
Coordinates:
0 0
-1.5 2
1.5 2
0 4

Center: 0 2

Area: 6
______________________________
0: Exit
1: Fake figure
2: Array figure
3: Real figure
2
Array rectangle : double
Coordinates:
0 0
10 0
```

```
0 8
10 8

Center: 5 4

Area: 80
______________________________
0: Exit
1: Fake figure
2: Array figure
3: Real figure
3
______________________________
0: Exit
1: Rhombus
2: Rectangle
3: Trapezoid
1
Input 4 coordinates of rhombus
-10 2
0 0
10 2
0 4
Coordinates:
-10 2
0 0
10 2
0 4


Center: 0 2

Area: 40
______________________________
0: Exit
1: Fake figure
2: Array figure
3: Real figure
0
```

Объяснение результатов

Программа получает на вход команды из меню. В зависимости от команды совершается одно из действий: выход, обработка фигуры из кортежа, обработка фигуры из массива, обработка фигуры из стандартного ввода.

Вывод

Были изучены основы метапрограммирования, применены в лабораторной работе. Применение шаблонов значительно расширяет возможности программы. Шаблоны сложны в изучении, однако будут очень полезны в практической деятельности и иногда незаменимы при написании программного кода.