

Московский авиационный институт  
(Национальный исследовательский университет)  
Факультет "Информационные технологии и прикладная математика"

**Лабораторная работа №4 по курсу  
“Объектно-ориентированное программирование”**

*Студент:* Живалев Е.А.

*Группа:* М8О-206Б

*Преподаватель:* Журавлев А.А.

*Вариант:* 5

*Оценка:* \_\_\_\_\_

*Дата:* \_\_\_\_\_

Москва  
2019

# 1 Исходный код

Ссылка на github : [https://github.com/QElderDelta/oop\\_exercise\\_04](https://github.com/QElderDelta/oop_exercise_04)

## vertex.hpp

```
1 #pragma once
2
3 #include <iostream>
4 #include <cmath>
5 #include <iomanip>
6
7 template <class T>
8 struct vertex_t {
9     T x;
10    T y;
11 };
12
13 template<class T>
14 std::istream& operator>>(std::istream& is, vertex_t<T>& p) {
15     is >> p.x >> p.y;
16     return is;
17 }
18
19 template<class T>
20 std::ostream& operator<<(std::ostream& os, const vertex_t<T>& p) {
21     os << std::fixed << std::setprecision(3) << "[" << p.x << ",
22     " << p.y << "]" ;
23     return os;
24 }
25
26 template<class T>
27 T calculateDistance(const vertex_t<T>& p1, const vertex_t<T>& p2)
28 {
29     return sqrt(pow(p2.x - p1.x, 2) + pow(p2.y - p1.y, 2));
30 }
31
32 template<class T>
33 T triangleArea(vertex_t<T> p1, vertex_t<T> p2, vertex_t<T> p3) {
34     return 0.5 * fabs((p1.x - p3.x) * (p2.y - p3.y) - (p2.x - p3.x
35     ) * (p1.y
36     - p3.y));
37 }
```

## rhombus.hpp

```
1 #pragma once
2
3 #include <array>
4
5 #include "vertex.hpp"
6
7 template<class T>
8 double checkIfRhombus(const vertex_t<T> p1, const vertex_t<T>& p2,
9     const vertex_t<T>& p3, const vertex_t<T>& p4) {
10     T d1 = calculateDistance(p1, p2);
11     T d2 = calculateDistance(p1, p3);
12     T d3 = calculateDistance(p1, p4);
13     if(d1 == d2) {
```

```

14         return d3;
15     } else if(d1 == d3) {
16         return d2;
17     } else if(d2 == d3) {
18         return d1;
19     } else {
20         throw std::invalid_argument("Entered coordinates are not
forming Rhombus. Try entering new coordinates");
21     }
22 }
23
24 template <class T>
25 struct Rhombus {
26     std::array<vertex_t<T>, 4> points;
27     T smallerDiagonal, biggerDiagonal;
28     Rhombus(const vertex_t<T>& p1, const vertex_t<T>& p2, const
vertex_t<T>& p3,
29             const vertex_t<T>& p4);
30     double area() const;
31     vertex_t<T> center() const;
32     void print(std::ostream& os) const;
33 };
34
35 template<class T>
36 Rhombus<T>::Rhombus(const vertex_t<T>& p1, const vertex_t<T>& p2,
37                     const vertex_t<T>& p3, const vertex_t<T>& p4) {
38     try {
39         T d1 = checkIfRhombus(p1, p2, p3, p4);
40         T d2 = checkIfRhombus(p2, p1, p3, p4);
41         T d3 = checkIfRhombus(p3, p1, p2, p4);
42         T d4 = checkIfRhombus(p4, p1, p2, p3);
43         if(d1 == d2 || d1 == d4) {
44             if(d1 < d3) {
45                 smallerDiagonal = d1;
46                 biggerDiagonal = d3;
47             } else {
48                 smallerDiagonal = d3;
49                 biggerDiagonal = d1;
50             }
51         } else if(d1 == d3) {
52             if(d1 < d2) {
53                 smallerDiagonal = d1;
54                 biggerDiagonal = d2;
55             } else {
56                 smallerDiagonal = d2;
57                 biggerDiagonal = d1;
58             }
59         }
60     } catch(std::exception& e) {
61         throw std::invalid_argument(e.what());
62         return;
63     }
64     points[0] = p1;
65     points[1] = p2;
66     points[2] = p3;
67     points[3] = p4;
68 }
69
70 template<class T>

```

```

71 double Rhombus<T>::area() const {
72     return smallerDiagonal * biggerDiagonal / 2.0;
73 }
74
75 template<class T>
76 vertex_t<T> Rhombus<T>::center() const {
77     if (calculateDistance(points[0], points[1]) == smallerDiagonal
78         ||
79         calculateDistance(points[0], points[1]) ==
80         biggerDiagonal) {
81         return {((points[0].x + points[1].x) / 2.0), ((points[0].y
82             + points[1].y) / 2.0)};
83     } else if (calculateDistance(points[0], points[2]) ==
84         smallerDiagonal ||
85         calculateDistance(points[0], points[2]) ==
86         biggerDiagonal) {
87         return {((points[0].x + points[2].x) / 2.0), ((points[0].y
88             + points[2].y) / 2.0)};
89     } else {
90         return {((points[0].x + points[3].x) / 2.0), ((points[0].y
91             + points[3].y) / 2.0)};
92     }
93 }
94
95 template<class T>
96 void Rhombus<T>::print(std::ostream& os) const {
97     os << "Rhombus: ";
98     for (const auto& p : points) {
99         os << p << ' ';
100     }
101     os << std::endl;
102 }

```

## pentagon.hpp

```

1 #pragma once
2
3 #include <array>
4 #include <cmath>
5
6 #include "vertex.hpp"
7
8 template <class T>
9 struct Pentagon {
10     std::array<vertex_t<T>, 5> points;
11     Pentagon(const vertex_t<T>& p1, const vertex_t<T>& p2, const
12         vertex_t<T>& p3,
13         const vertex_t<T>& p4, const vertex_t<T>& p5);
14     double area() const;
15     vertex_t<T> center() const;
16     void print(std::ostream& os) const;
17 };
18
19 template <class T>
20 Pentagon<T>::Pentagon(const vertex_t<T>& p1, const vertex_t<T>& p2
21     ,
22     const vertex_t<T>& p3, const vertex_t<T>& p4, const
23     vertex_t<T>& p5) {
24     points[0] = p1;

```

```

22     points[1] = p2;
23     points[2] = p3;
24     points[3] = p4;
25     points[4] = p5;
26 }
27
28 template <class T>
29 double Pentagon<T>::area() const {
30     double result = 0;
31     for(unsigned i = 0; i < points.size(); ++i) {
32         vertex_t p1 = i ? points[i - 1] : points[points.size() -
1];
33         vertex_t p2 = points[i];
34         result += (p1.x - p2.x) * (p1.y + p2.y);
35     }
36     return fabs(result) / 2.0;
37 }
38
39 template <class T>
40 vertex_t<T> Pentagon<T>::center() const {
41     T x = 0;
42     T y = 0;
43     for(const auto& p : points) {
44         x += p.x;
45         y += p.y;
46     }
47     x /= points.size();
48     y /= points.size();
49     return {x, y};
50 }
51
52 template <class T>
53 void Pentagon<T>::print(std::ostream& os) const {
54     os << "Pentagon: ";
55     for(const auto& p : points) {
56         os << p << ' ';
57     }
58     os << std::endl;
59 }

```

## hexagon.hpp

```

1 #pragma once
2
3 #include <array>
4 #include <cmath>
5
6 #include "vertex.hpp"
7
8 template <class T>
9 struct Hexagon {
10     std::array<vertex_t<T>, 6> points;
11     Hexagon(const vertex_t<T>& p1, const vertex_t<T>& p2, const
vertex_t<T>& p3,
12             const vertex_t<T>& p4, const vertex_t<T>& p5, const
vertex_t<T>& p6);
13     double area() const;
14     vertex_t<T> center() const;
15     void print(std::ostream& os) const;

```

```

16 };
17
18 template<class T>
19 Hexagon<T>::Hexagon(const vertex_t<T>& p1, const vertex_t<T>& p2,
20     const vertex_t<T>& p3, const vertex_t<T>& p4, const
21     vertex_t<T>& p5,
22     const vertex_t<T>& p6) {
23     points[0] = p1;
24     points[1] = p2;
25     points[2] = p3;
26     points[3] = p4;
27     points[4] = p5;
28     points[5] = p6;
29 }
30
31 template<class T>
32 double Hexagon<T>::area() const {
33     double result = 0;
34     for(unsigned i = 0; i < points.size(); ++i) {
35         vertex_t p1 = i ? points[i - 1] : points[points.size() -
36         1];
37         vertex_t p2 = points[i];
38         result += (p1.x - p2.x) * (p1.y + p2.y);
39     }
40     return fabs(result) / 2.0;
41 }
42
43 template<class T>
44 vertex_t<T> Hexagon<T>::center() const {
45     T x = 0;
46     T y = 0;
47     for(const auto& p : points) {
48         x += p.x;
49         y += p.y;
50     }
51     x /= points.size();
52     y /= points.size();
53     return {x, y};
54 }
55
56 template<class T>
57 void Hexagon<T>::print(std::ostream& os) const {
58     os << "Hexagon: ";
59     for(const auto& p : points) {
60         os << p << ' ';
61     }
62     os << std::endl;
63 }

```

## templates.hpp

```

1 #pragma once
2
3 #include <iostream>
4 #include <type_traits>
5 #include <tuple>
6
7 #include "vertex.hpp"
8

```

```

9  template<class T>
10 struct is_vertex : std::false_type {};
11
12 template<class T>
13 struct is_vertex<vertex_t<T>> : std::true_type {};
14
15 template<class T>
16 struct is_figurelike_tuple : std::false_type {};
17
18 template<class Head, class... Tail>
19 struct is_figurelike_tuple<std::tuple<Head, Tail...>> :
20     std::conjunction<is_vertex<Head>,
21         std::is_same<Head, Tail>...> {};
22
23 template<class Type, size_t SIZE>
24 struct is_figurelike_tuple<std::array<Type, SIZE>> :
25     is_vertex<Type> {};
26
27 template<class T>
28 inline constexpr bool is_figurelike_tuple_v =
29     is_figurelike_tuple<T>::value;
30
31 template<class T, class = void>
32 struct has_area_method : std::false_type {};
33
34 template<class T>
35 struct has_area_method<T,
36     std::void_t<decltype(std::declval<const T>().area())>> :
37     std::true_type {};
38
39 template<class T>
40 inline constexpr bool has_area_method_v =
41     has_area_method<T>::value;
42
43 template<class T>
44 std::enable_if_t<has_area_method_v<T>, double>
45     area(const T& figure) {
46     return figure.area();
47 }
48
49 template<class T, class = void>
50 struct has_print_method : std::false_type {};
51
52
53 template<class T>
54 struct has_print_method<T,
55     std::void_t<decltype(std::declval<const T>().print(std::cout))>>
56     :
57     std::true_type {};
58
59 template<class T>
60 inline constexpr bool has_print_method_v =
61     has_print_method<T>::value;
62
63 template<class T>
64 std::enable_if_t<has_print_method_v<T>, void>
65     print(const T& figure, std::ostream& os) {
66     figure.print(os);
67 }

```

```

67
68 template<class T, class = void>
69 struct has_center_method : std::false_type {};
70
71 template<class T>
72 struct has_center_method<T,
73     std::void_t<decltype(std::declval<const T>().center())>> :
74     std::true_type {};
75
76 template<class T>
77 inline constexpr bool has_center_method_v =
78     has_center_method<T>::value;
79
80 template<class T>
81 std::enable_if_t<has_center_method_v<T>, vertex_t<decltype(std::
82     declval<const T>().center().x)>>
83     center(const T& figure) {
84     return figure.center();
85 }
86
87 template<size_t ID, class T>
88 double single_area(const T& t) {
89     const auto& a = std::get<0>(t);
90     const auto& b = std::get<ID - 1>(t);
91     const auto& c = std::get<ID>(t);
92     const double dx1 = b.x - a.x;
93     const double dy1 = b.y - a.y;
94     const double dx2 = c.x - a.x;
95     const double dy2 = c.y - a.y;
96     return std::abs(dx1 * dy2 - dy1 * dx2) * 0.5;
97 }
98
99 template<size_t ID, class T>
100 double recursive_area(const T& t) {
101     if constexpr (ID < std::tuple_size_v<T>){
102         return single_area<ID>(t) + recursive_area<ID + 1>(t);
103     } else {
104         return 0;
105     }
106 }
107
108 template<class T>
109 std::enable_if_t<is_figurlike_tuple_v<T>, double>
110 area(const T& fake) {
111     return recursive_area<2>(fake);
112 }
113
114 template<size_t ID, class T>
115 double single_center_x(const T& t) {
116     return std::get<ID>(t).x / std::tuple_size_v<T>;
117 }
118
119 template<size_t ID, class T>
120 double single_center_y(const T& t) {
121     return std::get<ID>(t).y / std::tuple_size_v<T>;
122 }
123
124 template<size_t ID, class T>
125 double recursive_center_x(const T& t) {

```



```

125     if constexpr (ID < std::tuple_size_v<T>) {
126         return single_center_x<ID>(t) + recursive_center_x<ID +
127     1>(t);
128     } else {
129         return 0;
130     }
131 }
132
132 template<size_t ID, class T>
133 double recursive_center_y(const T& t) {
134     if constexpr (ID < std::tuple_size_v<T>) {
135         return single_center_y<ID>(t) + recursive_center_y<ID +
136     1>(t);
137     } else {
138         return 0;
139     }
140 }
141
141 template<class T>
142 std::enable_if_t<is_figurelike_tuple_v<T>, vertex_t<double>>
143 center(const T& tup) {
144     return {recursive_center_x<0>(tup), recursive_center_y<0>(
145     tup)};
146 }
147
147 template<size_t ID, class T>
148 void single_print(const T& t, std::ostream& os) {
149     os << std::get<ID>(t) << ' ';
150 }
151
151 template<size_t ID, class T>
152 void recursive_print(const T& t, std::ostream& os) {
153     if constexpr (ID < std::tuple_size_v<T>) {
154         single_print<ID>(t, os);
155         recursive_print<ID + 1>(t, os);
156     } else {
157         return;
158     }
159 }
160 }
161
161 template<class T>
162 std::enable_if_t<is_figurelike_tuple_v<T>, void>
163 print(const T& tup, std::ostream& os) {
164     recursive_print<0>(tup, os);
165     os << std::endl;
166 }
167 }

```

## main.cpp

```

1 #include <iostream>
2
3 #include "vertex.hpp"
4 #include "templates.hpp"
5 #include "rhombus.hpp"
6 #include "pentagon.hpp"
7 #include "hexagon.hpp"
8
9 int main() {
10     int command;

```

```

11     std::cout << "1 - Rhombus" << std::endl;
12     std::cout << "2 - Pentagon" << std::endl;
13     std::cout << "3 - Hexagon" << std::endl;
14     std::cout << "0 - Exit" << std::endl;
15     std::cin >> command;
16     while(command != 0) {
17         if(command == 1) {
18             vertex_t<double> p1, p2, p3, p4;
19             std::cin >> p1 >> p2 >> p3 >> p4;
20             try {
21                 Rhombus r{p1, p2, p3, p4};
22             } catch(std::exception& e) {
23                 std::cout << e.what() << std::endl;
24                 std::cin >> command;
25                 continue;
26             }
27             Rhombus r{p1, p2, p3, p4};
28             print(r, std::cout);
29             std::cout << area(r) << std::endl;
30             std::cout << center(r) << std::endl;
31             std::tuple<vertex_t<double>, vertex_t<double>,
vertex_t<double>,
32                 vertex_t<double>> r1{p1, p2, p3, p4};
33             std::cout << "Rhombus: ";
34             print(r1, std::cout);
35             std::cout << area(r1) << std::endl;
36             std::cout << center(r1) << std::endl;
37         } else if(command == 2) {
38             vertex_t<double> p1, p2, p3, p4, p5;
39             std::cin >> p1 >> p2 >> p3 >> p4 >> p5;
40             Pentagon r{p1, p2, p3, p4, p5};
41             print(r, std::cout);
42             std::cout << area(r) << std::endl;
43             std::cout << center(r) << std::endl;
44             std::tuple<vertex_t<double>, vertex_t<double>,
vertex_t<double>,
45                 vertex_t<double>, vertex_t<double>> r1{p1, p2, p3,
p4, p5};
46             std::cout << "Pentagon: ";
47             print(r1, std::cout);
48             std::cout << area(r1) << std::endl;
49             std::cout << center(r1) << std::endl;
50         } else if(command == 3) {
51             vertex_t<double> p1, p2, p3, p4, p5, p6;
52             std::cin >> p1 >> p2 >> p3 >> p4 >> p5 >> p6;
53             Hexagon r{p1, p2, p3, p4, p5, p6};
54             print(r, std::cout);
55             std::cout << area(r) << std::endl;
56             std::cout << center(r) << std::endl;
57             std::tuple<vertex_t<double>, vertex_t<double>,
vertex_t<double>,
58                 vertex_t<double>, vertex_t<double>, vertex_t<
double>>
59                 r1{p1, p2, p3, p4, p5, p6};
60             std::cout << "Hexagon: ";
61             print(r1, std::cout);
62             std::cout << area(r1) << std::endl;
63             std::cout << center(r1) << std::endl;
64         } else {

```

```

65         std::cout << "Wrong command" << std::endl;
66     }
67     std::cin >> command;
68 }
69 }

```

## CMakeLists.txt

```

1 cmake_minimum_required(VERSION 3.1)
2
3 project(lab4)
4
5
6
7
8
9
10 add_executable(lab4
11     main.cpp
12     vertex.hpp
13     rhombus.hpp
14     pentagon.hpp
15     hexagon.hpp)
16
17 set_property(TARGET lab3 PROPERTY CXX_STANDARD 17)
18
19 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Wextra -Werror")

```

## meson.build

```

1 project('lab4', 'cpp')
2
3 add_project_arguments('-std=c++17', '-Wall', '-Wextra', language :
4     'cpp')
5 executable('lab4_meson', 'main.cpp')

```

## 2 Тестирование

### test\_01.txt:

Попробуем создать фигуру с координатами (1, 2), (1, 3), (1, 4), (1, 5), которая очевидно не является ромбом, рассчитывая получить сообщение об ошибке. Затем создадим ромб с координатами (-2, 0), (0, 2), (2, 0), (0, -2), площадь которого равна 8, а центр находится в точке (0, 0), а также пятиугольник с координатами (-2.000, 0.000), (-1.000, 1.000), (1.000, 1.000), (2.000, 0.000), (1.000, -1.000), площадь которого равна 5 и шестиугольник с координатами (-2.000, 0.000), (-1.000, 1.000), (1.000, 1.000), (2.000, 0.000), (1.000, -1.000), (-1.000, -1.000) с площадью равной 6.

Результат:

1 - Rhombus

2 - Pentagon

3 - Hexagon

0 - Exit

Entered coordinates are not forming Rhombus. Try entering new coordinates

Rhombus: [-2.000, 0.000] [0.000, 2.000] [2.000, 0.000] [0.000, -2.000]

8.000

[0.000, 0.000]

Rhombus: [-2.000, 0.000] [0.000, 2.000] [2.000, 0.000] [0.000, -2.000]

8.000

[0.000, 0.000] Pentagon: [-2.000, 0.000] [-1.000, 1.000] [1.000, 1.000] [2.000, 0.000] [1.000, -1.000]

5.000

[0.200, 0.200]

Pentagon: [-2.000, 0.000] [-1.000, 1.000] [1.000, 1.000] [2.000, 0.000] [1.000, -1.000]

5.000

[0.200, 0.200]

Hexagon: [-2.000, 0.000] [-1.000, 1.000] [1.000, 1.000] [2.000, 0.000] [1.000, -1.000] [-1.000, -1.000]

6.000

[0.000, 0.000]

Hexagon: [-2.000, 0.000] [-1.000, 1.000] [1.000, 1.000] [2.000, 0.000] [1.000, -1.000] [-1.000, -1.000]

6.000

[0.000, 0.000]

### test\_02.txt

Создадим ромб с координатами [4.000, 0.000], [8.000, 2.000], [12.000, 0.000], [8.000, -2.000], центром в точке [8, 0] и площадью равной 16, квадрат с координатами [4.000, 2.000], [8.000, 2.000], [8.000, -2.000], [4.000, -2.000] с центром в точке [6, 0] и площадью равной 16, пятиугольник с координатами [4.000, 0.000], [8.000, 2.000], [12.000, 0.000], [8.000, -2.000], [6.000, -2.000] и площадью равной 18, шестиугольник с координатами [4.000, 0.000], [8.000, 2.000], [10.000, 2.000], [12.000, 0.000], [8.000, -2.000], [6.000, -2.000] и площадью равной 20.

Результат:

1 - Rhombus  
 2 - Pentagon  
 3 - Hexagon  
 0 - Exit  
 Rhombus: [4.000, 0.000] [8.000, 2.000] [12.000, 0.000] [8.000, -2.000]  
 16.000  
 [8.000, 0.000]  
 Rhombus: [4.000, 0.000] [8.000, 2.000] [12.000, 0.000] [8.000, -2.000]  
 16.000  
 [8.000, 0.000]  
 Rhombus: [4.000, 2.000] [8.000, 2.000] [8.000, -2.000] [4.000, -2.000]  
 16.000  
 [6.000, 0.000]  
 Rhombus: [4.000, 2.000] [8.000, 2.000] [8.000, -2.000] [4.000, -2.000]  
 16.000  
 [6.000, 0.000]  
 Pentagon: [4.000, 0.000] [8.000, 2.000] [12.000, 0.000] [8.000, -2.000] [6.000, -  
 2.000]  
 18.000  
 [7.600, -0.400]  
 Pentagon: [4.000, 0.000] [8.000, 2.000] [12.000, 0.000] [8.000, -2.000] [6.000, -  
 2.000]  
 18.000  
 [7.600, -0.400]  
 Hexagon: [4.000, 0.000] [8.000, 2.000] [10.000, 2.000] [12.000, 0.000] [8.000, -  
 2.000] [6.000, -2.000]  
 20.000  
 [8.000, 0.000]  
 Hexagon: [4.000, 0.000] [8.000, 2.000] [10.000, 2.000] [12.000, 0.000] [8.000, -  
 2.000] [6.000, -2.000]  
 20.000  
 [8.000, 0.000]

### 3 Объяснение результатов работы программы

При вводе координат для создания ромба производится проверка этих координат, ведь они могут не образовывать ромб. Для этого реализована функция `checkIfRhombus`, которая вычисляет расстояния от одной точки до трёх остальных, а поскольку фигура является ромбом, то два из них должны быть равны. Третье же значение функция возвращает, ведь оно равно длине одной из диагоналей. Площадь ромба вычисляется как половина произведения диагоналей, центр - точка пересечения диагоналей. Методы вычисления площади и центра для пяти- и шестиугольника совпадают. Чтобы найти площадь необходимо перебрать все ребра и сложить площади трапеций, ограниченных этими ребрами. Чтобы найти центр необходимо разбить фигуры на треугольники (найти одну точку внутри фигуры), для каждого треугольника найти центроид и площадь и перемножить их, просуммировать полученные величины и разделить на общую площадь фигуры.

## 4 Выводы

На мой взгляд, метапрограммирование очень хорошо развито в плюсах. Я на своем примере увидел насколько меньше можно написать кода, если использовать предлагаемые для этого языком механизмы. Также я познакомился с системой для автоматизации сборки meson. По-моему, она имеет более приятный синтаксис, чем make, и, в большинстве, из-за этого она мне нравится больше.