Московский Авиационный Институт
(Национальный исследовательский Университет)

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

Лабораторная работа №4
по курсу «ООП»

Тема:
Наследование. Полиморфизм.

Студент:	Обыденкова Ю.Ю.
Группа:	М80-208Б-18
Преподаватель:	Журавлев А.А.
Вариант:	18
Оценка:	
Дата:	

Москва
2019

1. **Тема**: Основы метапрограммирования в C++.
2. **Код программы:**

vertex.h

```
#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;
   return os;
}

#endif // D_VERTEX_H_
```

templates.h

```
#ifndef D_TEMPLATES_H_
#define D_TEMPLATES_H_ 1

#include <tuple>
#include <type_traits>

#include "vertex.h"

//basic
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;

//center
```

```
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> sngl_center(const T& t) {
   vertex<double> v;
   v.x = std::get<ID>(t).x;
   v.y = std::get<ID>(t).y;
   v.x = v.x / std::tuple_size_v<T>;
   v.y = v.y / std::tuple_size_v<T>;
   return v;
}

template<size_t ID, class T>
vertex<double> rcrsv_center(const T& t) {
   if constexpr (ID < std::tuple_size_v<T>){
      return  {sngl_center<ID>(t).x + rcrsv_center<ID+1>(t).x, sngl_center<ID>(t).y + rcrsv_center<ID+1>(t).y};
   } 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& tuple) {
   return rcrsv_center<0>(tuple);
}

//area
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 sngl_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 rcrsv_area(const T& t) {
   if constexpr (ID < std::tuple_size_v<T>){
      return sngl_area<ID>(t) + rcrsv_area<ID + 1>(t);
   }
   return 0;
}

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

//print
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::cout))>> :
      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,std::ostream& os) {
      return figure.print(os);
}

template<size_t ID, class T>
void sngl_print(const T& t) {
   std::cout << "[" << std::get<ID>(t) << "]";
   if (ID < 3){
      std::cout << " ";
   }
   return ;
}

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

```
    }
    return;
}

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

#endif // D_TEMPLATES_H_
```

square.h

```
#ifndef D_SQUARE_H_
#define D_SQUARE_H_ 1

#include <algorithm>
#include <iostream>
#include <cmath>
#include <cassert>

#include "vertex.h"

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

    square(std::istream& is);

    vertex<double> center() const;
    double area() const;
    void print(std::ostream& os) const;

};

template<class T>
square<T>::square(std::istream& is) {
    for(int i = 0; i < 4; ++i){
        is >> vertices[i];
    }
    assert(((vertices[1].x - vertices[0].x)*(vertices[3].x - vertices[0].x))+((vertices[1].y - vertices[0].y)*(vertices[3].y
- vertices[0].y)) == 0);
    assert(((vertices[2].x - vertices[1].x)*(vertices[2].x - vertices[3].x))+((vertices[2].y - vertices[1].y)*(vertices[2].y
- vertices[3].y)) == 0);
    assert(((vertices[3].x - vertices[2].x)*(vertices[1].x - vertices[2].x))+((vertices[3].y - vertices[2].y)*(vertices[1].y
- vertices[2].y)) == 0);
    assert((vertices[1].x - vertices[0].x) == (vertices[0].y - vertices[3].y));
    assert((vertices[2].x - vertices[1].x) == (vertices[1].y - vertices[0].y));
    assert((vertices[3].x - vertices[2].x) == (vertices[2].y - vertices[1].y));
}

template<class T>
vertex<double> square<T>::center() const {
    return {(vertices[0].x + vertices[1].x + vertices[2].x + vertices[3].x) * 0.25, (vertices[0].y + vertices[1].y +
vertices[2].y + vertices[3].y) * 0.25};
}

template<class T>
double square<T>::area() const {
    const T d1 = vertices[0].x - vertices[1].x;
    const T d2 = vertices[3].x - vertices[0].x;
    return abs(d1 * d1) + abs(d2 * d2);
}
```

```
template<class T>
void square<T>::print(std::ostream& os) const {
    os << "Square ";
   for(int i = 0; i < 4; ++i){
      os << "[" << vertices[i] << "]";
      if(i + 1 != 4){
      os << " ";
      }
   }
   os << '\n';
}

#endif // D_SQUARE_H_
```

rectangle.h

```
#ifndef D_RECTANGLE_H
#define D_RECTANGLE_H

#include <algorithm>
#include <iostream>
#include <cmath>
#include <cassert>

#include "vertex.h"

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

   rectangle(std::istream& is);

   vertex<double> center() const;
   double area() const;
   void print(std::ostream& os) const;

};

template<class T>
rectangle<T>::rectangle(std::istream& is) {
   for(int i = 0; i < 4; ++i){
      is >> vertices[i];
   }
   assert(((vertices[1].x - vertices[0].x)*(vertices[3].x - vertices[0].x))+((vertices[1].y - vertices[0].y)*(vertices[3].y
- vertices[0].y)) == 0);
   assert(((vertices[2].x - vertices[1].x)*(vertices[2].x - vertices[3].x))+((vertices[2].y - vertices[1].y)*(vertices[2].y
- vertices[3].y)) == 0);
   assert(((vertices[3].x - vertices[2].x)*(vertices[1].x - vertices[2].x))+((vertices[3].y - vertices[2].y)*(vertices[1].y
- vertices[2].y)) == 0);
}

template<class T>
vertex<double> rectangle<T>::center() const {
   return {(vertices[0].x + vertices[1].x + vertices[2].x + vertices[3].x) * 0.25, (vertices[0].y + vertices[1].y +
vertices[2].y + vertices[3].y) * 0.25};
}

template<class T>
double rectangle<T>::area() const {
   const T dx1 = vertices[1].x - vertices[0].x;
   const T dy1 = vertices[0].y - vertices[3].y;
   const T dx2 = vertices[0].x - vertices[3].x;
   const T dy2 = vertices[1].y - vertices[0].y;
```

```
    return abs(dx1 * dy1) + abs(dx2 * dy2);
}

template<class T>
void rectangle<T>::print(std::ostream& os) const {
     os << "Rectangle ";
    for(int i = 0; i < 4; ++i){
        os << "[" << vertices[i] << "]";
        if(i + 1 != 4){
        os << " ";
        }
    }
    os << '\n';
}


#endif // D_RECTANGLE_H_
```

trapeze.h

```
#ifndef D_TRAPEZE_H
#define D_TRAPEZE_H

#include <algorithm>
#include <iostream>
#include <cmath>
#include <cassert>

#include "vertex.h"

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

  trapeze(std::istream& is);

  vertex<double> center() const;
  double area() const;
  void print(std::ostream& os) const;

};

template<class T>
trapeze<T>::trapeze(std::istream& is) {
  for(int i = 0; i < 4; ++i){
    is >> vertices[i];
  }
  assert(((vertices[1].x - vertices[0].x)*(vertices[3].y - vertices[2].y)) == ((vertices[3].x - vertices[2].x)*(vertices[1].y
- vertices[0].y)));
}

template<class T>
vertex<double> trapeze<T>::center() const {
  return {(vertices[0].x + vertices[1].x + vertices[2].x + vertices[3].x) * 0.25, (vertices[0].y + vertices[1].y +
vertices[2].y + vertices[3].y) * 0.25};
}

template<class T>
double trapeze<T>::area() const {
  const double l11 = vertices[1].x - vertices[0].x;
  const double l12 = vertices[1].y - vertices[0].y;
  const double l21 = vertices[3].x - vertices[2].x;
  const double l22 = vertices[3].y - vertices[2].y;
```

```
  const double lh = ((vertices[3].x - vertices[0].x)*(vertices[3].x - vertices[2].x)+(vertices[3].y -
vertices[0].y)*(vertices[3].y - vertices[2].y))/sqrt((vertices[3].x - vertices[2].x)*(vertices[3].x - vertices[2].x)+
(vertices[3].y - vertices[2].y)*(vertices[3].y - vertices[2].y));
  const double h = sqrt((vertices[3].x - vertices[0].x)*(vertices[3].x - vertices[0].x)+(vertices[3].y -
vertices[0].y)*(vertices[3].y - vertices[0].y)-lh*lh);
  return ((abs(l11) + abs(l12) + abs(l21) + abs(l22)) * abs(h) * 0.5);
}

template<class T>
void trapeze<T>::print(std::ostream& os) const {
  os << "Trapezoid ";
  for(int i = 0; i < 4; ++i){
    os << "[" << vertices[i] << "]";
    if(i + 1 != 4){
      os << " ";
    }
  }
  os << '\n';
}

#endif // D_TRAPEZE_H_
```

lab4.cpp

```
#include <iostream>
#include <tuple>

#include "vertex.h"
#include "square.h"
#include "rectangle.h"
#include "trapeze.h"

#include "templates.h"

enum Commands{
    cmd_quit,
    cmd_sqr,
    cmd_rect,
    cmd_trpz,
    cmd_tpl
};

template<class T>
void process() {
    T object(std::cin);
    //void read(std::cin, object);
    print(object, std::cout);
    std::cout << "Center: [" << center(object) << "]" << std::endl;
    std::cout << "Area: " << area(object) << std::endl;
}


int main(){
    for (;;){
        int command;
        std::cin >> command;
        switch (command){
            case cmd_quit:
                exit(0);
            case cmd_sqr:
                process<square<int>>();
                break;
            case cmd_rect:
                process<rectangle<int>>();
```

```
            break;
        case cmd_trpz:
            process<trapeze<int>>();
            break;
        case cmd_tpl:
            vertex<int> vrtx[4];
            for(int i = 0; i < 4; ++i){
                std::cin >> vrtx[i];
            }
            std::tuple<vertex<int>, vertex<int>, vertex<int>, vertex<int>>
                f_tuple{{vrtx[0].x, vrtx[0].y}, {vrtx[1].x, vrtx[1].y}, {vrtx[2].x, vrtx[2].y}, {vrtx[3].x, vrtx[3].y}};
            if ((((vrtx[1].x - vrtx[0].x)*(vrtx[3].x - vrtx[0].x))+((vrtx[1].y - vrtx[0].y)*(vrtx[3].y - vrtx[0].y)) == 0)
&& (((vrtx[2].x - vrtx[1].x)*(vrtx[2].x - vrtx[3].x))+((vrtx[2].y - vrtx[1].y)*(vrtx[2].y - vrtx[3].y)) == 0) &&
(((vrtx[3].x - vrtx[2].x)*(vrtx[1].x - vrtx[2].x))+((vrtx[3].y - vrtx[2].y)*(vrtx[1].y - vrtx[2].y)) == 0)) {
                if (((vrtx[1].x - vrtx[0].x) == (vrtx[0].y - vrtx[3].y)) && ((vrtx[2].x - vrtx[1].x) == (vrtx[1].y -
vrtx[0].y)) && ((vrtx[3].x - vrtx[2].x) == (vrtx[2].y - vrtx[1].y))){
                    std::cout << "Tuple (As Square) ";
                } else {
                    std::cout << "Tuple (As Rectangle) ";
                }
            } else if (((vrtx[1].x - vrtx[0].x)*(vrtx[3].y - vrtx[2].y)) == ((vrtx[3].x - vrtx[2].x)*(vrtx[1].y - vrtx[0].y)))
{
                std::cout << "Tuple (As Trapezoid) ";
            } else {
                std::cout << "Tuple ";
            }
            print(f_tuple);
            std::cout << std::endl;
            std::cout << "Center: [" << center(f_tuple) << "]" << std::endl;
            std::cout << "Area: " << area(f_tuple) << std::endl;
    }
  }
}
```

CMakeLists.txt

```
project(lab4)

set(CMAKE_CXX_STANDARD 17)

add_executable(lab4
  ./lab4.cpp)

set(CMAKE_CXX_FLAGS
  "${CMAKE_CXX_FLAGS} -Wall -Wextra")
  "${CMAKE_CXX_FLAGS} -Wall -Wextra")
```

3. **Ссылка на репозиторий:**
   https://github.com/GitGood2000/oop_exercise_04

4. **Набор testcases:**

test_00.test

```
1
0 2 2 2 2 0 0 0
2
1 7 4 7 4 2 1 2
3
2 4 6 4 7 1 1 1
0
```

test_00.result

Square [0 2] [2 2] [2 0] [0 0]
Center: [1 1]
Area: 4
Rectangle [1 7] [4 7] [4 2] [1 2]
Center: [2.5 4.5]
Area: 15
Trapezoid [2 4] [6 4] [7 1] [1 1]
Center: [4 2.5]
Area: 15

test_01.test
1
0 2 2 3 3 1 1 0
1
1 3 4 6 7 3 4 0
2
0 2 4 4 5 2 1 0
2
1 3 7 5 8 2 2 0
3
0 3 3 5 6 5 0 1
3
1 5 7 3 5 1 2 2
0

test_01.result
Square [0 2] [2 3] [3 1] [1 0]
Center: [1.5 1.5]
Area: 5
Square [1 3] [4 6] [7 3] [4 0]
Center: [4 3]
Area: 18
Rectangle [0 2] [4 4] [5 2] [1 0]
Center: [2.5 2]
Area: 10
Rectangle [1 3] [7 5] [8 2] [2 0]
Center: [4.5 2.5]
Area: 20
Trapezoid [0 3] [3 5] [6 5] [0 1]
Center: [2.25 3.5]
Area: 7.5
Trapezoid [1 5] [7 3] [5 1] [2 2]
Center: [3.75 2.75]
Area: 12

test_02.test
1
-1 1 0 2 1 1 0 0
2
-1 1 1 3 2 2 0 0
3
-1 1 0 2 2 2 0 0
4
-1 1 0 2 1 1 0 0
4
-1 1 1 3 2 2 0 0
4
-1 1 0 2 2 2 0 0
0

test_02.result
Square [-1 1] [0 2] [1 1] [0 0]
Center: [0 1]

Area: 2
Rectangle [-1 1] [1 3] [2 2] [0 0]
Center: [0.5 1.5]
Area: 4
Trapezoid [-1 1] [0 2] [2 2] [0 0]
Center: [0.25 1.25]
Area: 3
Tuple (As Square) [-1 1] [0 2] [1 1] [0 0]
Center: [0 1]
Area: 2
Tuple (As Rectangle) [-1 1] [1 3] [2 2] [0 0]
Center: [0.5 1.5]
Area: 4
Tuple (As Trapezoid) [-1 1] [0 2] [2 2] [0 0]
Center: [0.25 1.25]
Area: 3
Tuple [-1 1] [1 4] [5 8] [6 2]
Center: [2.75 3.75]
Area: 23.5

5. **Результаты выполнения тестов:**

user@PSB133S01ZFH:~/3sem_projects/oop_exercise_04/tests$ bash test.sh ../build/lab4
Test test_00.test: SUCCESS
Test test_01.test: SUCCESS
Test test_02.test: SUCCESS

6. **Объяснение результатов работы программы:**
   1) Программа выполняет определённые действия по введённым командам:
      A) 0 — выход из программы;
      B) 1,2,3 — создание фигуры(Квадрат, Прямоугольник, Трапеция соответственно), получение вершин через ввод, проверка, вывод данных вершин, вычисление центра и площади;
      C) 4 — создание кортежа как производного четырёхугольника, получение вершин через ввод, проверка, вывод данных вершин, вычисление центра и площади;
   2) Шаблонная функция print() печатает координаты всех точек данной фигуры или кортежа. Она определена для моих фигур и tuple. Во втором случае все дело вычисляется рекурсивно.
   3) Функция center() возвращает точку с x –деление суммы иксов всех точек данной фигуры на их количество, y – аналогично x. Она определена для моих фигур и tuple. Во втором случае все дело вычисляется рекурсивно;
   4) Функция area() вычисляет площадь данной фигуры или совокупности точек в кортеже в зависимости от типа фигуры по методу Гаусса (формула землемера, метод шунтирования) и возвращает это значение.

7. **Вывод:** 1) Ознакомились с шаблонами и кортежами в C++ и усвоили навык работы с ними;
   2) Написана программа, производящая операции с помощью шаблонов и работающая с кортежами.