МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСОЙ ФЕДЕРАЦИИ МОСКОВСКИЙ АВЦИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЬЕЛЬСКИЙ УНИВЕРСТИТЕТ)

ЛАБОРАТОРНАЯ РАБОТА №4 **по курсу**

объектно-ориентированное программирование I семестр, 2019/20 уч. год

Студент *Артемьев Дмитрий Иванович, группа М8О-206Б-18*

Преподаватель *Журавлёв Андрей Андреевич*

Условие

Задание: Вариант 1: Треугольник, Квадрат, Прямоугольник. Разработать шаблоны классов согласно варианту задания. Параметром шаблона должен являться скалярный тип данных, задающий тип данных для оси координат. Классы должны иметь публичные поля.

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

Создать программу, которая позволяет:

1. Вводить из стандартного ввода std::cin фигуры, согласно варианту задания (как в виде класса, так и в виде std::tuple).
2. Вызывать для неё шаблонные функции(1-3).

Описание программы

Исходный код лежит в 11 файлах:

1. src/main.cpp: основная программа, взаимодействие с пользователем посредством комманд из меню
2. include/figure.hpp: описание шаблона класса фигур
3. include/point.hpp: описание шаблона класса точки
4. include/triangle.hpp: описание шаблона класса треугольника, наследующегося от figures
5. include/rectangle.hpp: описание шаблона класса прямоугольника, наследующегося от figures
6. include/square.hpp: описание шаблона класса квадрата, наследующегося от rectangle
7. include/template.hpp: функции для чтения/вывода tuple
8. include/functions.hpp: функции для нахождения площади, центра фигуры и вывода её координат на экран

Дневник отладки

Долгие попытки прикрутить фичи C++17.

Недочёты

Нечитабельный код.

Выводы

Научился использовать шаблоны в C++, изучил коллекцию tuple и некоторые фишки C++17.

Исходный код

main.cpp

```cpp
#include <iostream>
#include <string>
#include <tuple>

#include "point.hpp"
#include "triangle.hpp"
#include "square.hpp"
#include "rectangle.hpp"

#include "template.hpp"
#include "functions.hpp"


template<class T>
void process(std::ostream& out, T& obj) {
    print(out, obj);
    out << area(obj) << std::endl;
    out << center(obj) << std::endl;
}

void processTuple(std::istream& in, std::ostream& out) {
    std::string objType;
    in >> objType;

    using pii = Point<int>;
    using triangle  = std::tuple<pii, pii, pii>;
    using square    = std::tuple<pii, pii, pii, pii>;
    using rectangle = std::tuple<pii, pii, pii, pii>;

    if (objType == "triangle") {
        triangle obj;
        read(in, obj);
        process<triangle>(out, obj);
    }
    else if (objType == "square") {
        square obj;
        read(in, obj);
        process<square>(out, obj);
    }
```

```
    else if (objType == "rectangle") {
        rectangle obj;
        read(in, obj);
        process<rectangle>(out, obj);
    }
    else {
        out << "Wrong object type!" << std::endl;
        return;
    }
}

void processFigure(std::istream& in, std::ostream& out) {
    std::string objType;
    in >> objType;

    using triangle  = Triangle<int>;
    //using square    = Square<int>;
    //using rectangle = Rectangle<int>;
    if (objType == "triangle") {
        triangle obj(in);
        process<triangle>(out, obj);
    }
    // else if (objType == "square") {
    //     Square<int> obj(in);
    //     process<Square<int>>(obj);
    // }
    // else if (objType == "rectangle") {
    //     Rectangle<int> obj(in);
    //     process<Rectangle<int>>(obj);
    //}
    else {
        out << "Wrong object type!" << std::endl;
        return;
    }
}

int main() {

    while (std::cin) {
        std::cout << "Enter form(tuple/figure), ";
        std::cout << "figure type(triangle, square, rectangle), " << std::endl;
```

```
        std::string objForm;
        std::cin >> objForm;

        if (objForm == "tuple") {
            processTuple(std::cin, std::cout);
        }
        else if (objForm == "figure") {
            processFigure(std::cin, std::cout);
        }
        else {
            std::cout << "Wrong object form!" << std::endl;
        }
    }
}
```

figure.hpp

```cpp
#pragma once

#include <iostream>

#include "template.hpp"
#include "point.hpp"

template<typename T, size_t S>
class Figure {
public:
    Figure (std::istream& in) {
        for (int i = 0; i < S; i++) {
            in >> pts[i];
        }
    }
    double area() const;
    Point<double> center() const;
    virtual void print(std::ostream& out) const = 0;

    Point<T>& operator[] (size_t id) {
        return pts[id];
    }
    const Point<T>& operator[] (size_t id) const {
        return pts[id];
    }

    virtual ~Figure() {};
protected:
    Point<T> pts[S];
};

template<typename T, size_t S>
double Figure<T, S>::area() const {
    double ans = 0;
    for (int i = 1; i < S; i++) {
        ans += crossProd(pts[0], pts[i], pts[i-1]);
    }
    return std::abs(ans) / 2.0;
}
```

```cpp
template<typename T, size_t S>
Point<double> Figure<T, S>::center() const {
    Point<double> ans{0, 0};
    for (int i = 1; i < S; i++) {
        ans = ans + pts[i];
    }
    return ans / (double)(S);
}
```

point.hpp

```cpp
#pragma once

#include <iostream>

template<class T>
struct Point {
    T x, y;

    Point(T x, T y) : x(x), y(y) {};
    Point() = default;
};

template<class T>
std::istream& operator>> (std::istream& in, Point<T>& pt) {
    in >> pt.x >> pt.y;
    return in;
}

template<class T>
std::ostream& operator<< (std::ostream& out, Point<T> const& pt) {
    out << " [" << pt.x << ' ' << pt.y << "]";
    return out;
}

template<class T, class R>
Point<T> operator+ (Point<T> const& a, Point<R> const& b) {
    return Point<T>{a.x + b.x, a.y + b.y};
}

template<class T, class R>
Point<T> operator- (Point<T> const& a, Point<R> const& b) {
    return Point<T>{a.x - b.x, a.y - b.y};
}

template<class T, class R>
Point<double> operator* (Point<T> const& a, R const& b) {
    return Point<double>{a.x * b, a.y * b};
}

template<class T, class R>
Point<double> operator/ (Point<T> const& a, R const& b) {
```

```
    return Point<double>{double(a.x) / b, double(a.y) / b};
}
```

triangle.hpp

```cpp
#pragma once

#include <iostream>

#include "point.hpp"
#include "figure.hpp"

template<class T>
class Triangle : public Figure<T, 3> {
public:
    Triangle() = default;
    Triangle(std::istream& in) : Figure<T, 3>(in) {};

    void print(std::ostream& out) const override {
        out << "Triangle: " << std::endl;
        for (int i = 0; i < 3; i++) {
            out << Figure<T, 3>::pts[i] << std::endl;
        }
    }
    ~Triangle() {};
};
```

rectangle.hpp

```cpp
#pragma once

#include <iostream>
#include <algorithm>

#include "point.hpp"
#include "template.hpp"

template<typename T>
class Rectangle : Figure<T, 4> {
public:
    Rectangle() = default;
    Rectangle(std::istream& in) : Figure<T, 4>(in) {
        int a = dist(Figure<T, 4>::pts[0], Figure<T, 4>::pts[1]);
        bool is_Rectangle = (dist(Figure<T, 4>::pts[0], Figure<T, 4>::pts[1]) == dist(Fi
                             dist(Figure<T, 4>::pts[0], Figure<T, 4>::pts[3]) == dist(Fi
        for (int i = 0; i < 4; i++) {
            if (scalarProd(Figure<T, 4>::pts[i], Figure<T, 4>::pts[(i + 1) % 4], Figure<
                is_Rectangle = false;
            }
        }
        if (is_Rectangle == false) {
            std::cout << "GO FUCK" << std::endl;
            ~Rectangle();
        }
    };

    void print(std::ostream& out) const override {
        out << "Rectangle: " << std::endl;
        for (int i = 0; i < 4; i++) {
            out << Figure<T, 4>::pts[i] << std::endl;
        }
    }
    ~Rectangle() {};
};
```

square.hpp

```cpp
#pragma once

#include <iostream>
#include <algorithm>

#include "point.hpp"
#include "template.hpp"
#include "figure.hpp"

template<class T>
class Square : Figure<T, 4> {
public:
    Square() = default;
    Square(std::istream& in) : Figure<T, 4>(in) {
        int a = dist(Figure<T, 4>::pts[0], Figure<T, 4>::pts[1]);
        bool is_Square = true;
        for (int i = 0; i < 4; i++) {
            if (dist(Figure<T, 4>::pts[i], Figure<T, 4>::pts[(i + 1) % 4]) != a ||
                scalarProd(Figure<T, 4>::pts[i], Figure<T, 4>::pts[(i + 1) % 4], Figure<
                is_Square = false;
            }
        }
        if (is_Square == false) {
            std::cout << "GO FUCK" << std::endl;
            ~Square();
        }
    };

    void print(std::ostream& out) const override {
        out << "Square: " << std::endl;
        for (int i = 0; i < 4; i++) {
            out << Figure<T, 4>::pts[i] << std::endl;
        }
    }
    ~Square() {};
};
```

template.hpp

```cpp
#pragma once

#include <iostream>
#include <tuple>
#include <cmath>

// read tuple
template<size_t I, class... Ts>
struct reader {
    static void read(std::istream& i, std::tuple<Ts...>& t) {
        i >> std::get<I>(t);
        reader<I - 1, Ts...>::read(i, t);
    }
};

template<class... Ts>
struct reader<0, Ts...> {
    static void read(std::istream& i, std::tuple<Ts...>& t) {
        i >> std::get<0>(t);
    }
};

template<class... Ts>
void read(std::istream& i, std::tuple<Ts...>& t) {
    reader<sizeof...(Ts) - 1, Ts...>::read(i, t);
}
// end read tuple

// point functions
template<class T>
T crossProd(Point<T> b, Point<T> a, Point<T> c) {
    return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
}

template<class T>
T scalarProd(Point<T> b, Point<T> a, Point<T> c) {
    return (b.x - a.x) * (c.x - a.x) + (b.y - a.y) * (c.x - a.x);
}

template<class T>
T dist(Point<T> a, Point<T> b) {
```

```
    return sqrt((b.x - a.x) * (b.x - a.x) + (b.y - a.y) * (b.y - a.y));
}
// end point functions
```

functions.hpp

```cpp
#pragma once

#include <iostream>
#include <tuple>
#include <type_traits>

#include "point.hpp"
#include "template.hpp"

template<class T>
struct is_point : std::false_type {};

template<class T>
struct is_point<Point<T>> : std::true_type {};

template<class T>
struct is_figure_like_tuple : std::false_type {};

template<class Head, class... Tail>
struct is_figure_like_tuple<std::tuple<Head, Tail...>> :
std::conjunction<is_point<Head>,
        std::is_same<Head, Tail>...> {};

template<class T>
inline constexpr bool is_figure_like_tuple_v =
        is_figure_like_tuple<T>::value;

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>,
```

```
                  decltype(std::declval<const T>().area())>
area(const T& figure) {
    return figure.area();
}


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>,
                  decltype(std::declval<const T>().print(std::cout))>
print (std::ostream& os, const T& figure) {
    return figure.print(os);
}

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>,
                  decltype(std::declval<T>().center()) >
center (const T& figure) {
    return figure.center();
```

```
}


// print figure

template<class T>
bool is_triangle(const std::tuple<T,T,T>& tup) {
    return true;
}

template<class T>
bool is_triangle(const T& tup) {
    return true;
}


template<class T>
bool is_square(const std::tuple<T,T,T,T>& tup) {
    auto [a, b, c, d] = tup;
    if (scalarProd(a, b, c) == 0 && dist(a, b) == dist(b, a) &&
        scalarProd(b, c, d) == 0 && dist(b, c) == dist(c, d) &&
        scalarProd(c, d, a) == 0 && dist(c, d) == dist(d, a) &&
        scalarProd(d, a, b) == 0 && dist(d, a) == dist(a, b))
        return true;
    return false;
}

template<class T>
bool is_square(const T& tup) {
    return false;
}


template<class T>
bool is_rectangle(const std::tuple<T,T,T,T>& tup) {
    auto [a, b, c, d] = tup;
    if (scalarProd(a, b, c) == 0 && scalarProd(b, c, d) == 0 &&
        scalarProd(c, d, a) == 0 && scalarProd(d, a, b) == 0 &&
        dist(a, b) == dist(c, d) && dist(b, c) == dist(a, d))
        return true;
    return false;
}
```

```cpp
template<class T>
bool is_rectangle(const T& tup) {
    return false;
}

template<class T, std::size_t... N>
auto recursive_print(std::ostream& out, const T& tup, std::index_sequence<N...>) {
    (out << ... << std::get<N>(tup));
}

template<class T>
std::enable_if_t<is_figure_like_tuple_v<T>, void>
print(std::ostream& out, const T& tup) {
    auto constexpr tuple_size = std::tuple_size<T>::value;


    if (tuple_size == 3 && is_triangle(tup)) {
        out << "Triangle: " << std::endl;
    }
    else if (tuple_size == 4 && is_square(tup)) {
        out << "Square: " << std::endl;
    }
    else if (tuple_size == 4 && is_rectangle(tup)) {
        out << "Rectangle: " << std::endl;
    }
    else {
        out << "Strange figure: " << std::endl;
    }

    recursive_print(out, tup, std::make_index_sequence<tuple_size>{});
    std::cout << std::endl;
}
// end print figure

// center figure
template<class T, size_t N>
Point<int> recursive_center(const T& tup) {
    if constexpr (N >= std::tuple_size_v<T>) {
            return Point<int>{0,0};
        }
    else {
```

```
        return std::get<N>(tup) + recursive_center<T, N+1>(tup) ;
    }
}

template<class T>
std::enable_if_t<is_figure_like_tuple_v<T>, Point<double>>
center(const T& tup) {
    auto constexpr tuple_size = std::tuple_size<T>{}();

    Point<int> center = recursive_center<T, 0>(tup);
    return center / tuple_size;
}
// end center figure

// shift index sequence
template<size_t N1, size_t ...N2>
std::index_sequence<(N1 + N2)...> shift_sequence(std::index_sequence<N2...>) {
    return {};
}

template<size_t N1, size_t N2>
auto shifted_index_sequence() {
    return shift_sequence<N2>(std::make_index_sequence<N1>{});
}
// end shift index sequence

// area figure
template<typename _T, size_t... _Ix>
double area(const _T& tuple, std::index_sequence<_Ix...>) {
    auto constexpr tuple_size = std::tuple_size<_T>{}();

    using std::get;

    double result = ((get<_Ix>(tuple).x * (get<_Ix + 1>(tuple).y - get<_Ix - 1>(tuple).y
    auto constexpr first = 0;
    auto constexpr last = tuple_size - 1;
    result += get<first>(tuple).x * (get<first + 1>(tuple).y - get<last>(tuple).y);
    result += get<last>(tuple).x * (get<first>(tuple).y - get<last - 1>(tuple).y);
    result /= 2;

    return std::abs(result);
}
```

```cpp
template<typename T>
std::enable_if_t<is_figure_like_tuple_v<T>, double>
 area(const T& tuple) {
    auto constexpr tuple_size = std::tuple_size<T>{}();
    return area(tuple, shifted_index_sequence<1, tuple_size - 2>());
}
// end area figure
```