Московский Авиационный Институт
(Национальный исследовательский Университет)

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

Лабораторная работа
по курсу «ООП»

Тема:
Основы метапрограммирования.

Студент:	Петрин С.А.
Группа:	М80-207Б-18
Преподаватель:	Чернышов Л.Н.
Вариант:	17
Оценка:	
Дата:	

Москва
2019

1. Код программы на языке C++:

main.cpp:

```
#include <iostream>
#include <cstring>
#include <exception>
#include "rectangle.h"
#include "square.h"
#include "triangle.h"
#include "vector.h"
#include "templates.h"

template<typename T>
using vertex_t = std::pair<T, T>;

template<typename T>
void process() {
	T object;
	std::cout << "Input points: ";
	try {
		Read(std::cin, object);
	}
	catch (std::exception &e) {
		std::cout << e.what() << std::endl;
		return;
	}
	Print(std::cout, object);
	std::cout << std::endl<< Area(object) << std::endl;
	std::cout << Center(object) << std::endl;
}

int main() {
	std::string command;
	std::cout << "Figure / Tuple" << std::endl;
	std::cout << "_____" << std::endl;
	while (std::cin >> command) {
		std::string object_type;
		if (command == "Figure") {
			std::cout << "_____" << std::endl;
			std::cout << "Triangle / Square / Rectangle" << std::endl;
			std::cout << "_____" << std::endl;
			std::cin >> object_type;
			std::cout << "_____" << std::endl;
			if (object_type == "Triangle") {
				process<Triangle<double>>();
			}
			else if (object_type == "Square") {
				process<Square<double>>();
			}
			else if (object_type == "Rectangle") {
				process<Rectangle<double>>();
			}
			else {
```

```
                std::cout << "Invalid figure" << std::endl;
                std::cin.clear();
                std::cin.ignore(30000, '\n');
                std::cout << "_____" << std::endl;
                std::cout << "Figure / Tuple" << std::endl;
                std::cout << "_____" << std::endl;
                continue;
            }
        }
        else if (command == "Tuple") {
            std::cout << "_____" << std::endl;
            std::cout << "Triangle / Square / Rectangle" << std::endl;
            std::cout << "_____" << std::endl;
            std::cin >> object_type;
            std::cout << "_____" << std::endl;
            if (object_type == "Triangle") {
                process<std::tuple<vertex_t<double>, vertex_t<double>,
                                   vertex_t<double>>>();
            }
            else if (object_type == "Square") {
                process<std::tuple<vertex_t<double>, vertex_t<double>,
                                   vertex_t<double>, vertex_t<double>>>();
            }
            else if (object_type == "Rectangle") {
                process<std::tuple<vertex_t<double>, vertex_t<double>,
                                   vertex_t<double>, vertex_t<double>>>();
            }
            else {
                std::cout << "Invalid figure" << std::endl;
                std::cin.clear();
                std::cin.ignore(30000, '\n');
                std::cout << "_____" << std::endl;
                std::cout << "Figure / Tuple" << std::endl;
                std::cout << "_____" << std::endl;
                continue;
            }
        }
        else {
            std::cout << "Invalid command" << std::endl;
            std::cin.clear();
            std::cin.ignore(30000, '\n');
            std::cout << "_____" << std::endl;
            continue;
        }
        std::cout << "_____" << std::endl;
        std::cout << "Figure / Tuple" << std::endl;
        std::cout << "_____" << std::endl;
    }

    return 0;
}
```

templates.h:

```
#ifndef D_TEMPLATES_H
#define D_TEMPLATES_H 1

#include <tuple>
#include <utility>
#include <type_traits>
#include <exception>


#include "vector.h"

template<typename T>
using vertex_t = std::pair<T, T>;

template<typename T>
struct is_vertex : std::false_type {};

template<typename T>
struct is_vertex<std::pair<T, 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<typename T>
inline constexpr bool is_figurelike_tuple_v = is_figurelike_tuple<T>::value;

template<typename T, typename = void>
struct has_method_area : std::false_type {};

template<typename T>
struct has_method_area<T, decltype(std::declval<const T&>().Area())> :
      std::true_type {};

template<typename T>
inline constexpr bool has_method_area_v = has_method_area<T>::value;

template<typename T>
std::enable_if_t<has_method_area_v<T>, double>
Area(const T& object) {
      return object.Area();
}

template<typename T, typename = void>
struct has_method_center : std::false_type {};

template<typename T>
```

```
struct has_method_center<T, decltype(std::declval<const T&>().Center())> :
    std::true_type {};

template<typename T>
inline constexpr bool has_method_center_v = has_method_center<T>::value;

template<typename T>
std::enable_if_t<has_method_center_v<T>, vertex_t<double>>
Center(const T& object) {
    return object.Center();
}

template<typename T, typename = void>
struct has_method_print : std::false_type {};

template<typename T>
struct has_method_print<T, decltype(std::declval<const T&>().Print())> :
    std::true_type {};

template<typename T>
inline constexpr bool has_method_print_v = has_method_print<T>::value;

template<typename T>
std::enable_if_t<has_method_print_v<T>, std::ostream &>
Print(std::ostream &os, const T& object) {
    return Print(os, object);
}

template<typename T, typename = void>
struct has_method_read : std::false_type {};

template<typename T>
struct has_method_read<T, decltype(std::declval<const T&>().Read())> :
    std::true_type {};

template<typename T>
inline constexpr bool has_method_read_v = has_method_read<T>::value;

template<typename T>
std::enable_if_t<has_method_read_v<T>, std::istream &>
Read(std::istream &is, T& object) {
    return Read(is, object);
}

template<size_t Id, typename T>
double compute_area(const T &tuple) {
    if constexpr(Id >= std::tuple_size_v<T>) {
        return 0;
    }
    else {
        const auto dx1 = std::get<Id - 0>(tuple).first - std::get<0>(tuple).first;
        const auto dy1 = std::get<Id - 0>(tuple).second - std::get<0>(tuple).second;
```

```
        const auto dx2 = std::get<Id - 1>(tuple).first - std::get<0>(tuple).first;
        const auto dy2 = std::get<Id - 1>(tuple).second - std::get<0>(tuple).second;
        const double local_area = std::abs(dx1 * dy2 - dy1 * dx2) * 0.5;
        return local_area + compute_area<Id + 1>(tuple);
    }
}

template<typename T>
std::enable_if_t<is_figurelike_tuple_v<T>, double>
Area(const T& object) {
    if constexpr (std::tuple_size_v<T> < 3){
        return 0;
    }
    else{
        return compute_area<2>(object);
    }
}

template<size_t Id, typename T>
double recursive_center_x(const T &tuple) {
    if constexpr (Id >= std::tuple_size_v<T>) {
        return 0;
    }
    else {
        return (std::get<Id>(tuple).first / std::tuple_size_v<T>) + recursive_center_x<Id +
1>(tuple);
    }
}

template<size_t Id, typename T>
double recursive_center_y(const T &tuple) {
    if constexpr (Id >= std::tuple_size_v<T>) {
        return 0;
    }
    else {
        return (std::get<Id>(tuple).second / std::tuple_size_v<T>) + recursive_center_y<Id +
1>(tuple);
    }
}


template<size_t Id, typename T>
vertex_t<double> compute_center(const T &tuple) {
    if constexpr (Id >= std::tuple_size_v<T>) {
        return 0;
    }
    else {
        return {recursive_center_x<Id>(tuple), recursive_center_y<Id>(tuple)} ;
    }
}

template<typename T>
```

```
std::enable_if_t<is_figurelike_tuple_v<T>, vertex_t<double>>
Center(const T& object) {
    return compute_center<0>(object);
}

template<size_t Id, typename T>
void recursive_print(std::ostream &os, const T &tuple) {
    if constexpr (Id >= std::tuple_size_v<T>) {
        return;
    }
    else {
        os << std::get<Id>(tuple) << " ";
        recursive_print<Id + 1>(os, tuple);
    }
}

template<typename T>
std::enable_if_t<is_figurelike_tuple_v<T>, void>
Print(std::ostream &os, const T& object) {
    recursive_print<0>(os, object);
}

template<typename T>
std::enable_if_t<is_figurelike_tuple_v<T>, void>
Check_triangle(T& object) {
    double AB = Length(std::get<0>(object), std::get<1>(object)),
           BC = Length(std::get<1>(object), std::get<2>(object)),
           AC = Length(std::get<0>(object), std::get<2>(object));
    if (AB >= BC + AC || BC >= AB + AC || AC >= AB + BC) {
        throw std::logic_error("Vertices must not be on the same line.");
    }
}

template<typename T>
std::enable_if_t<is_figurelike_tuple_v<T>, bool>
Check_rectangle(T& object) {
    Vector<decltype(std::get<0>(object).first)> AB = {std::get<0>(object), std::get<1>(object)},
           BC = {std::get<1>(object), std::get<2>(object)},
           CD = {std::get<2>(object), std::get<3>(object)},
           DA = {std::get<3>(object), std::get<0>(object)};
    if (!is_parallel(DA, BC) || !is_parallel(AB, CD)) {
        throw std::logic_error("Vertices must be entered clockwise or counterclockwise");
    }
    if (AB * BC || BC * CD || CD * DA || DA * AB) {
        throw std::logic_error("The sides should be perpendicular");
    }
    if (!Length(AB) || !Length(BC) || !Length(CD) || !Length(DA)) {
        throw std::logic_error("The sides must be greater than zero");
    }

    return true;
}
```

```
template<typename T>
std::enable_if_t<is_figurelike_tuple_v<T>, void>
Check(T& object) {
	if constexpr (std::tuple_size_v<T> == 3) {
		Check_triangle(object);
	}
	else if (std::tuple_size_v<T> == 4) {
		Check_rectangle(object);
	}
}

template<size_t Id, typename T>
void recursive_read(std::istream &is, T &tuple) {
	if constexpr (Id >= std::tuple_size_v<T>) {
		return;
	}
	else {
		is >> std::get<Id>(tuple);
		recursive_read<Id + 1>(is, tuple);
	}
}

template<typename T>
std::enable_if_t<is_figurelike_tuple_v<T>, void>
Read(std::istream &is, T& object) {
	recursive_read<0>(is, object);
	Check(object);
}


#endif // D_TEMPLATES_H
```

vertex.h:

```
#ifndef VERTEX_H
#define VERTEX_H 1

template<typename T>
struct vertex {
	using vertex_t = std::pair<T, T>;
};

template<typename T>
std::istream &operator>>(std::istream &is, std::pair<T, T> &v) {
	is >> v.first >> v.second;

	return is;
}

template<typename T>
std::ostream &operator<<(std::ostream &os, const std::pair<T,T> &v) {
	os << "[" << v.first << ", " << v.second << "]";
```

```
    return os;
}

#endif // VERTEX_H
```

vector.h:

```
#ifndef VECTOR_H
#define VECTOR_H 1

#include <utility>
#include <cmath>
#include <iostream>

#include "vertex.h"

template<typename T>
struct Vector {
    using vertex_t = std::pair<T, T>;
    T p1, p2;

    Vector(T x_cord, T y_cord) : p1{x_cord}, p2{y_cord} {};
    Vector(vertex_t &p1, vertex_t &p2) : p1{p2.first - p1.first},
                p2{p2.second - p1.second} {};
    double operator*(const Vector<T> &a) const {
        return (p1 * a.p1) + (p2 * a.p2);
    }
    Vector<T> &operator=(const Vector<T> &a) {
        p1 = a.p1;
        p2 = a.p2;
        return *this;
    }
};

template<typename T>
double Length(const Vector<T> &vector) {
    return sqrt(vector.p1 * vector.p1 + vector.p2 * vector.p2);
}

template<typename T>
double Length(const std::pair<T, T> &A,
              const std::pair<T, T> &B) {
    return sqrt(pow((B.first - A.first), 2) +
                pow((B.second - A.second), 2));
}

template<typename T>
bool is_parallel(const Vector<T> &A, const Vector<T> &B) {
    return (A.p1 * B.p2) - (A.p2 * B.p1) == 0;
}

#endif //VECTOR_H
```

rectangle.h:

```
#ifndef RECTANGLE_H
#define RECTANGLE_H 1

#include <utility>
#include <iostream>

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

template<typename T>
struct Rectangle {
	using vertex_t = std::pair<T,T>;
	vertex_t vertices[4];
};

template<typename T>
typename Rectangle<T>::vertex_t Center(const Rectangle<T> &r);

template<typename T>
double Area(const Rectangle<T> &r);

template<typename T>
std::ostream &Print(std::ostream &os, const Rectangle<T> &r);

template<typename T>
std::istream &Read(std::istream &is, Rectangle<T> &r);

template<typename T>
std::istream &operator>>(std::istream &is, Rectangle<T> &r);

template<typename T>
std::ostream &operator<<(std::ostream &os, const Rectangle<T> &r);

template<typename T>
typename Rectangle<T>::vertex_t Center(const Rectangle<T> &r) {
	T x, y;
	x = (r.vertices[0].first + r.vertices[1].first + r.vertices[2].first + r.vertices[3].first) / 4;
	y = (r.vertices[0].second + r.vertices[1].second + r.vertices[2].second + r.vertices[3].second) / 4;

	return std::make_pair(x, y);
}

template<typename T>
double Area(const Rectangle<T> &r) {
	double res = 0;
	for (int i = 0; i <= 2; i++) {
		res += (r.vertices[i].first * r.vertices[i + 1].second -
				r.vertices[i + 1].first * r.vertices[i].second);
	}
	res += (r.vertices[2].first * r.vertices[0].second -
			r.vertices[0].first * r.vertices[2].second);
```

```
	res = 0.5 * std::abs(res);

	return res;
}

template<typename T>
std::ostream &Print(std::ostream &os, const Rectangle<T> &r) {
	for (int i = 0; i < 4; i++) {
		os << r.vertices[i];
		if (i != 3) {
			os << " ";
		}
	}

	return os;
}

template<typename T>
std::istream &Read(std::istream &is, Rectangle<T> &r) {
	for (int i = 0; i < 4; i++) {
		is >> r.vertices[i].first >> r.vertices[i].second;
	}
	Vector<T> AB = {r.vertices[0], r.vertices[1]},
		  BC = {r.vertices[1], r.vertices[2]},
		  CD = {r.vertices[2], r.vertices[3]},
		  DA = {r.vertices[3], r.vertices[0]};
	if (!is_parallel(DA, BC)) {
		std::swap(r.vertices[0], r.vertices[1]);
		AB = {r.vertices[0], r.vertices[1]};
	   BC = {r.vertices[1], r.vertices[2]};
	   CD = {r.vertices[2], r.vertices[3]};
	   DA = {r.vertices[3], r.vertices[0]};
	}
	if (!is_parallel(AB, CD)) {
		std::swap(r.vertices[1], r.vertices[2]);
		AB = {r.vertices[0], r.vertices[1]};
	   BC = {r.vertices[1], r.vertices[2]};
	   CD = {r.vertices[2], r.vertices[3]};
	   DA = {r.vertices[3], r.vertices[0]};
	}
	if (AB * BC || BC * CD || CD * DA || DA * AB) {
		throw std::logic_error("The sides of the Rectangle should be perpendicular");
	}
	if (!Length(AB) || !Length(BC) || !Length(CD) || !Length(DA)) {
		throw std::logic_error("The sides of the Rectangle must be greater than zero");
	}


	return is;
}

template<typename T>
```

```
std::istream &operator>>(std::istream &is, Rectangle<T> &r) {
	return Read(is, r);
}

template<typename T>
std::ostream &operator<<(std::ostream &os, const Rectangle<T> &r) {
	return Print(os, r);
}

#endif // RECTANGLE_H
```

square.h:

```
#ifndef SQUARE_H
#define SQUARE_H 1

#include <utility>
#include <iostream>

#include "vector.h"

template<typename T>
struct Square {
	using vertex_t = std::pair<T,T>;
	vertex_t vertices[4];
};

template<typename T>
typename Square<T>::vertex_t Center(const Square<T> &s);

template<typename T>
double Area(const Square<T> &s);

template<typename T>
std::ostream &Print(std::ostream &os, const Square<T> &s);

template<typename T>
std::istream &Read(std::istream &is, Square<T> &s);

template<typename T>
std::istream &operator>>(std::istream &is, Square<T> &s);

template<typename T>
std::ostream &operator<<(std::ostream &os, const Square<T> &s);

template<typename T>
typename Square<T>::vertex_t Center(const Square<T> &s) {
	T x, y;
	x = (s.vertices[0].first + s.vertices[1].first + s.vertices[2].first + s.vertices[3].first) / 4;
	y = (s.vertices[0].second + s.vertices[1].second + s.vertices[2].second + s.vertices[3].second) /
4;

	return std::make_pair(x, y);
```

```
}

template<typename T>
double Area(const Square<T> &s) {
	double res = 0;
	for (int i = 0; i <= 2; i++) {
		res += (s.vertices[i].first * s.vertices[i + 1].second -
				s.vertices[i + 1].first * s.vertices[i].second);
	}
	res += (s.vertices[2].first * s.vertices[0].second -
			s.vertices[0].first * s.vertices[2].second);
	res = 0.5 * std::abs(res);

	return res;
}

template<typename T>
std::ostream &Print(std::ostream &os, const Square<T> &s) {
	for (int i = 0; i < 4; i++) {
		os << s.vertices[i];
		if (i != 3) {
			os << " ";
		}
	}

	return os;
}

template<typename T>
std::istream &Read(std::istream &is, Square<T> &s) {
	for (int i = 0; i < 4; i++) {
		is >> s.vertices[i].first >> s.vertices[i].second;
	}
	Vector<T> AB = {s.vertices[0], s.vertices[1]},
		BC = {s.vertices[1], s.vertices[2]},
		CD = {s.vertices[2], s.vertices[3]},
		DA = {s.vertices[3], s.vertices[0]};
	if (!is_parallel(DA, BC)) {
		std::swap(s.vertices[0], s.vertices[1]);
		AB = {s.vertices[0], s.vertices[1]};
	   BC = {s.vertices[1], s.vertices[2]};
	   CD = {s.vertices[2], s.vertices[3]};
	   DA = {s.vertices[3], s.vertices[0]};
	}
	if (!is_parallel(AB, CD)) {
		std::swap(s.vertices[1], s.vertices[2]);
		AB = {s.vertices[0], s.vertices[1]};
	   BC = {s.vertices[1], s.vertices[2]};
	   CD = {s.vertices[2], s.vertices[3]};
	   DA = {s.vertices[3], s.vertices[0]};
	}
	if (AB * BC || BC * CD || CD * DA || DA * AB) {
```

```
            throw std::logic_error("The sides of the square should be perpendicular");
        }
        if (Length(AB) != Length(BC) || Length(BC) != Length(CD) || Length(CD) != Length(DA) ||
Length(DA) != Length(AB)) {
            throw std::logic_error("The sides of the square should be equal");
        }
        if (!Length(AB) || !Length(BC) || !Length(CD) || !Length(DA)) {
            throw std::logic_error("The sides of the square must be greater than zero");
        }


        return is;
}

template<typename T>
std::istream &operator>>(std::istream &is, Square<T> &s) {
        return Read(is, s);
}

template<typename T>
std::ostream &operator<<(std::ostream &os, const Square<T> &s) {
        return Print(os, s);
}

#endif // SQUARE_H
```

triangle.h:

```
#ifndef TRIANGLE_H
#define TRIANGLE_H

#include <utility>
#include <iostream>

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

template<typename T>
struct Triangle {
        using vertex_t = std::pair<T,T>;
        vertex_t vertices[3];
};

template<typename T>
typename Triangle<T>::vertex_t Center(const Triangle<T> &t);

template<typename T>
double Area(const Triangle<T> &t);

template<typename T>
std::ostream &Print(std::ostream &os, const Triangle<T> &t);

template<typename T>
```

```
std::istream &Read(std::istream &is, Triangle<T> &t);

template<typename T>
std::istream &operator>>(std::istream &is, Triangle<T> &t);

template<typename T>
std::ostream &operator<<(std::ostream &os, const Triangle<T> &t);

template<typename T>
typename Triangle<T>::vertex_t Center(const Triangle<T> &t) {
	T x, y;
	x = (t.vertices[0].first + t.vertices[1].first + t.vertices[2].first) / 3;
	y = (t.vertices[0].second + t.vertices[1].second + t.vertices[2].second) / 3;

	return std::make_pair(x, y);
}

template<typename T>
double Area(const Triangle<T> &t) {
	double res = 0;
	for (int i = 0; i <= 1; i++) {
		res += (t.vertices[i].first * t.vertices[i + 1].second -
				t.vertices[i + 1].first * t.vertices[i].second);
	}
	res += (t.vertices[2].first * t.vertices[0].second -
			t.vertices[0].first * t.vertices[2].second);
	res = 0.5 * std::abs(res);

	return res;
}

template<typename T>
std::ostream &Print(std::ostream &os, const Triangle<T> &t) {
	for (int i = 0; i < 3; i++) {
		os << t.vertices[i];
		if (i != 2) {
			os << " ";
		}
	}

	return os;
}

template<typename T>
std::istream &Read(std::istream &is, Triangle<T> &t) {
	for (int i = 0; i < 3; i++) {
		is >> t.vertices[i].first >> t.vertices[i].second;
	}
	double AB = Length(t.vertices[0], t.vertices[1]),
		BC = Length(t.vertices[1], t.vertices[2]),
		AC = Length(t.vertices[0], t.vertices[2]);
	if (AB >= BC + AC || BC >= AB + AC || AC >= AB + BC) {
```

```
            throw std::logic_error("Vertices must not be on the same line.");
        }

        return is;
}

template<typename T>
std::istream &operator>>(std::istream &is, Triangle<T> &t) {
        return Read(is, t);
}

template<typename T>
std::ostream &operator<<(std::ostream &os, const Triangle<T> &t) {
        return Print(os, t);
}

#endif // TRIANGLE_H
```

2. Ссылка на репозиторий на GitHub.

https://github.com/SergeiPetrin/OOP/tree/master/oop_exercise_04

3. Набор testcases.

test_00.test:

```
Figure
Triangle
0 0 1 1 1 0
Figure
Square
0 0 1 0 1 1 0 1
Figure
Rectangle
0 0 2 0 2 1 0 1
```

test_01.test:

```
Tuple
Triangle
0 0 1 1 2 2
Tuple
Square
-5 -5 -5 5 5 5 5 -5
Tuple
Rectangle
-5 -5 -5 10 5 10 5 -5
```

4. Результаты выполнения тестов.

test_00.result:

Figure / Tuple

Figure

Triangle / Square / Rectangle

Triangle

Input points: 0 0 1 1 1 0
[0, 0] [1, 1] [1, 0]
0.5
[0.666667, 0.333333]

Figure / Tuple

Figure

Triangle / Square / Rectangle

Square

Input points: 0 0 1 0 1 1 0 1
[0, 0] [1, 0] [1, 1] [0, 1]
1
[0.5, 0.5]

Figure / Tuple

Figure

Triangle / Square / Rectangle

Rectangle

Input points: 0 0 2 0 2 1 0 1
[0, 0] [2, 0] [2, 1] [0, 1]
2
[1, 0.5]

---

test_01.result:

Figure / Tuple

Tuple

Triangle / Square / Rectangle

Triangle

Input points: 0 0 1 1 2 2
Vertices must not be on the same line.

Figure / Tuple

Tuple

Triangle / Square / Rectangle

Square

Input points: -5 -5 -5 5 5 5 5 -5
[-5, -5] [-5, 5] [5, 5] [5, -5]
100
[0, 0]

Figure / Tuple

Tuple

Triangle / Square / Rectangle

Rectangle

Input points: -5 -5 -5 10 5 10 5 -5
[-5, -5] [-5, 10] [5, 10] [5, -5]
150
[0, 2.5]

5. Объяснение результатов работы программы.

1) Шаблонная функция Center() возвращает вершину, первой координатой которой является деление суммы иксов всех точек данной фигуры на их количество, со второй координатой аналогично. Эта функция определена для классов фигур и tuple. Во втором случае центр вычисляется рекурсивно.

2) Функция Print() выводит координаты всех точек данной фигуры. Эта функция определена для классов фигур и tuple. Во втором случае печать выполняется рекурсивно.

3) Функция Area() вычисляет площадь данной фигуры по методу Гаусса.

6. Вывод.

Научился использовать шаблоны, где в качестве параметра используются скалярными данные, для работы с шаблонными классами и кортежами. Узнал о применении шаблонов в метапрограммировании. Также я познакомился с полезными заголовочными файлами <tuple> и <type_traits>.