Московский Авиационный Институт
(Национальный исследовательский Университет)

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

Лабораторная работа
по курсу «ООП»

Тема:
Асинхронное программирование.

Студент:	Марков А.Н.
Группа:	М80-208Б-18
Преподаватель:	Журавлев А.А.
Вариант:	17
Оценка:	
Дата:	

Москва
2019

1. Постановка задачи

Вариант №17. Фигуры: треугольник, квадрат, прямоугольник.
Программа должна содержать внутренний буфер, в который помещаются фигуры. Для создания буфера допускается использовать контейнеры STL. Размер буфера задается параметром командной строки. При накоплении буфера они должны запускаться на асинхронную обработку, после чего буфер должен очищаться. Обработка должна производиться в отдельном потоке. Реализовать два обработчика, которые должны обрабатывать данные буфера:

- Вывод информации о фигурах в буфере на экран;
- Вывод информации о фигурах в буфере в файл. Для каждого буфера должен создаваться файл с уникальным именем.

В программе должно быть равно два потока. Один основной и второй для обработчиков.

2. Код программы на языке C++

main.cpp:

```
#include <condition_variable>
#include <fstream>
#include <iostream>
#include <memory>
#include <mutex>
#include <string>
#include <thread>
#include <vector>
#include <cstdlib>

#include "factory.h"
#include "figures.h"
#include "subscriber.h"

int main(int argc, char *argv[]) {
	if (argc != 2) {
		std::cout << "./name size\n";
		return 1;
	}

	size_t vector_size = std::atoi(argv[1]);
	Factory factory;

	Subscriber subscriber;
	subscriber.buffer.reserve(vector_size);
	subscriber.processes.push_back(std::make_shared<Console_process>());
	subscriber.processes.push_back(std::make_shared<File_process>());
```

```
    std::thread subscriber_thread(std::ref(subscriber));

    std::string cmd;
    std::cout << "quit or add\n";
    while (std::cin >> cmd) {
        std::unique_lock<std::mutex> main_lock(subscriber.mtx);
        if (cmd == "quit" || cmd == "exit" || cmd == "q" || cmd == "e") {
            subscriber.end = true;
            subscriber.cv.notify_all();
            break;
        } else if (cmd == "add" || cmd == "a") {
            std::string figure_type;

            for (size_t id = 0; id < vector_size; id++) {
                std::cout << "figure type\n";
                std::cin >> figure_type;
                if (figure_type == "triangle" || figure_type == "t") {
                    std::pair<double, double> *vertices = new std::pair<double,
double>[3];
                    for (int i = 0; i < 3; i++) {
                        std::cin >> vertices[i].first >> vertices[i].second;
                    }
                    try {
                        subscriber.buffer.push_back(factory.FigureCreate(TRIANGLE,
vertices, id));
                    } catch (std::logic_error &e) {
                        std::cout << e.what() << "\n";
                        id--;
                    }
                } else if (figure_type == "square" || figure_type == "s") {
                    std::pair<double, double> *vertices = new std::pair<double,
double>[4];
                    for (int i = 0; i < 4; i++) {
                        std::cin >> vertices[i].first >> vertices[i].second;
                    }
                    try {
                        subscriber.buffer.push_back(factory.FigureCreate(SQUARE,
vertices, id));
                    } catch (std::logic_error &e) {
                        std::cout << e.what() << "\n";
                        id--;
                    }
                } else if (figure_type == "rectangle" || figure_type == "r") {
                    std::pair<double,   double>   *vertices   =   new   std::pair<double,
double>[4];
                    for (int i = 0; i < 4; i++) {
                        std::cin >> vertices[i].first >> vertices[i].second;
                    }
                    try {

    subscriber.buffer.push_back(factory.FigureCreate(RECTANGLE, vertices, id));
                    } catch (std::logic_error &e) {
                        std::cout << e.what() << "\n";
                        id--;
                    }
                }
            }

            if (subscriber.buffer.size() == vector_size) {
                //main_lock.unlock();
                subscriber.cv.notify_all();
                subscriber.cv.wait(main_lock, [&subscriber]() {
                    return subscriber.success == true;
                });
                subscriber.success = false;
            }
        }
    }

    subscriber_thread.join();

    return 0;
}
```

subscriber.h:

```
#ifndef SUBSCRIBER_H
#define SUBSCRIBER_H 1

struct Subscribers_process {
    virtual void Process(std::vector<std::shared_ptr<Figure>> &buffer) = 0;
    virtual ~Subscribers_process() = default;
};

struct Console_process : Subscribers_process {
    void Process(std::vector<std::shared_ptr<Figure>> &buffer) override {
        for (const auto figure : buffer) {
            figure->Print(std::cout);
        }
    }
};

struct File_process : Subscribers_process {
    void Process(std::vector<std::shared_ptr<Figure>> &buffer) override {
        std::string filename;
        std::cout << "Input name of file: ";
        std::cin >> filename;
        std::ofstream os(filename);
        for (const auto figure : buffer) {
            figure->Print(os);
        }
    }
};

struct Subscriber {

    void operator()() {
        for(;;) {
            std::unique_lock<std::mutex> guard(mtx);
            cv.wait(guard, [&](){
                return buffer.size() == buffer.capacity() || end;
            });
            if (end) {
                break;
            }
            for (size_t i = 0; i < processes.size(); i++) {
                processes[i]->Process(buffer);
            }
            buffer.clear();
            success = true;
            cv.notify_all();
        }
    }

    bool end = false;
    bool success = false;
    std::vector<std::shared_ptr<Figure>> buffer;
    std::vector<std::shared_ptr<Subscribers_process>> processes;
    std::condition_variable cv;
    std::mutex mtx;
};

#endif // SUBSCRIBER_H
```

factory.h:

```
#ifndef FACTORY_H
#define FACTORY_H 1

#include "figures.h"

class Factory {
public:
	using Vertex = std::pair<double, double>;

	std::shared_ptr<Figure> FigureCreate(FigureType type) const {
		std::shared_ptr<Figure> res;
		if (type == TRIANGLE) {
			res = std::make_shared<Triangle>();
		} else if (type == SQUARE) {
			res = std::make_shared<Square>();
		} else if (type == RECTANGLE) {
			res = std::make_shared<Rectangle>();
		}

		return res;
	}

	std::shared_ptr<Figure> FigureCreate(FigureType type, Vertex *vertices, int id) const {
		std::shared_ptr<Figure> res;
		if (type == TRIANGLE) {
			res = std::make_shared<Triangle>(vertices[0], vertices[1], vertices[2], id);
		} else if (type == SQUARE) {
			res = std::make_shared<Square>(vertices[0], vertices[1], vertices[2], vertices[3], id);
		} else if (type == RECTANGLE) {
			res = std::make_shared<Rectangle>(vertices[0], vertices[1], vertices[2], vertices[3], id);
		}

		return res;
	}
};

#endif //FACTORY_H
```

figures.h:

```
#ifndef FIGURES_H
#define FIGURES_H 1

#include <iostream>
#include <fstream>
#include <utility>
#include <cmath>
#include <memory>


enum FigureType {
    TRIANGLE,
    SQUARE,
    RECTANGLE
};

class Figure {
public:
    virtual double Area() const = 0;
    virtual std::pair<double, double> Center() const = 0;
    virtual std::ostream &Print(std::ostream &out) const = 0;
    virtual void Serialize(std::ofstream &os) const = 0;
    virtual void Deserialize(std::ifstream &is) = 0;
    virtual int getId() const = 0;
    virtual ~Figure() = default;
};

namespace Geometry {
    using Vertex = std::pair<double, double>;
    double Product(const Vertex &v1, const Vertex &v2) {
        return v1.first * v2.first + v1.second * v2.second;
    }

    double PointDistance(const Vertex &v1, const Vertex &v2) {
        return sqrt(pow((v2.first - v1.first), 2) +
            pow((v2.second - v1.second), 2));
    }

    class Vector {
        double x, y;
    public:
        Vector(double x_cord, double y_cord) : x{x_cord}, y{y_cord} {};

        Vector(Vertex &v1, Vertex &v2) : x{v2.first - v1.first},
                        y{v2.second - v1.second} {};

        double operator*(const Vector &a) const {
            return (x * a.x) + (y * a.y);
        }

        Vector &operator=(const Vector &a) {
            x = a.x;
            y = a.y;

            return *this;
        }
        friend double LengthVector(const Vector &a);
        friend bool VectorsAreParallel(const Vector &a, const Vector &b);
    };

    double LengthVector(const Vertex &v1, const Vertex &v2) {
        return PointDistance(v1, v2);
    }

    double LengthVector(const Vector &a) {
        return sqrt(pow(a.x, 2) + pow(a.y, 2));
    }

    bool VectorsAreParallel(const Vector &a, const Vector &b) {
        return (a.x * b.y) - (a.y * b.x) == 0;
    }

    double Area(const Vertex *vertices, int n) {
        double res = 0;

        for (int i = 0; i < n - 1; i++) {
            res += (vertices[i].first * vertices[i + 1].second -
                vertices[i + 1].first * vertices[i].second);
        }
        res += (vertices[n - 1].first * vertices[0].second -
                vertices[0].first * vertices[n - 1].second);

        return 0.5 * std::abs(res);
    }

    Vertex Center(const Vertex *vertices, int n) {
        double x = 0, y = 0;

        for (int i = 0; i < n; i++) {
            x += vertices[i].first;
            y += vertices[i].second;
        }

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

std::ostream &operator<<(std::ostream &out, std::pair<double, double> v) {
    out << "(" << v.first << ", " << v.second << ")";
    return out;
}

class Triangle : public Figure {
    using Vertex = std::pair<double, double>;
    int Id;
    Vertex *vertices;
public:
    Triangle() : Id{0}, vertices{new Vertex[3]} {
        for (int i = 0; i < 3; i++) {
            vertices[i] = std::make_pair(0, 0);
        }
    }

    Triangle(Vertex a, Vertex b, Vertex c, int id) : Id{id},
                                    vertices{new Vertex[3]} {
        vertices[0] = a;
        vertices[1] = b;
        vertices[2] = c;
        double AB = Geometry::PointDistance(a, b), BC =
        Geometry::PointDistance(b, c), AC = Geometry::PointDistance(a, c);
        if (AB >= BC + AC || BC >= AB + AC || AC >= AB + BC) {
            throw std::logic_error("Points must not be on the same line.");
        }
    }

    ~Triangle() {
        delete [] vertices;
        vertices = nullptr;
    }

    double Area() const override {
        return Geometry::Area(vertices, 3);
    }
    Vertex Center() const override {
        return Geometry::Center(vertices, 3);
    }

    std::ostream &Print(std::ostream &out) const override{
        out << "Id: " << Id << "\n";
        out << "Figure: Triangle\n";
        out << "Coords:\n";
        for (int i = 0; i < 3; i++) {
            out << vertices[i] << "\n";
        }
        return out;
    }

    void Serialize(std::ofstream &os) const override{
        FigureType type = TRIANGLE;
        os.write((char *) &type, sizeof(type));
        os.write((char *) &Id, sizeof(Id));
        for (int i = 0; i < 3; i++) {
            os.write((char *) &(vertices[i].first),
                sizeof(vertices[i].first));
            os.write((char *) &(vertices[i].second),
                sizeof(vertices[i].second));
        }
    }

    void Deserialize(std::ifstream &is) override {
        is.read((char *) &Id, sizeof(Id));
        for (int i = 0; i < 3; i++) {
            is.read((char *) &(vertices[i].first),
                sizeof(vertices[i].first));
            is.read((char *) &(vertices[i].second),
                sizeof(vertices[i].second));
        }
    }

    int getId() const override {
        return Id;
    }
};

class Square : public Figure {
    using Vertex = std::pair<double, double>;
    int Id;
    Vertex *vertices;
```

```
public:
    Square() : Id{0}, vertices{new Vertex[4]} {
        for (int i = 0; i < 4; i++) {
            vertices[i] = std::make_pair(0, 0);
        }
    }

    Square(Vertex a, Vertex b, Vertex c, Vertex d, int id) :
                        Id{id}, vertices{new Vertex[4]} {
        vertices[0] = a;
        vertices[1] = b;
        vertices[2] = c;
        vertices[3] = d;
        Geometry::Vector AB{ a, b }, BC{ b, c }, CD{ c, d }, DA{ d, a };
        if (!Geometry::VectorsAreParallel(DA, BC)) {
            std::swap(vertices[0], vertices[1]);
            AB = { vertices[0], vertices[1] };
            BC = { vertices[1], vertices[2] };
            CD = { vertices[2], vertices[3] };
            DA = { vertices[3], vertices[0] };
        }
        if (!Geometry::VectorsAreParallel(AB, CD)) {
            std::swap(vertices[1], vertices[2]);
            AB = { vertices[0], vertices[1] };
            BC = { vertices[1], vertices[2] };
            CD = { vertices[2], vertices[3] };
            DA = { vertices[3], vertices[0] };
        }

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

    ~Square() {
        delete [] vertices;
        vertices = nullptr;
    }

    double Area() const override {
        return Geometry::Area(vertices, 4);
    }

    Vertex Center() const override {
        return Geometry::Center(vertices, 4);
    }

    std::ostream &Print(std::ostream &out) const override{
        out << "Id: " << Id << "\n";
        out << "Figure: Square\n";
        out << "Coords:\n";
        for (int i = 0; i < 4; i++) {
            out << vertices[i] << "\n";
        }
        return out;
    }

    void Serialize(std::ofstream &os) const override{
        FigureType type = SQUARE;
        os.write((char *) &type, sizeof(type));
        os.write((char *) &Id, sizeof(Id));
        for (int i = 0; i < 4; i++) {
            os.write((char *) &(vertices[i].first),
                sizeof(vertices[i].first));
            os.write((char *) &(vertices[i].second),
                sizeof(vertices[i].second));
        }
    }

    void Deserialize(std::ifstream &is) override {
        is.read((char *) &Id, sizeof(Id));
        for (int i = 0; i < 4; i++) {
            is.read((char *) &(vertices[i].first),
                sizeof(vertices[i].first));
            is.read((char *) &(vertices[i].second),
                sizeof(vertices[i].second));
        }
    }

    int getId() const override {
        return Id;
    }
};

class Rectangle : public Figure {
    using Vertex = std::pair<double, double>;
    int Id;
    Vertex *vertices;
public:
    Rectangle() : Id{0}, vertices{new Vertex[4]} {
        for (int i = 0; i < 4; i++) {
            vertices[i] = std::make_pair(0, 0);
        }
    }

    Rectangle(Vertex a, Vertex b, Vertex c, Vertex d, int id) :
                    Id{id}, vertices{new Vertex[4]} {
        vertices[0] = a;
        vertices[1] = b;
        vertices[2] = c;
        vertices[3] = d;
        Geometry::Vector AB{ a, b }, BC{ b, c }, CD{ c, d }, DA{ d, a };
        if (!Geometry::VectorsAreParallel(DA, BC)) {
            std::swap(vertices[0], vertices[1]);
            AB = { vertices[0], vertices[1] };
            BC = { vertices[1], vertices[2] };
            CD = { vertices[2], vertices[3] };
            DA = { vertices[3], vertices[0] };
        }
        if (!Geometry::VectorsAreParallel(AB, CD)) {
            std::swap(vertices[1], vertices[2]);
            AB = { vertices[0], vertices[1] };
            BC = { vertices[1], vertices[2] };
            CD = { vertices[2], vertices[3] };
            DA = { vertices[3], vertices[0] };
        }

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

```
}

~Rectangle() {
    delete [] vertices;
    vertices = nullptr;
}

double Area() const override {
    return Geometry::Area(vertices, 4);
}

Vertex Center() const override {
    return Geometry::Center(vertices, 4);
}

std::ostream &Print(std::ostream &out) const override{
    out << "Id: " << Id << "\n";
    out << "Figure: Rectangle\n";
    out << "Coords:\n";
    for (int i = 0; i < 4; i++) {
        out << vertices[i] << "\n";
    }
    return out;
}

void Serialize(std::ofstream &os) const override{
    FigureType type = RECTANGLE;
    os.write((char *) &type, sizeof(type));
    os.write((char *) &Id, sizeof(Id));
    for (int i = 0; i < 4; i++) {
        os.write((char *) &(vertices[i].first),
            sizeof(vertices[i].first));
        os.write((char *) &(vertices[i].second),
            sizeof(vertices[i].second));
    }
}

void Deserialize(std::ifstream &is) override {
    is.read((char *) &Id, sizeof(Id));
    for (int i = 0; i < 4; i++) {
        is.read((char *) &(vertices[i].first),
            sizeof(vertices[i].first));
        is.read((char *) &(vertices[i].second),
            sizeof(vertices[i].second));
    }
    }

    int getId() const override {
        return Id;
    }
};

#endif // FIGURES_H
```

3. Ссылка на репозиторий на GitHub.

https://github.com/Markov-A-N/oop_exercise_08.git

4. Набор testcases.

test_01.txt:

```
add
t
0 0 1 1 10 0
s
0 0 1 1 1 0 0 1
r
0 0 2 0 2 1 0 1
file
quit
```

test_02.txt:

```
add
t
0 0 0 0 0 0
t
0 0 1 1 1 0
t
0 0 2 2 2 0
file
quit
```

5. Результаты выполнения тестов.

test_01.txt:

```
oem@Alex-PC:~/Documents/oop/oop_exercise_08/build$ ./oop_exercise_08 3 < ../tests/test_01.txt
quit or add
```

```
figure type
figure type
figure type
Id: 0
Figure: Triangle
Coords:
(0, 0)
(1, 1)
(10, 0)
Id: 1
Figure: Square
Coords:
(0, 0)
(1, 0)
(1, 1)
(0, 1)
Id: 2
Figure: Rectangle
Coords:
(0, 0)
(2, 0)
(2, 1)
(0, 1)
Input name of file:
```

test_02.txt:

```
quit or add
figure type
Points must not be on the same line.
figure type
figure type
Id: 0
Figure: Triangle
Coords:
(0, 0)
(1, 1)
(1, 0)
Id: 1
Figure: Triangle
Coords:
(0, 0)
(2, 2)
(2, 0)
Input name of file:
```

6. Объяснение результатов работы программы.

Основная идея программы в том, что она имеет 2 потока. Основной считывает команды и добавляет элементы в буфер. Когда буфер заполняется, основной поток уведомляет второй и начинает ждать ответа от него. Второй поток вызывает вывод в консоль и в файл. Затем буфер очищается и второй поток уведомляет основной, после чего цикл повторяется.

7. Вывод.

Выполняя данную работу, я познакомился с thread, mutex, unique_lock, condition_variable и использовал их в своей программе.