Московский Авиационный Институт
(Национальный Исследовательский Университет)

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

Лабораторная работа
Дисциплина: «Объектно-ориентированное программирование»
III семестр
Задание 8: «Асинхронное программирование»

Группа:	М8О-206Б-18, №27
Студент:	Шорохов Алексей Павлович
Преподаватель:	Журавлёв Андрей Андреевич
Оценка:	
Дата:	.12.2019

Москва, 2019

Задание

27.	Прямоугольник	Трапеция	Ромб

Адрес репозитория на GitHub

Код программы на C++

Point.h

```
#ifndef D_POINT_H_
#define D_POINT_H_

#include <iostream>

struct Point {

	double x, y;

};

std::istream& operator>> (std::istream& is, Point &p);
std::ostream& operator<< (std::ostream& os, const Point &p);
bool operator == (Point a, Point b);

#endif
```

Point.cpp

```
#include "point.h"

std::istream& operator >> (std::istream& is, Point &p) {
	return is >> p.x >> p.y;
}

std::ostream& operator << (std::ostream& os, const Point &p) {
   return os << p.x << " " << p.y << '\n';
}

bool operator == (Point a, Point b) {
	return (a.x == b.x && a.y == b.y);
}
```

Rectangle.h

```
#ifndef D_RECTANGLE_H_
#define D_RECTANGLE_H_

#include "figure.h"
```

```
class Rectangle : public Figure {
public:
	Rectangle() = default;
	Rectangle (std::istream&);
	Rectangle (Point p1, Point p2, Point p3, Point p4);

	Point center() const override;
	void print(std::ostream&) const override;
	void input(std::istream&) override;
	double square() const override;
private:
	Point p1, p2, p3, p4;
};


#endif
```

Rectangle.cpp

```
#include "rectangle.h"
#include <iostream>
#include <cmath>


Rectangle::Rectangle(std::istream& is) {
	is >> p1 >> p2 >> p3 >> p4;

	if (isPerpendicular(Vector(p1, p2), Vector(p1, p4)) &&
isPerpendicular(Vector(p1, p2), Vector(p2, p3)) &&
		isPerpendicular(Vector(p2, p3), Vector(p3, p4)) &&
isPerpendicular(Vector(p3, p4), Vector(p1, p4))) {



	} else if (isPerpendicular(Vector(p1, p4), Vector(p4, p2)) &&
isPerpendicular(Vector(p4, p2), Vector(p2, p3)) &&
		isPerpendicular(Vector(p2, p3), Vector(p3, p1)) &&
isPerpendicular(Vector(p1, p3), Vector(p1, p4))) {

			Point tmp;
			tmp = p1;
			p1 = p4;
			p4 = tmp;

	} else if (isPerpendicular(Vector(p1, p2), Vector(p2, p4)) &&
isPerpendicular(Vector(p2, p4), Vector(p4, p3)) &&
```

```
            isPerpendicular(Vector(p4, p3), Vector(p3, p1)) &&
isPerpendicular(Vector(p3, p1), Vector(p1, p2))) {

                Point tmp;
                tmp = p3;
                p3 = p4;
                p4 = tmp;

        } else if (p1 == p2 || p1 == p3 || p1 == p4 || p2 == p3 || p2 == p4 || p3 == p4)
{
            throw std::logic_error("No points are able to be equal");
        } else {
            throw std::logic_error("That's not a Rectangle, sides are not
Perpendicular");
        }

        if (!(Vector(p1, p2).length() == Vector(p3, p4).length() && Vector(p2,
p3).length() == Vector(p1, p4).length())) {
            throw std::logic_error("That's not a Rectangle, sides are not equal");
        }

}

Rectangle::Rectangle(Point p1, Point p2, Point p3, Point p4) : p1(p1), p2(p2),
p3(p3), p4(p4) {

        if (isPerpendicular(Vector(p1, p2), Vector(p1, p4)) &&
isPerpendicular(Vector(p1, p2), Vector(p2, p3)) &&
            isPerpendicular(Vector(p2, p3), Vector(p3, p4)) &&
isPerpendicular(Vector(p3, p4), Vector(p1, p4))) {



        } else if (isPerpendicular(Vector(p1, p4), Vector(p4, p2)) &&
isPerpendicular(Vector(p4, p2), Vector(p2, p3)) &&
            isPerpendicular(Vector(p2, p3), Vector(p3, p1)) &&
isPerpendicular(Vector(p1, p3), Vector(p1, p4))) {

                Point tmp;
                tmp = p1;
                p1 = p4;
                p4 = tmp;

        } else if (isPerpendicular(Vector(p1, p2), Vector(p2, p4)) &&
isPerpendicular(Vector(p2, p4), Vector(p4, p3)) &&
```

```
            isPerpendicular(Vector(p4, p3), Vector(p3, p1)) &&
isPerpendicular(Vector(p3, p1), Vector(p1, p2))) {

                Point tmp;
                tmp = p3;
                p3 = p4;
                p4 = tmp;

        } else if (p1 == p2 || p1 == p3 || p1 == p4 || p2 == p3 || p2 == p4 || p3 == p4)
{
            throw std::logic_error("No points are able to be equal");
        } else {
            throw std::logic_error("That's not a Rectangle, sides are not
Perpendicular");
        }

        if (!(Vector(p1, p2).length() == Vector(p3, p4).length() && Vector(p2,
p3).length() == Vector(p1, p4).length())) {
            throw std::logic_error("That's not a Rectangle, sides are not equal");
        }


}

Point Rectangle::center() const{
        Point p;
        p.x = (p1.x + p2.x + p3.x + p4.x) / 4;
        p.y = (p1.y + p2.y + p3.y + p4.y) / 4;
        return p;
}

void Rectangle::print(std::ostream& os) const{
        os << "Rectangle\n";
        os << p1 << p2 << p3 << p4;
}

void Rectangle::input(std::istream& is) {
        Point p1,p2,p3,p4;
        is >> p1 >> p2 >> p3 >> p4;
        *this = Rectangle(p1,p2,p3,p4);
}

double Rectangle::square() const{

        return Vector(p1, p2).length() * Vector(p2, p3).length();
```

```
}
```
Rhombus.h

```
#ifndef D_RHOMBUS_H_
#define D_RHOMBUS_H_

#include "figure.h"

class Rhombus : public Figure {
public:
	Rhombus() = default;
	Rhombus (std::istream&);
	Rhombus (Point p1, Point p2, Point p3, Point p4);

	Point center() const override;
	void print(std::ostream&) const override;
	void input(std::istream&) override;
	double square() const override;
private:
	Point p1, p2, p3, p4;
};


#endif
```
Rhombus.cpp
```
#include "rhombus.h"
#include <iostream>
#include <cmath>


Rhombus::Rhombus(std::istream& is) {
	is >> p1 >> p2 >> p3 >> p4;

	if (Vector(p1, p2).length() == Vector(p2, p3).length() && Vector(p2,
p3).length() == Vector(p3, p4).length()
	 && Vector(p1, p2).length() == Vector(p1, p4).length()) {

	} else if (Vector(p1, p2).length() == Vector(p2, p4).length() && Vector(p2,
p4).length() == Vector(p3, p4).length()
	 && Vector(p1, p2).length() == Vector(p1, p3).length()) {
		Point tmp = p4;
		p4 = p3;
		p3 = tmp;
```

```
	} else if (Vector(p1, p3).length() == Vector(p4, p3).length() && Vector(p4,
p3).length() == Vector(p2, p4).length()
		&& Vector(p1, p2).length() == Vector(p1, p3).length()) {
			Point tmp = p4;
			p4 = p3;
			p3 = tmp;
	} else if (p1 == p2 || p1 == p3 || p1 == p4 || p2 == p3 || p2 == p4 || p3 == p4)
{
			throw std::logic_error("No points are able to be equal");
	} else {
			throw std::logic_error("This is not a Rhombus, sides are not equal");
	}


	Vector v1(p1, p2);
	Vector v2(p2, p3);
	Vector v3(p3, p4);
	Vector v4(p4, p1);

	double cos1 = v1 * v2 / (v1.length() * v2.length());
	double cos2 = v2 * v3 / (v2.length() * v3.length());
	double cos3 = v3 * v4 / (v3.length() * v4.length());
	double cos4 = v1 * v4 / (v1.length() * v4.length());

	if (cos1 != cos3 || cos2 != cos4) {
			throw std::logic_error("This is not a Rhombus, opposite angles are not
equal");
	}
}

Rhombus::Rhombus(Point p1, Point p2, Point p3, Point p4) : p1(p1), p2(p2),
p3(p3), p4(p4) {

	if (Vector(p1, p2).length() == Vector(p2, p3).length() && Vector(p2,
p3).length() == Vector(p3, p4).length()
		&& Vector(p1, p2).length() == Vector(p1, p4).length()) {

	} else if (Vector(p1, p2).length() == Vector(p2, p4).length() && Vector(p2,
p4).length() == Vector(p3, p4).length()
		&& Vector(p1, p2).length() == Vector(p1, p3).length()) {
			Point tmp = p4;
			p4 = p3;
			p3 = tmp;
	} else if (Vector(p1, p3).length() == Vector(p4, p3).length() && Vector(p4,
p3).length() == Vector(p2, p4).length()
```

```
        && Vector(p1, p2).length() == Vector(p1, p3).length()) {
            Point tmp = p4;
            p4 = p3;
            p3 = tmp;
        } else if (p1 == p2 || p1 == p3 || p1 == p4 || p2 == p3 || p2 == p4 || p3 == p4)
{
            throw std::logic_error("No points are able to be equal");
        } else {
            throw std::logic_error("This is not a Rhombus, sides are not equal");
        }


        Vector v1(p1, p2);
        Vector v2(p2, p3);
        Vector v3(p3, p4);
        Vector v4(p4, p1);

        double cos1 = v1 * v2 / (v1.length() * v2.length());
        double cos2 = v2 * v3 / (v2.length() * v3.length());
        double cos3 = v3 * v4 / (v3.length() * v4.length());
        double cos4 = v1 * v4 / (v1.length() * v4.length());

        if (cos1 != cos3 || cos2 != cos4) {
            throw std::logic_error("This is not a Rhombus, opposite angles are not
equal");
        }

}

Point Rhombus::center() const{
        Point p;
        p.x = (p1.x + p2.x + p3.x + p4.x) / 4;
        p.y = (p1.y + p2.y + p3.y + p4.y) / 4;
        return p;
}

void Rhombus::print(std::ostream& os) const{
        os << "Rhombus\n";
        os << p1 << p2 << p3 << p4;
}

void Rhombus::input(std::istream& is)  {
        Point p1,p2,p3,p4;
        is >> p1 >> p2 >> p3 >> p4;
        *this = Rhombus(p1,p2,p3,p4);
```

```
}

double Rhombus::square() const{
	return Vector(p1, p3).length() * Vector(p2, p4).length() / 2;
}
```

Trapezoid.h

```
#ifndef D_TRAPEZOID_H_
#define D_TRAPEZOID_H_

#include "figure.h"

class Trapezoid : public Figure {
public:
	Trapezoid() = default;
	Trapezoid (std::istream&);
	Trapezoid (Point p1, Point p2, Point p3, Point p4);

	Point center() const override;
	void print(std::ostream&) const override;
	void input(std::istream&) override;
	double square() const override;
private:
	Point p1, p2, p3, p4;
};


#endif
```

Trapezoid.cpp

```
#include "trapezoid.h"
#include <iostream>
#include <cmath>

Trapezoid::Trapezoid(std::istream& is) {
	is >> p1 >> p2 >> p3 >> p4;

	if (isParallel(Vector(p1, p4), Vector(p2, p3))) {

	} else if (isParallel(Vector(p1, p3), Vector(p4, p2))) {

		Point tmp;
		tmp = p2;
		p2 = p4;
		p4 = tmp;
		tmp = p3;
		p3 = p4;
```

```
            p4 = tmp;

        } else if (isParallel(Vector(p1, p3), Vector(p2, p4))) {

            Point tmp;
            tmp = p3;
            p3 = p4;
            p4 = tmp;

        } else if (p1 == p2 || p1 == p3 || p1 == p4 || p2 == p3 || p2 == p4 || p3 == p4)
{
            throw std::logic_error("No points are able to be equal");
        } else {
            throw std::logic_error("At least 2 sides of trapeze must be parallel");
        }
}

Trapezoid::Trapezoid(Point p1, Point p2, Point p3, Point p4) : p1(p1), p2(p2),
p3(p3), p4(p4) {

        if (isParallel(Vector(p1, p4), Vector(p2, p3))) {

        } else if (isParallel(Vector(p1, p3), Vector(p4, p2))) {

            Point tmp;
            tmp = p2;
            p2 = p4;
            p4 = tmp;
            tmp = p3;
            p3 = p4;
            p4 = tmp;

        } else if (isParallel(Vector(p1, p3), Vector(p2, p4))) {

            Point tmp;
            tmp = p3;
            p3 = p4;
            p4 = tmp;

        } else if (p1 == p2 || p1 == p3 || p1 == p4 || p2 == p3 || p2 == p4 || p3 == p4)
{
            throw std::logic_error("No points are able to be equal");
        } else {
            throw std::logic_error("At least 2 sides of trapeze must be parallel");
        }
```

```
}

Point Trapezoid::center() const{
	Point p;
	p.x = (p1.x + p2.x + p3.x + p4.x) / 4;
	p.y = (p1.y + p2.y + p3.y + p4.y) / 4;
	return p;
}

void Trapezoid::print(std::ostream& os) const{
	os << "Trapezoid\n";
	os << p1 << p2 << p3 << p4;
}

void Trapezoid::input(std::istream& is)  {
	Point p1,p2,p3,p4;
	is >> p1 >> p2 >> p3 >> p4;
	*this = Trapezoid(p1,p2,p3,p4);
}

double Trapezoid::square() const{

	double a = p2.y - p3.y;
	double b = p3.x - p2.x;
	double c = p2.x * p3.y - p3.x * p2.y;
	double height = (std::abs(a * p1.x + b * p1.y + c) / sqrt(a * a + b * b));
	return (Vector(p1, p2).length() + Vector(p3, p4).length()) * height / 2;
}
```

Vector.h

```
#ifndef VECTOR_H_
#define VECTOR_H_

#include "point.h"
#include <cmath>
#include <numeric>
#include <limits>

class Vector {
public:
	explicit Vector(Point a, Point b);
	double length() const;
	double x;
	double y;
	friend double operator* (Vector a, Vector b) ;
```

```
        bool operator== (Vector b);
};

bool isParallel(const Vector a, const Vector b);
bool isPerpendicular(const Vector a, const Vector b);



#endif
```

Vector.cpp

```
#include "vector.h"

Vector::Vector(Point a, Point b) {
        x = b.x - a.x;
        y = b.y - a.y;
}

double Vector::length() const{
        return sqrt(x * x + y * y);
}

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

bool isPerpendicular(const Vector a, const Vector b) {
        return (a.x * b.x + a.y * b.y) == 0;
}

double operator* (Vector a, Vector b) {
        return a.x * b.x + a.y * b.y;
}

bool Vector::operator== (Vector b) {
        return std::abs(x - b.x) < std::numeric_limits<double>::epsilon() * 100
        && std::abs(y - b.y) < std::numeric_limits<double>::epsilon() * 100;
}
```

Figure.h

```
#ifndef D_FIGURE_H_
#define D_FIGURE_H_

#include <iostream>


#include "point.h"
```

```
#include "vector.h"


class Figure {
public:
	virtual Point center() const = 0;
	virtual void print(std::ostream&) const = 0;
	virtual void input(std::istream&) = 0;
	virtual double square() const = 0;
	virtual ~Figure() = default;
};

std::ostream& operator << (std::ostream& os, const Figure& f);

#endif // D_FIGURE_H_
```

Figure.cpp

```
#include "figure.h"

std::ostream& operator << (std::ostream& os, const Figure& f) {
	f.print(os);
	return os;
}
```

Subscriber.h

```
#pragma once

#include <fstream>
#include <memory>
#include <vector>
#include <queue>
#include <map>
#include <thread>
#include <mutex>
#include <condition_variable>

#include "figure.h"


class Task {
public:

	Task(bool type, const std::vector<std::shared_ptr<Figure>>& data);
	bool isExit() const;
	std::vector<std::shared_ptr<Figure>> getData() const;

private:
```

```
        bool type;
        std::vector<std::shared_ptr<Figure>> data;
};

struct Subscriber {
public:

        virtual void print(std::shared_ptr<Task> task) const = 0;
        virtual ~Subscriber() = default;

};

struct ConsoleSubscriber : public Subscriber {
public:

        void print(std::shared_ptr<Task> task) const override;

};

struct FileSubscriber : public Subscriber {
public:

        void print(std::shared_ptr<Task> task) const override;

};


class Executor {
public:

        void subscribe(std::shared_ptr<Subscriber>& s);

        void notify(std::shared_ptr<Task> task);

private:

        std::vector<std::shared_ptr<Subscriber>> subscribers;

};
```

Subscriber.cpp

```
#include "Subscriber.h"
```

```
Task::Task(bool type, const std::vector<std::shared_ptr<Figure>>& data) :
type(type), data(data) {};

bool Task::isExit() const {

	return type;

}

std::vector<std::shared_ptr<Figure>> Task::getData() const {

	return data;

}



void ConsoleSubscriber::print(std::shared_ptr<Task> task) const  {

	for (size_t i = 0; i < task -> getData().size(); ++i) {

		task -> getData()[i] -> print(std::cout);

	}

}

void FileSubscriber::print(std::shared_ptr<Task> task) const  {

	std::ofstream os(std::to_string(rand() % 1337) + ".txt");

	for (size_t i = 0; i < task -> getData().size(); ++i) {

		task -> getData()[i] -> print(os);

	}

}

void Executor::subscribe(std::shared_ptr<Subscriber>& s) {

	subscribers.push_back(s);

}
```

```
void Executor::notify(std::shared_ptr<Task> task) {

        for(const auto& subscriber : subscribers) {

                //task -> getData()[0] -> print(std::cout);
                subscriber -> print(task);

        }

}
```

Source.cpp

```
#include <iostream>
#include "Subscriber.h"

#include "figure.h"
#include "rhombus.h"
#include "trapezoid.h"
#include "rectangle.h"


struct ThreadFunc {
public:
    ThreadFunc(const Executor& executor) : executor(executor) {};

    void addTask(std::unique_ptr<Task> task) {

        std::lock_guard<std::mutex> lock(queueMutex);
        tasks.push(std::move(task));

    }

    void startWorking() {
        working = true;
    }

    void stopWorking() {
        working = false;
    }

    bool isWorking() {
        return working;
    }

    std::condition_variable& getCondition1() {
```

```
        return condition1;
    }

    std::condition_variable& getCondition2() {
        return condition2;
    }

    std::mutex& getReadMutex() {
        return readMutex;
    }

    void operator()() {
        while(true) {
            std::unique_lock<std::mutex> mainLock(readMutex);
            while(!working) {
                condition2.wait(mainLock);
            }
            if(!tasks.empty()) {
                {
                    std::lock_guard<std::mutex> lock(queueMutex);
                    std::shared_ptr<Task> currentTask = std::move(tasks.front());
                    tasks.pop();
                    if(currentTask->isExit()) {
                        break;
                    } else {
                        executor.notify(std::move(currentTask));
                    }
                    this->stopWorking();
                    condition1.notify_one();
                }
            }

        }
    }
private:
    Executor executor;
    std::mutex readMutex;
    std::condition_variable condition1;
    std::condition_variable condition2;
    std::mutex queueMutex;
    std::queue<std::shared_ptr<Task>> tasks;
    bool working = false;
};

void menu() {
```

```
    std::cout << "1 : add\n";
    std::cout << "0 : exit\n";
    std::cout << "> ";
}

int main(int argc, char** argv) {

    unsigned bufferSize;
    if(argc != 2) {
        std::cout << "No args!" << std::endl;
        return -1;
    }

    bufferSize = std::atoi(argv[1]);
    std::vector<std::shared_ptr<Figure>> figures;
    int command;
    int command2;

    std::shared_ptr<Subscriber> consolePrint(new ConsoleSubscriber());
    std::shared_ptr<Subscriber> filePrint(new FileSubscriber());

    Executor executor;
    executor.subscribe(consolePrint);
    executor.subscribe(filePrint);

    ThreadFunc func(executor);
    std::thread thread(std::ref(func));

    while(true) {
        menu();
        std::cin >> command;


        if(command == 0) {
            std::unique_ptr<Task>t(new Task(true, figures));

            func.addTask(std::move(t));
            func.startWorking();
            func.getCondition2().notify_one();
            break;
        } else if(command == 1) {
            std::shared_ptr<Figure> f;
            std::cout << "1 - Rhombus, 2 - Rectangle, 3 - Trapezoid" << std::endl;
            std::cin >> command2;
            try {
```

```
            if(command2 == 1) {
                f = std::make_shared<Rhombus>();
                f -> input(std::cin);
            } else if(command2 == 2) {
                f = std::make_shared<Rectangle>();
                f -> input(std::cin);
            } else if(command2 == 3) {
                f = std::make_shared<Trapezoid>();
                f -> input(std::cin);
            } else {
                std::cout << "Wrong input" << std::endl;
            }
            figures.push_back(f);
        } catch(std::exception& e) {
            std::cerr << e.what() << std::endl;
        }
        if(figures.size() == bufferSize) {
            std::unique_ptr<Task>t(new Task(false, figures));
            func.addTask(std::move(t));

            func.startWorking();
            func.getCondition2().notify_one();
            std::unique_lock<std::mutex> lock(func.getReadMutex());
            while(func.isWorking()) {
                func.getCondition1().wait(lock);
            }
            figures.resize(0);
        }
    } else {
        std::cout << "Unknown command" << std::endl;
    }
  }
  thread.join();
  return 0;
}
```

Объяснение результатов

Программа получает на вход команды из меню. В зависимости от команды совершается одно из действий: добавление фигуры в буфер или выход из программы. При заполнении буфера программа выводит в консоль и созданный ею файл введенные пользователем данные и очищает буфер.

Вывод

Я познакомился с темой асинхронного программирования, потоками в C++, а так же с концепцией Publish-Subscribe.