**Московский авиационный институт**

**(Национальный исследовательский университет)**

Факультет: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Объектно-ориентированное программирование»

**Лабораторная работа № 8**

Тема: Асинхронное программирование

Студент: Черемисинов Максим (староста)

Группа: 80-201

Преподаватель: Чернышов Л.Н.

Дата:

Оценка:

Москва, 2019

1. **Постановка задачи**

Написать программу, которая позволяет считывать фигуры, затем, при накоплении буфера, в отдельных обработчиках выводится информация о фигурах и их сохранение в сгенерированный файл с расширением .fig.

1. **Репозиторий github**

<https://github.com/devepodete/oop_exercise_08>

1. **Описание программы**

Пользователь при запуске программы указывает максимальный размер буфера. Фигуры вводятся двумя числами: координатой центра и одной вершины. Когда буфер заполняется, в отдельных потоках запускается 2 обработчика: вывод информации о фигурах и их запись в файл, после чего буфер очищается.

1. **Набор testcases**

**test\_01.txt**

create rhombus 0 0 0 1

Figure added

create pentagon 2 2 3 3

Figure added

.....ID: 1

Figure: Rhombus

Coors:

<0, 1> <-1, 1.22465e-16> <-1.83697e-16, -1> <1, -2.44929e-16>

ID: 2

Figure: Pentagon

Coors:

<3, 3> <1.35796, 3.26007> <0.603198, 1.77877> <1.77877, 0.603198> <3.26007, 1.35796>

quit

**zero@aspire:~/documents/mai/oop$** ls

/\* yixwmikqrw.fig \*/ <- сохраненный файл

//проверка работы обработчиков

**test\_02.txt**

create qwe 1 2 3 4

create hexagon 0 0 3 3

quit

ls

//проверка обработки корректности введенных названий фигур

1. **Результаты выполнения тестов.**

Программа выдает правильные ответы, фигуры сохраняются в файлы, обработчик печатает необходимую информацию

1. **Листинг программы**

**main.cpp**

#include <iostream>

#include <vector>

#include <memory>

#include <thread>

#include <queue>

#include "event.hpp"

#include "figures.hpp"

#include "event\_loop.hpp"

int main(int argc, char \*argv[]){

if(argc != 2){

std::cout << "usage:" << argv[0] << " BUFFER\_SIZE\n";

return 0;

}

int buffer\_size = std::atoi(argv[1]);

std::vector<std::shared\_ptr<IFigure>> shapes;

EventLoop eventLoop;

std::shared\_ptr<IHandler> handlerPrinter(new HandlerPrinter());

std::shared\_ptr<IHandler> handlerLogger(new HandlerLogger());

eventLoop.addHandler(EventType::print, handlerPrinter);

eventLoop.addHandler(EventType::log, handlerLogger);

RhombusFactory rhombusFactory;

PentagonFactory pentagonFactory;

HexagonFactory hexagonFactory;

std::string cmd;

int id = 1;

std::thread thHandler(std::ref(eventLoop));

while(std::cin >> cmd){

if(cmd == "quit"){

Event ev = Event(EventType::quit, std::make\_shared<EventData>(), std::make\_shared<EventContext>(), [](auto){});

eventLoop.addEvent(ev);

break;

}else if(cmd == "create"){

std::string type;

std::pair<double, double> center;

std::pair<double, double> point;

std::cin >> type >> center.first >> center.second >> point.first >> point.second;

if(type == "rhombus"){

shapes.push\_back(rhombusFactory.FigureCreate(id, center, point));

}else if(type == "pentagon"){

shapes.push\_back(pentagonFactory.FigureCreate(id, center, point));

}else if(type == "hexagon"){

shapes.push\_back(hexagonFactory.FigureCreate(id, center, point));

}else{

std::cerr << "Error: incorrect figure type" << std::endl;

continue;

}

++id;

std::cout << "Figure added\n";

}

bool printer\_done = false;

bool context\_done = false;

if(shapes.size() == buffer\_size){

Event ev1 = Event(EventType::print,

std::make\_shared<EventDataPrinter>(shapes),

std::make\_shared<EventContextPrinter>(printer\_done),

[](std::shared\_ptr<EventContext> cnt){

if(auto ptr = std::static\_pointer\_cast<EventContextPrinter>(cnt))

ptr->done = true;

});

Event ev2 = Event(EventType::log,

std::make\_shared<EventDataLogger>(shapes),

std::make\_shared<EventContextLogger>(context\_done),

[](std::shared\_ptr<EventContext> cnt){

if(auto ptr = std::static\_pointer\_cast<EventContextPrinter>(cnt))

ptr->done = true;

});

eventLoop.addEvent(ev1);

eventLoop.addEvent(ev2);

while(!printer\_done && !context\_done){

std::cout << '.' << std::flush;

std::this\_thread::sleep\_for(std::chrono::milliseconds(200));

}

shapes.resize(0);

}

}

thHandler.join();

return 0;

}

**event.hpp**

#ifndef TASK\_H

#define TASK\_H

#include <vector>

#include <string>

#include <memory>

#include <functional>

#include <fstream>

#include <iostream>

#include <random>

#include <unordered\_map>

#include "figures.hpp"

std::string strstr\_gen(int len) {

std::random\_device rd;

std::mt19937 randstr\_gen(rd());

std::uniform\_int\_distribution<std::mt19937::result\_type> distribution(97, 122);

std::string res = "";

for (int i = 0; i < len; ++i) {

res += (char)(distribution(randstr\_gen));

}

res += ".fig";

return res;

}

enum class EventType {

log,

print,

quit

};

class EventContext{

public:

virtual ~EventContext() = default;

};

struct EventContextPrinter: public EventContext{

EventContextPrinter(bool &d): done(d) {};

bool &done;

};

struct EventContextLogger: public EventContext{

EventContextLogger(bool &d): done(d) {};

bool &done;

};

class EventData {

public:

virtual ~EventData() = default;

};

struct Event {

Event(EventType tp, std::shared\_ptr<EventData> dt, std::shared\_ptr<EventContext> c, std::function<void(std::shared\_ptr<EventContext>)> f):

type(tp), data(dt), ctx(c), callback(f)

{}

EventType type;

std::shared\_ptr<EventData> data;

std::shared\_ptr<EventContext> ctx;

std::function<void(std::shared\_ptr<EventContext>)> callback;

};

class EventDataPrinter : public EventData {

public:

EventDataPrinter(const std::vector<std::shared\_ptr<IFigure>>& shapes): Shapes(shapes) {

}

const std::vector<std::shared\_ptr<IFigure>> Shapes;

};

class EventDataLogger : public EventData {

public:

EventDataLogger(const std::vector<std::shared\_ptr<IFigure>>& shapes): Shapes(shapes) {

}

const std::vector<std::shared\_ptr<IFigure>> Shapes;

};

class IHandler {

public:

virtual bool event(Event&) = 0;

virtual ~IHandler() = default;

};

class HandlerPrinter : public IHandler {

public:

bool event(Event& event) override {

std::shared\_ptr<EventDataPrinter> data = std::static\_pointer\_cast<EventDataPrinter>(event.data);

if (!data)

return false;

std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));

for (const std::shared\_ptr<IFigure>& shape : data->Shapes) {

shape->Print(std::cout);

}

event.callback(event.ctx);

return true;

}

};

class HandlerLogger : public IHandler {

public:

bool event(Event& event) override {

std::ofstream file;

file.open(strstr\_gen(10), std::ios\_base::out);

std::shared\_ptr<EventDataLogger> data = std::static\_pointer\_cast<EventDataLogger>(event.data);

if (!data) {

file.close();

return false;

}

std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));

for (const std::shared\_ptr<IFigure>& shape : data->Shapes) {

shape->Print(file);

}

event.callback(event.ctx);

file.close();

return true;

}

};

#endif

**event\_loop.hpp**

#ifndef EVENT\_LOOP\_H

#define EVENT\_LOOP\_H

#include <vector>

#include <queue>

#include <utility>

#include <algorithm>

#include <unordered\_map>

#include <memory>

#include <thread>

#include <mutex>

#include "figures.hpp"

#include "event.hpp"

class EventManager {

public:

void subscribe(EventType type, std::shared\_ptr<IHandler>& handler) {

Handlers.emplace(type, handler);

}

void notify(EventType type, Event& event) {

auto its = Handlers.equal\_range(type);

std::for\_each(its.first, its.second, [&event](auto p) {

p.second->event(event);

});

}

private:

std::unordered\_multimap<EventType, std::shared\_ptr<IHandler>> Handlers;

};

class EventLoop {

public:

void addHandler(EventType type, std::shared\_ptr<IHandler>& handler) {

EvManager.subscribe(type, handler);

}

void addEvent(Event& event) {

std::lock\_guard<std::mutex> guard(mut);

Events.push(event);

}

void operator()() {

while (!quit) {

if (!Events.empty()) {

Event ev = Events.front();

Events.pop();

switch(ev.type) {

case EventType::quit:

quit = true;

break;

default:

EvManager.notify(ev.type, ev);

}

} else {

std::this\_thread::sleep\_for(std::chrono::microseconds(10));

}

}

}

private:

bool quit = false;

std::queue<Event> Events;

EventManager EvManager;

std::mutex mut;

};

#endif

**figures.hpp**

#ifndef FIGURES\_H

#define FIGURES\_H

#include <utility>

#include <vector>

#include <iostream>

#include <fstream>

#include <memory>

#include <cmath>

#include <cstdint>

static double GetEps()

{

double epsilon = 1.0;

while (1.0 + 0.5 \* epsilon != 1.0) {

epsilon \*= 0.5;

}

return epsilon;

}

enum FigureType {

RHOMBUS,

PENTAGON,

HEXAGON

};

namespace NGeomAlgo {

double DotProduct(const std::pair<double, double>& v1, const std::pair<double, double>& v2) {

return v1.first \* v2.first + v1.second \* v2.second;

}

double DotDistance(const std::pair<double, double>& p1, const std::pair<double, double>& p2) {

return sqrt(pow((p1.first - p2.first), 2) + pow((p1.second - p2.second), 2));

}

int GetQuarter(std::pair<double, double> center, std::pair<double, double> point) {

if (fabs(point.first) < GetEps() || fabs(point.second) < GetEps())

return 0;

if (point.first > center.first && point.second > center.second)

return 1;

if (point.first < center.first && point.second > center.second)

return 2;

if (point.first < center.first && point.second < center.second)

return 3;

if (point.first > center.first && point.second < center.second)

return 4;

return 0;

}

double CornerCalc(const std::pair<double, double>& v1, const std::pair<double, double>& v2) {

double radius = DotDistance(v1, v2);

std::pair<double, double> u1 = std::make\_pair(v2.first - v1.first, v2.second - v1.second);

std::pair<double, double> u2 = std::make\_pair(radius, 0);

double corner = acos(NGeomAlgo::DotProduct(u1, u2) / (radius \* radius));

double quarter = NGeomAlgo::GetQuarter(v1, v2);

if (quarter == 2)

return M\_PI - corner;

if (quarter == 3)

return M\_PI + corner;

return corner;

}

double NGonSquare(int n, double radius) {

return (double)n / 2 \* pow(radius, 2) \* sin(2 \* M\_PI / (double)n);

}

std::vector<std::pair<double, double>> GetNGonCoors

(int n, const std::pair<double, double>& center, const std::pair<double, double>& point) {

std::vector<std::pair<double, double>> res;

double corner = NGeomAlgo::CornerCalc(center, point);

double r = NGeomAlgo::DotDistance(center, point);

res.push\_back(point);

for (int i = 1; i < n; ++i)

res.emplace\_back(center.first + r \* cos(corner + 2 \* M\_PI \* i / n), center.second + r \* sin(corner + 2 \* M\_PI \* i / n));

return res;

}

}

std::ostream& operator<< (std::ostream& os, const std::pair<double, double>& v) {

os << "<" << v.first << ", " << v.second << ">";

return os;

}

class IFigure {

public:

virtual double Square() 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 uint32\_t Id() const = 0;

virtual ~IFigure() = default;

};

class Rhombus : public IFigure {

public:

Rhombus(): Id\_(0), Center\_(std::make\_pair(0, 0)), FirstPoint\_(std::make\_pair(0, 0))

{

}

Rhombus(uint32\_t id, std::pair<double, double> center, std::pair<double, double> v1):

Id\_(id), Center\_(std::move(center)), FirstPoint\_(std::move(v1))

{

}

uint32\_t Id() const override {

return Id\_;

}

double Square() const override {

return NGeomAlgo::NGonSquare(4, GetRadius());

}

std::pair<double, double> Center() const override {

return Center\_;

}

std::ostream& Print(std::ostream& out) const override {

out << "ID: " << Id\_ << "\n";

out << "Figure: Rhombus" << "\n";

out << "Coords:\n";

std::vector<std::pair<double, double>> coors = NGeomAlgo::GetNGonCoors(4, Center\_, FirstPoint\_);

for (auto& p : coors)

out << p << " ";

out << "\n";

return out;

}

void Serialize(std::ofstream& os) const override {

FigureType type = RHOMBUS;

os.write(reinterpret\_cast<char\*>(&type), sizeof(type));

os.write((char\*)(&Id\_), sizeof(Id\_));

os.write((char\*)(&Center\_.first), sizeof(Center\_.first));

os.write((char\*)(&Center\_.second), sizeof(Center\_.second));

os.write((char\*)(&FirstPoint\_.first), sizeof(FirstPoint\_.first));

os.write((char\*)(&FirstPoint\_.second), sizeof(FirstPoint\_.second));

}

void Deserialize(std::ifstream& is) override {

is.read((char\*)(&Id\_), sizeof(Id\_));

is.read((char\*)(&Center\_.first), sizeof(Center\_.first));

is.read((char\*)(&Center\_.second), sizeof(Center\_.second));

is.read((char\*)(&FirstPoint\_.first), sizeof(FirstPoint\_.first));

is.read((char\*)(&FirstPoint\_.second), sizeof(FirstPoint\_.second));

}

private:

uint32\_t Id\_;

std::pair<double, double> Center\_;

std::pair<double, double> FirstPoint\_;

double GetRadius() const {

return NGeomAlgo::DotDistance(Center\_, FirstPoint\_);

}

};

class Pentagon : public IFigure {

public:

Pentagon(): Id\_(0), Center\_(std::make\_pair(0, 0)), FirstPoint\_(std::make\_pair(0, 0))

{

}

uint32\_t Id() const override {

return Id\_;

}

Pentagon(uint32\_t id, std::pair<double, double> center, std::pair<double, double> v1):

Id\_(id), Center\_(std::move(center)), FirstPoint\_(std::move(v1))

{

}

double Square() const override {

return NGeomAlgo::NGonSquare(5, GetRadius());

}

std::pair<double, double> Center() const override {

return Center\_;

}

std::ostream& Print(std::ostream& out) const override {

out << "ID: " << Id\_ << "\n";

out << "Figure: Pentagon" << "\n";

out << "Coords:\n";

std::vector<std::pair<double, double>> coors = NGeomAlgo::GetNGonCoors(5, Center\_, FirstPoint\_);

for (auto& p : coors)

out << p << " ";

out << "\n";

return out;

}

void Serialize(std::ofstream& os) const override {

FigureType type = PENTAGON;

os.write(reinterpret\_cast<char\*>(&type), sizeof(type));

os.write((char\*)(&Id\_), sizeof(Id\_));

os.write((char\*)(&Center\_.first), sizeof(Center\_.first));

os.write((char\*)(&Center\_.second), sizeof(Center\_.second));

os.write((char\*)(&FirstPoint\_.first), sizeof(FirstPoint\_.first));

os.write((char\*)(&FirstPoint\_.second), sizeof(FirstPoint\_.second));

}

void Deserialize(std::ifstream& is) override {

is.read((char\*)(&Id\_), sizeof(Id\_));

is.read((char\*)(&Center\_.first), sizeof(Center\_.first));

is.read((char\*)(&Center\_.second), sizeof(Center\_.second));

is.read((char\*)(&FirstPoint\_.first), sizeof(FirstPoint\_.first));

is.read((char\*)(&FirstPoint\_.second), sizeof(FirstPoint\_.second));

}

private:

uint32\_t Id\_;

std::pair<double, double> Center\_;

std::pair<double, double> FirstPoint\_;

double GetRadius() const {

return NGeomAlgo::DotDistance(Center\_, FirstPoint\_);

}

};

class Hexagon : public IFigure {

public:

Hexagon(): Id\_(0), Center\_(std::make\_pair(0, 0)), FirstPoint\_(std::make\_pair(0, 0))

{

}

Hexagon(uint32\_t id, std::pair<double, double> center, std::pair<double, double> v1):

Id\_(id), Center\_(std::move(center)), FirstPoint\_(std::move(v1))

{

}

uint32\_t Id() const override {

return Id\_;

}

double Square() const override {

return NGeomAlgo::NGonSquare(6, GetRadius());

}

std::pair<double, double> Center() const override {

return Center\_;

}

std::ostream& Print(std::ostream& out) const override {

out << "ID: " << Id\_ << "\n";

out << "Figure: Hexagon" << "\n";

out << "Coords:\n";

std::vector<std::pair<double, double>> coors = NGeomAlgo::GetNGonCoors(6, Center\_, FirstPoint\_);

for (auto& p : coors)

out << p << " ";

out << "\n";

return out;

}

void Serialize(std::ofstream& os) const override {

FigureType type = HEXAGON;

os.write(reinterpret\_cast<char\*>(&type), sizeof(type));

os.write((char\*)(&Id\_), sizeof(Id\_));

os.write((char\*)(&Center\_.first), sizeof(Center\_.first));

os.write((char\*)(&Center\_.second), sizeof(Center\_.second));

os.write((char\*)(&FirstPoint\_.first), sizeof(FirstPoint\_.first));

os.write((char\*)(&FirstPoint\_.second), sizeof(FirstPoint\_.second));

}

void Deserialize(std::ifstream& is) override {

is.read((char\*)(&Id\_), sizeof(Id\_));

is.read((char\*)(&Center\_.first), sizeof(Center\_.first));

is.read((char\*)(&Center\_.second), sizeof(Center\_.second));

is.read((char\*)(&FirstPoint\_.first), sizeof(FirstPoint\_.first));

is.read((char\*)(&FirstPoint\_.second), sizeof(FirstPoint\_.second));

}

private:

uint32\_t Id\_;

using TVertex = std::pair<double, double>;

TVertex Center\_;

TVertex FirstPoint\_;

double GetRadius() const {

return NGeomAlgo::DotDistance(Center\_, FirstPoint\_);

}

};

class IFactory {

public:

using TVertex = std::pair<double, double>;

virtual std::shared\_ptr<IFigure> FigureCreate(uint32\_t id, TVertex center, TVertex v1) const = 0;

virtual std::shared\_ptr<IFigure> FigureCreate() const = 0;

};

class RhombusFactory : public IFactory {

public:

std::shared\_ptr<IFigure> FigureCreate(uint32\_t id, TVertex center, TVertex v1) const override {

return std::shared\_ptr<IFigure>(new Rhombus(id, center, v1));

}

std::shared\_ptr<IFigure> FigureCreate() const override {

return std::shared\_ptr<IFigure>(new Rhombus());

}

};

class PentagonFactory : public IFactory {

public:

std::shared\_ptr<IFigure> FigureCreate(uint32\_t id, TVertex center, TVertex v1) const override {

return std::shared\_ptr<IFigure>(new Pentagon(id, center, v1));

}

std::shared\_ptr<IFigure> FigureCreate() const override {

return std::shared\_ptr<IFigure>(new Pentagon());

}

};

class HexagonFactory : public IFactory {

public:

std::shared\_ptr<IFigure> FigureCreate(uint32\_t id, TVertex center, TVertex v1) const override {

return std::shared\_ptr<IFigure>(new Hexagon(id, center, v1));

}

std::shared\_ptr<IFigure> FigureCreate() const override {

return std::shared\_ptr<IFigure>(new Hexagon());

}

};

#endif

1. **Вывод**

В программе использовался паттерн Factory. Она позволяет создавать фигуры используя только 2 точки: центр и вершину, после того, как количество добавленных фигур превысило заранее известное максимальное значение, запускается 2 обработчика, один из которых выводит характеристики всех фигур на экран, а другой генерирует файл с расширением .fig и сохраняет их в него.

**Список литературы**

1. Справочник по языку С++ [Электронный ресурс]. URL:

<https://en.cppreference.com/w/> (дата обращения: 20.12.2019).