**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра Вычислительной техники**

отчет

**по лабораторной работе № 2**

**по дисциплине «Алгоритмы и структуры данных»**

# Тема: **«ПОДДЕРЖКА ОБРАБОТКИ ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3311 | Баймухамедов Р. Р. |  |
| Преподаватель | Манирагена В. |  |

Са

Санкт-Петербург

2025

**Введение**

1. **Цель работы**

Разработать механизм обработки исключительных ситуаций в библиотеке работы с графическими фигурами, обеспечивающий:

Контроль корректности параметров фигур.

Визуализацию ошибок на экране.

Минимизацию искажений итоговой картинки при возникновении ошибок.

Демонстрацию перехвата и обработки ошибок разного уровня сложности.

**Постановка задачи**

1. **Реализовать обработку исключений для следующих ситуаций:**

Точка фигуры выходит за границы экрана (PointOutOfBoundsException).

Некорректные параметры фигуры (нулевая длина линии, вырожденный прямоугольник и т. д.) (InvalidShapeParametersException).

Нехватка места при повороте, отражении или перемещении фигуры (PlacementException).

Повторный поворот/отражение уже изменённой фигуры (RotationException).

Неправильное присоединение несимметричной фигуры (AttachmentException).

1. **Обеспечить:**

Удаление некорректной фигуры из списка рисования или её замену на "знак ошибки" (например, красный крест).

Визуализацию состояния фигуры в момент ошибки.

Перехват исключений на уровне main() или shape\_refresh(), а не в месте их возникновения.

1. **Продемонстрировать:**

Обработку как минимум двух типов ошибок разной сложности (например, выход за границы экрана и некорректный поворот).

Минимальные искажения итоговой картинки при ошибках.

**Код добавлений в Shape.h**

**#ifndef SHAPE\_H**

**#define SHAPE\_H**

**#include <list>**

**#include <algorithm>**

**#include <stdexcept>**

**#include <string>**

**#include "screen.h"**

**using std::list;**

**using std::runtime\_error;**

**using std::string;**

**using std::swap;**

**class ShapeException : public runtime\_error {**

**public:**

**ShapeException(const string& msg) : runtime\_error(msg) {}**

**};**

**class PointOutOfBoundsException : public ShapeException {**

**public:**

**PointOutOfBoundsException(const string& msg) : ShapeException(msg) {}**

**};**

**class InvalidShapeParametersException : public ShapeException {**

**public:**

**InvalidShapeParametersException(const string& msg) : ShapeException(msg) {}**

**};**

**class RotationException : public ShapeException {**

**public:**

**RotationException(const string& msg) : ShapeException(msg) {}**

**};**

**class ResizeException : public ShapeException {**

**public:**

**ResizeException(const string& msg) : ShapeException(msg) {}**

**};**

**class PlacementException : public ShapeException {**

**public:**

**PlacementException(const string& msg) : ShapeException(msg) {}**

**};**

**struct shape {**

**static list<shape\*> shapes;**

**shape() { shapes.push\_back(this); }**

**virtual point north() const = 0;**

**virtual point south() const = 0;**

**virtual point east() const = 0;**

**virtual point west() const = 0;**

**virtual point neast() const = 0;**

**virtual point seast() const = 0;**

**virtual point nwest() const = 0;**

**virtual point swest() const = 0;**

**virtual void draw() = 0;**

**virtual void move(int, int) = 0;**

**virtual void resize(double) = 0;**

**virtual void draw\_error\_state() const = 0;**

**virtual ~shape() { shapes.remove(this); }**

**};**

**list<shape\*> shape::shapes;**

**void shape\_refresh();**

**class rotatable : virtual public shape {**

**protected:**

**enum class rotated { left, no, right };**

**rotated state;**

**public:**

**rotatable(rotated r = rotated::no) : state(r) { }**

**virtual void rotate\_left() { state = rotated::left; }**

**virtual void rotate\_right() { state = rotated::right; }**

**};**

**class reflectable : virtual public shape {**

**protected:**

**bool hor, vert;**

**public:**

**reflectable(bool h = false, bool v = false) : hor(h), vert(v) { }**

**virtual void flip\_horizontally() { hor = !hor; }**

**virtual void flip\_vertically() { vert = !vert; }**

**};**

**class line : public shape {**

**protected:**

**point w, e;**

**public:**

**line(point a, point b);**

**line(point a, int L);**

**point north() const override;**

**point south() const override;**

**point east() const override;**

**point west() const override;**

**point neast() const override;**

**point seast() const override;**

**point nwest() const override;**

**point swest() const override;**

**void move(int a, int b) override;**

**void draw() override;**

**void resize(double d) override;**

**void draw\_error\_state() const override;**

**};**

**class rectangle : public rotatable {**

**protected:**

**point sw, ne;**

**public:**

**rectangle(point a, point b);**

**point north() const override;**

**point south() const override;**

**point east() const override;**

**point west() const override;**

**point neast() const override;**

**point seast() const override;**

**point nwest() const override;**

**point swest() const override;**

**void rotate\_right() override;**

**void rotate\_left() override;**

**void move(int a, int b) override;**

**void resize(double d) override;**

**void draw() override;**

**void draw\_error\_state() const override;**

**};**

**class right\_triangle : public rotatable, public reflectable {**

**protected:**

**point sw, ne;**

**public:**

**right\_triangle(point a, point b, point dummy);**

**point north() const override;**

**point south() const override;**

**point east() const override;**

**point west() const override;**

**point neast() const override;**

**point seast() const override;**

**point nwest() const override;**

**point swest() const override;**

**void move(int a, int b) override;**

**void resize(double d) override;**

**void rotate\_right() override;**

**void rotate\_left() override;**

**void flip\_vertically() override;**

**void flip\_horizontally() override;**

**void draw() override;**

**void draw\_error\_state() const override;**

**};**

**class parallelogram : public shape {**

**point p1, p2, p3, p4;**

**public:**

**parallelogram(point a, point b, point c);**

**void draw() override;**

**point north() const override;**

**point south() const override;**

**point east() const override;**

**point west() const override;**

**point neast() const override;**

**point nwest() const override;**

**point swest() const override;**

**point seast() const override;**

**void move(int dx, int dy) override;**

**void resize(double scale) override;**

**void draw\_error\_state() const override;**

**};**

**class hat\_with\_emblem : public rectangle {**

**public:**

**hat\_with\_emblem(point a, point b);**

**void draw() override;**

**void draw\_error\_state() const override;**

**};**

**void up(shape& p, const shape& q);**

**#endif // SHAPE\_H**

**Код добавлений в Shape.срр**

**#include <locale.h>**

**#include <iostream>**

**#include <algorithm>**

**#include <thread>**

**#include <chrono>**

**#include "screen.h"**

**#include "shape.h"**

**char screen[YMAX][XMAX];**

**void screen\_init() {**

**for (auto y = 0; y < YMAX; ++y)**

**for (auto &x : screen[y]) x = '.';**

**}**

**void screen\_destroy() {**

**for (auto y = 0; y < YMAX; ++y)**

**for (auto &x : screen[y]) x = '\*';**

**}**

**bool on\_screen(int a, int b) {**

**return 0 <= a && a < XMAX && 0 <= b && b < YMAX;**

**}**

**void put\_point(int a, int b) {**

**if (on\_screen(a,b)) screen[b][a] = '\*';**

**}**

**void put\_line(int x0, int y0, int x1, int y1) {**

**int dx = 1;**

**int a = x1 - x0; if (a < 0) dx = -1, a = -a;**

**int dy = 1;**

**int b = y1 - y0; if (b < 0) dy = -1, b = -b;**

**int two\_a = 2\*a;**

**int two\_b = 2\*b;**

**int xcrit = -b + two\_a;**

**int eps = 0;**

**for (;;) {**

**put\_point(x0, y0);**

**if (x0 == x1 && y0 == y1) break;**

**if (eps <= xcrit) x0 += dx, eps += two\_b;**

**if (eps >= a || a < b) y0 += dy, eps -= two\_a;**

**}**

**}**

**void screen\_clear() { screen\_init(); }**

**void screen\_refresh() {**

**for (int y = YMAX-1; 0 <= y; --y) {**

**for (auto x : screen[y]) std::cout << x;**

**std::cout << '\n';**

**}**

**}**

**void put\_point(point p) { put\_point(p.x, p.y); }**

**void put\_line(point a, point b) { put\_line(a.x, a.y, b.x, b.y); }**

**void show\_error(const std::string& message, const shape& s) {**

**screen\_clear();**

**s.draw\_error\_state();**

**int len = message.length();**

**int start\_x = (XMAX - len - 4) / 2;**

**if (start\_x < 0) start\_x = 0;**

**for (int x = start\_x; x < start\_x + len + 4 && x < XMAX; x++) {**

**screen[0][x] = '-';**

**screen[2][x] = '-';**

**}**

**screen[1][start\_x] = '|';**

**screen[1][start\_x + len + 3] = '|';**

**for (int i = 0; i < len && start\_x + 2 + i < XMAX; i++) {**

**screen[1][start\_x + 2 + i] = message[i];**

**}**

**screen\_refresh();**

**std::this\_thread::sleep\_for(std::chrono::seconds(2));**

**}**

**void shape\_refresh() {**

**screen\_clear();**

**for (auto p : shape::shapes) {**

**try {**

**p->draw();**

**} catch (const std::exception& e) {**

**std::cerr << "Drawing error: " << e.what() << std::endl;**

**}**

**}**

**screen\_refresh();**

**}**

**// Line implementation**

**void line::draw\_error\_state() const {**

**put\_line(w, e);**

**put\_point(w.x, w.y);**

**put\_point(e.x, e.y);**

**}**

**line::line(point a, point b) : w(a), e(b) {**

**if (!on\_screen(a.x, a.y) || !on\_screen(b.x, b.y)) {**

**show\_error("Line points out of bounds", \*this);**

**throw PointOutOfBoundsException(**

**"Line points (" + std::to\_string(a.x) + "," + std::to\_string(a.y) + ") and (" +**

**std::to\_string(b.x) + "," + std::to\_string(b.y) + ") out of bounds"**

**);**

**}**

**}**

**line::line(point a, int L) : w(a), e(point(a.x + L - 1, a.y)) {**

**if (!on\_screen(w.x, w.y) || !on\_screen(e.x, e.y)) {**

**show\_error("Line points out of bounds", \*this);**

**throw PointOutOfBoundsException(**

**"Line points (" + std::to\_string(w.x) + "," + std::to\_string(w.y) + ") and (" +**

**std::to\_string(e.x) + "," + std::to\_string(e.y) + ") out of bounds"**

**);**

**}**

**}**

**point line::north() const { return point((w.x+e.x)/2, e.y<w.y? w.y : e.y); }**

**point line::south() const { return point((w.x+e.x)/2, e.y<w.y? e.y : w.y); }**

**point line::east() const { return point(e.x<w.x? w.x : e.x, (w.y+e.y)/2); }**

**point line::west() const { return point(e.x<w.x? e.x : w.x, (w.y+e.y)/2); }**

**point line::neast() const { return point(w.x<e.x? e.x : w.x, e.y<w.y? w.y : e.y); }**

**point line::seast() const { return point(w.x<e.x? e.x : w.x, e.y<w.y? e.y : w.y); }**

**point line::nwest() const { return point(w.x<e.x? w.x : e.x, e.y<w.y? w.y : e.y); }**

**point line::swest() const { return point(w.x<e.x? w.x : e.x, e.y<w.y? e.y : w.y); }**

**void line::move(int a, int b) {**

**point new\_w(w.x + a, w.y + b);**

**point new\_e(e.x + a, e.y + b);**

**if (!on\_screen(new\_w.x, new\_w.y) || !on\_screen(new\_e.x, new\_e.y)) {**

**show\_error("Line would be out of bounds", \*this);**

**throw PointOutOfBoundsException(**

**"Line would be out of bounds after move to (" +**

**std::to\_string(new\_w.x) + "," + std::to\_string(new\_w.y) + ") and (" +**

**std::to\_string(new\_e.x) + "," + std::to\_string(new\_e.y) + ")"**

**);**

**}**

**w = new\_w;**

**e = new\_e;**

**}**

**void line::draw() { put\_line(w, e); }**

**void line::resize(double d) {**

**if (d <= 0) {**

**show\_error("Invalid resize factor", \*this);**

**throw ResizeException("Invalid resize factor: " + std::to\_string(d));**

**}**

**point new\_e(w.x + (e.x - w.x) \* d, w.y + (e.y - w.y) \* d);**

**if (!on\_screen(new\_e.x, new\_e.y)) {**

**show\_error("Line would be out of bounds", \*this);**

**throw PlacementException(**

**"Line would be out of bounds after resize to (" +**

**std::to\_string(new\_e.x) + "," + std::to\_string(new\_e.y) + ")"**

**);**

**}**

**e = new\_e;**

**}**

**// Rectangle implementation**

**void rectangle::draw\_error\_state() const {**

**put\_line(nwest(), ne);**

**put\_line(ne, seast());**

**put\_line(seast(), sw);**

**put\_line(sw, nwest());**

**put\_point(sw.x, sw.y);**

**put\_point(ne.x, ne.y);**

**}**

**rectangle::rectangle(point a, point b) : sw(a), ne(b) {**

**if (!on\_screen(a.x, a.y) || !on\_screen(b.x, b.y)) {**

**show\_error("Rectangle points out of bounds", \*this);**

**throw PointOutOfBoundsException(**

**"Rectangle points (" + std::to\_string(a.x) + "," + std::to\_string(a.y) + ") and (" +**

**std::to\_string(b.x) + "," + std::to\_string(b.y) + ") out of bounds"**

**);**

**}**

**if (a.x == b.x || a.y == b.y) {**

**show\_error("Invalid rectangle dimensions", \*this);**

**throw InvalidShapeParametersException(**

**"Rectangle has zero width or height between points (" +**

**std::to\_string(a.x) + "," + std::to\_string(a.y) + ") and (" +**

**std::to\_string(b.x) + "," + std::to\_string(b.y) + ")"**

**);**

**}**

**}**

**point rectangle::north() const { return point((sw.x + ne.x)/2, ne.y); }**

**point rectangle::south() const { return point((sw.x + ne.x)/2, sw.y); }**

**point rectangle::east() const { return point(ne.x, (sw.y + ne.y)/2); }**

**point rectangle::west() const { return point(sw.x, (sw.y + ne.y)/2); }**

**point rectangle::neast() const { return ne; }**

**point rectangle::seast() const { return point(ne.x, sw.y); }**

**point rectangle::nwest() const { return point(sw.x, ne.y); }**

**point rectangle::swest() const { return sw; }**

**void rectangle::rotate\_right() {**

**if (state == rotated::right) {**

**show\_error("Already rotated right", \*this);**

**throw RotationException("Rectangle is already rotated right");**

**}**

**int w = ne.x - sw.x, h = ne.y - sw.y;**

**point new\_sw(ne.x - h, sw.y);**

**point new\_ne(ne.x, sw.y + w);**

**if (!on\_screen(new\_sw.x, new\_sw.y) || !on\_screen(new\_ne.x, new\_ne.y)) {**

**show\_error("Rotation would go out of bounds", \*this);**

**throw PlacementException("Rectangle would be out of bounds after rotation");**

**}**

**sw = new\_sw;**

**ne = new\_ne;**

**state = rotated::right;**

**}**

**void rectangle::rotate\_left() {**

**if (state == rotated::left) {**

**show\_error("Already rotated left", \*this);**

**throw RotationException("Rectangle is already rotated left");**

**}**

**int w = ne.x - sw.x, h = ne.y - sw.y;**

**point new\_sw(sw.x, sw.y + w);**

**point new\_ne(sw.x + h, ne.y);**

**if (!on\_screen(new\_sw.x, new\_sw.y) || !on\_screen(new\_ne.x, new\_ne.y)) {**

**show\_error("Rotation would go out of bounds", \*this);**

**throw PlacementException("Rectangle would be out of bounds after rotation");**

**}**

**sw = new\_sw;**

**ne = new\_ne;**

**state = rotated::left;**

**}**

**void rectangle::move(int a, int b) {**

**point new\_sw(sw.x + a, sw.y + b);**

**point new\_ne(ne.x + a, ne.y + b);**

**if (!on\_screen(new\_sw.x, new\_sw.y) || !on\_screen(new\_ne.x, new\_ne.y)) {**

**show\_error("Move would go out of bounds", \*this);**

**throw PointOutOfBoundsException(**

**"Rectangle would be out of bounds after move to (" +**

**std::to\_string(new\_sw.x) + "," + std::to\_string(new\_sw.y) + ") and (" +**

**std::to\_string(new\_ne.x) + "," + std::to\_string(new\_ne.y) + ")"**

**);**

**}**

**sw = new\_sw;**

**ne = new\_ne;**

**}**

**void rectangle::resize(double d) {**

**if (d <= 0) {**

**show\_error("Invalid resize factor", \*this);**

**throw ResizeException("Invalid resize factor: " + std::to\_string(d));**

**}**

**point new\_ne(sw.x + (ne.x - sw.x) \* d, sw.y + (ne.y - sw.y) \* d);**

**if (!on\_screen(new\_ne.x, new\_ne.y)) {**

**show\_error("Resize would go out of bounds", \*this);**

**throw PlacementException(**

**"Rectangle would be out of bounds after resize to (" +**

**std::to\_string(new\_ne.x) + "," + std::to\_string(new\_ne.y) + ")"**

**);**

**}**

**ne = new\_ne;**

**}**

**void rectangle::draw() {**

**put\_line(nwest(), ne);**

**put\_line(ne, seast());**

**put\_line(seast(), sw);**

**put\_line(sw, nwest());**

**}**

**// RightTriangle implementation**

**void right\_triangle::draw\_error\_state() const {**

**put\_line(swest(), nwest());**

**put\_line(swest(), seast());**

**put\_line(nwest(), seast());**

**put\_point(sw.x, sw.y);**

**put\_point(ne.x, ne.y);**

**}**

**right\_triangle::right\_triangle(point a, point b, point dummy) : sw(a), ne(b) {**

**if (!on\_screen(a.x, a.y) || !on\_screen(b.x, b.y)) {**

**show\_error("Triangle points out of bounds", \*this);**

**throw PointOutOfBoundsException(**

**"Triangle points (" + std::to\_string(a.x) + "," + std::to\_string(a.y) + ") and (" +**

**std::to\_string(b.x) + "," + std::to\_string(b.y) + ") out of bounds"**

**);**

**}**

**if (a.x == b.x || a.y == b.y) {**

**show\_error("Invalid triangle dimensions", \*this);**

**throw InvalidShapeParametersException(**

**"Triangle has zero width or height between points (" +**

**std::to\_string(a.x) + "," + std::to\_string(a.y) + ") and (" +**

**std::to\_string(b.x) + "," + std::to\_string(b.y) + ")"**

**);**

**}**

**}**

**point right\_triangle::north() const { return point((sw.x + ne.x)/2, ne.y); }**

**point right\_triangle::south() const { return point((sw.x + ne.x)/2, sw.y); }**

**point right\_triangle::east() const { return point(ne.x, (sw.y + ne.y)/2); }**

**point right\_triangle::west() const { return point(sw.x, (sw.y + ne.y)/2); }**

**point right\_triangle::neast() const { return ne; }**

**point right\_triangle::seast() const { return point(ne.x, sw.y); }**

**point right\_triangle::nwest() const { return point(sw.x, ne.y); }**

**point right\_triangle::swest() const { return sw; }**

**void right\_triangle::move(int a, int b) {**

**point new\_sw(sw.x + a, sw.y + b);**

**point new\_ne(ne.x + a, ne.y + b);**

**if (!on\_screen(new\_sw.x, new\_sw.y) || !on\_screen(new\_ne.x, new\_ne.y)) {**

**show\_error("Move would go out of bounds", \*this);**

**throw PointOutOfBoundsException(**

**"Triangle would be out of bounds after move to (" +**

**std::to\_string(new\_sw.x) + "," + std::to\_string(new\_sw.y) + ") and (" +**

**std::to\_string(new\_ne.x) + "," + std::to\_string(new\_ne.y) + ")"**

**);**

**}**

**sw = new\_sw;**

**ne = new\_ne;**

**}**

**void right\_triangle::resize(double d) {**

**if (d <= 0) {**

**show\_error("Invalid resize factor", \*this);**

**throw ResizeException("Invalid resize factor: " + std::to\_string(d));**

**}**

**point new\_ne(sw.x + (ne.x - sw.x) \* d, sw.y + (ne.y - sw.y) \* d);**

**if (!on\_screen(new\_ne.x, new\_ne.y)) {**

**show\_error("Resize would go out of bounds", \*this);**

**throw PlacementException(**

**"Triangle would be out of bounds after resize to (" +**

**std::to\_string(new\_ne.x) + "," + std::to\_string(new\_ne.y) + ")"**

**);**

**}**

**ne = new\_ne;**

**}**

**void right\_triangle::rotate\_right() {**

**if (state == rotated::right) {**

**show\_error("Already rotated right", \*this);**

**throw RotationException("Triangle is already rotated right");**

**}**

**int dx = ne.x - sw.x;**

**int dy = ne.y - sw.y;**

**point new\_sw(sw.x, sw.y + dx);**

**point new\_ne(sw.x + dy, sw.y);**

**if (!on\_screen(new\_sw.x, new\_sw.y) || !on\_screen(new\_ne.x, new\_ne.y)) {**

**show\_error("Rotation would go out of bounds", \*this);**

**throw PlacementException("Triangle would be out of bounds after rotation");**

**}**

**sw = new\_sw;**

**ne = new\_ne;**

**state = rotated::right;**

**}**

**void right\_triangle::rotate\_left() {**

**if (state == rotated::left) {**

**show\_error("Already rotated left", \*this);**

**throw RotationException("Triangle is already rotated left");**

**}**

**int dx = ne.x - sw.x;**

**int dy = ne.y - sw.y;**

**point new\_sw(sw.x - dy, sw.y);**

**point new\_ne(ne.x, ne.y + dx);**

**if (!on\_screen(new\_sw.x, new\_sw.y) || !on\_screen(new\_ne.x, new\_ne.y)) {**

**show\_error("Rotation would go out of bounds", \*this);**

**throw PlacementException("Triangle would be out of bounds after rotation");**

**}**

**sw = new\_sw;**

**ne = new\_ne;**

**state = rotated::left;**

**}**

**void right\_triangle::flip\_vertically() {**

**int cx = (sw.x + ne.x)/2;**

**sw.x = 2\*cx - sw.x;**

**ne.x = 2\*cx - ne.x;**

**if (sw.x < ne.x) swap(ne.x, sw.x);**

**vert = !vert;**

**}**

**void right\_triangle::flip\_horizontally() {**

**point old\_sw = sw;**

**point old\_ne = ne;**

**sw = point(old\_sw.x, old\_ne.y);**

**ne = point(old\_ne.x, old\_sw.y);**

**if (sw.x < ne.x) swap(ne.x, sw.x);**

**hor = !hor;**

**}**

**void right\_triangle::draw() {**

**put\_line(swest(), nwest());**

**put\_line(swest(), seast());**

**put\_line(nwest(), seast());**

**}**

**// Parallelogram implementation**

**void parallelogram::draw\_error\_state() const {**

**put\_line(p1, p2);**

**put\_line(p2, p3);**

**put\_line(p3, p4);**

**put\_line(p4, p1);**

**put\_point(p1.x, p1.y);**

**put\_point(p2.x, p2.y);**

**put\_point(p3.x, p3.y);**

**put\_point(p4.x, p4.y);**

**}**

**parallelogram::parallelogram(point a, point b, point c) : p1(a), p2(b), p3(c) {**

**if (!on\_screen(a.x, a.y) || !on\_screen(b.x, b.y) || !on\_screen(c.x, c.y)) {**

**show\_error("Parallelogram points out of bounds", \*this);**

**throw PointOutOfBoundsException("Parallelogram points are out of bounds");**

**}**

**p4 = point(p1.x + (p3.x - p2.x), p1.y + (p3.y - p2.y));**

**if (!on\_screen(p4.x, p4.y)) {**

**show\_error("Parallelogram would be out of bounds", \*this);**

**throw PointOutOfBoundsException("Parallelogram would be out of bounds");**

**}**

**if ((p1.x == p2.x && p1.y == p2.y) ||**

**(p1.x == p3.x && p1.y == p3.y) ||**

**(p2.x == p3.x && p2.y == p3.y)) {**

**show\_error("Degenerate parallelogram", \*this);**

**throw InvalidShapeParametersException("Degenerate parallelogram");**

**}**

**}**

**void parallelogram::draw() {**

**put\_line(p1, p2);**

**put\_line(p2, p3);**

**put\_line(p3, p4);**

**put\_line(p4, p1);**

**}**

**point parallelogram::north() const {**

**int y = std::max({p1.y, p2.y, p3.y, p4.y});**

**int x = (p1.x + p2.x + p3.x + p4.x)/4;**

**return point(x, y);**

**}**

**point parallelogram::south() const {**

**int y = std::min({p1.y, p2.y, p3.y, p4.y});**

**int x = (p1.x + p2.x + p3.x + p4.x)/4;**

**return point(x, y);**

**}**

**point parallelogram::east() const {**

**int x = std::max({p1.x, p2.x, p3.x, p4.x});**

**int y = (p1.y + p2.y + p3.y + p4.y)/4;**

**return point(x, y);**

**}**

**point parallelogram::west() const {**

**int x = std::min({p1.x, p2.x, p3.x, p4.x});**

**int y = (p1.y + p2.y + p3.y + p4.y)/4;**

**return point(x, y);**

**}**

**point parallelogram::neast() const {**

**return point(std::max({p1.x, p2.x, p3.x, p4.x}),**

**std::max({p1.y, p2.y, p3.y, p4.y}));**

**}**

**point parallelogram::nwest() const {**

**return point(std::min({p1.x, p2.x, p3.x, p4.x}),**

**std::max({p1.y, p2.y, p3.y, p4.y}));**

**}**

**point parallelogram::swest() const {**

**return point(std::min({p1.x, p2.x, p3.x, p4.x}),**

**std::min({p1.y, p2.y, p3.y, p4.y}));**

**}**

**point parallelogram::seast() const {**

**return point(std::max({p1.x, p2.x, p3.x, p4.x}),**

**std::min({p1.y, p2.y, p3.y, p4.y}));**

**}**

**void parallelogram::move(int dx, int dy) {**

**point new\_p1(p1.x + dx, p1.y + dy);**

**point new\_p2(p2.x + dx, p2.y + dy);**

**point new\_p3(p3.x + dx, p3.y + dy);**

**point new\_p4(p4.x + dx, p4.y + dy);**

**if (!on\_screen(new\_p1.x, new\_p1.y) ||**

**!on\_screen(new\_p2.x, new\_p2.y) ||**

**!on\_screen(new\_p3.x, new\_p3.y) ||**

**!on\_screen(new\_p4.x, new\_p4.y)) {**

**show\_error("Move would go out of bounds", \*this);**

**throw PointOutOfBoundsException("Parallelogram would be out of bounds after move");**

**}**

**p1 = new\_p1;**

**p2 = new\_p2;**

**p3 = new\_p3;**

**p4 = new\_p4;**

**}**

**void parallelogram::resize(double scale) {**

**if (scale <= 0) {**

**show\_error("Invalid resize factor", \*this);**

**throw ResizeException("Invalid resize factor: " + std::to\_string(scale));**

**}**

**point center((p1.x + p2.x + p3.x + p4.x)/4,**

**(p1.y + p2.y + p3.y + p4.y)/4);**

**point new\_p1(center.x + (p1.x - center.x) \* scale, center.y + (p1.y - center.y) \* scale);**

**point new\_p2(center.x + (p2.x - center.x) \* scale, center.y + (p2.y - center.y) \* scale);**

**point new\_p3(center.x + (p3.x - center.x) \* scale, center.y + (p3.y - center.y) \* scale);**

**point new\_p4(center.x + (p4.x - center.x) \* scale, center.y + (p4.y - center.y) \* scale);**

**if (!on\_screen(new\_p1.x, new\_p1.y) || !on\_screen(new\_p2.x, new\_p2.y) ||**

**!on\_screen(new\_p3.x, new\_p3.y) || !on\_screen(new\_p4.x, new\_p4.y)) {**

**show\_error("Resize would go out of bounds", \*this);**

**throw PlacementException("Parallelogram would be out of bounds after resize");**

**}**

**p1 = new\_p1;**

**p2 = new\_p2;**

**p3 = new\_p3;**

**p4 = new\_p4;**

**}**

**// HatWithEmblem implementation**

**hat\_with\_emblem::hat\_with\_emblem(point a, point b) : rectangle(a, b) {}**

**void hat\_with\_emblem::draw\_error\_state() const {**

**rectangle::draw\_error\_state();**

**int centerX = (sw.x + ne.x)/2;**

**int centerY = (sw.y + ne.y)/2;**

**int length = (ne.x - sw.x)/3;**

**try {**

**put\_line(centerX - length, centerY - length, centerX + length, centerY + length);**

**put\_line(centerX + length, centerY - length, centerX - length, centerY + length);**

**} catch (...) {**

**// Ignore errors in emblem drawing for error state**

**}**

**}**

**void hat\_with\_emblem::draw() {**

**rectangle::draw();**

**int centerX = (sw.x + ne.x)/2;**

**int centerY = (sw.y + ne.y)/2;**

**int length = (ne.x - sw.x)/3;**

**try {**

**put\_line(centerX - length, centerY - length, centerX + length, centerY + length);**

**put\_line(centerX + length, centerY - length, centerX - length, centerY + length);**

**} catch (const PointOutOfBoundsException&) {**

**// Если крест не помещается, рисуем только прямоугольник**

**}**

**}**

**void up(shape& p, const shape& q) {**

**point n = q.north();**

**point s = p.south();**

**try {**

**p.move(0, n.y - s.y);**

**} catch (const PointOutOfBoundsException& e) {**

**p.draw\_error\_state();**

**throw PlacementException("Cannot place shape: " + string(e.what()));**

**}**

**}**

**int main() {**

**setlocale(LC\_ALL, "Rus");**

**screen\_init();**

**try {**

**// 1. Демонстрация обработки ошибки создания фигуры**

**try {**

**rectangle bad\_rect(point(-10, -10), point(300, 300)); // Вызовет исключение**

**} catch (const PointOutOfBoundsException& e) {**

**std::cerr << "Error 1: " << e.what() << ". Creating replacement...\n";**

**rectangle good\_rect(point(10, 10), point(20, 20));**

**good\_rect.draw();**

**}**

**// 2. Демонстрация ошибки поворота**

**right\_triangle tri(point(30, 10), point(40, 20), point(0,0));**

**try {**

**tri.rotate\_right();**

**tri.rotate\_right(); // Второй поворот вызовет исключение**

**} catch (const RotationException& e) {**

**std::cerr << "Error 2: " << e.what() << "\n";**

**}**

**// 3. Демонстрация ошибки изменения размера**

**parallelogram para(point(50, 10), point(60, 10), point(55, 20));**

**try {**

**para.resize(-1); // Недопустимый коэффициент**

**} catch (const ResizeException& e) {**

**std::cerr << "Error 3: " << e.what() << "\n";**

**para.resize(0.5); // Пробуем допустимый размер**

**}**

**// 4. Демонстрация ошибки размещения**

**line ln(point(70, 10), point(80, 10));**

**try {**

**ln.move(100, 100); // Выведет за границы экрана**

**} catch (const PointOutOfBoundsException& e) {**

**std::cerr << "Error 4: " << e.what() << "\n";**

**ln.move(10, 10); // Допустимое перемещение**

**}**

**shape\_refresh();**

**std::cout << "=== Result with error handling ===\n";**

**std::cin.get();**

**} catch (const std::exception& e) {**

**std::cerr << "Fatal error: " << e.what() << "\n";**

**// Рисуем знак ошибки**

**screen\_clear();**

**rectangle error(point(XMAX/2-5, YMAX/2-2), point(XMAX/2+5, YMAX/2+2));**

**line cross1(point(XMAX/2-4, YMAX/2-1), point(XMAX/2+4, YMAX/2+1));**

**line cross2(point(XMAX/2-4, YMAX/2+1), point(XMAX/2+4, YMAX/2-1));**

**error.draw();**

**cross1.draw();**

**cross2.draw();**

**screen\_refresh();**

**std::cin.get();**

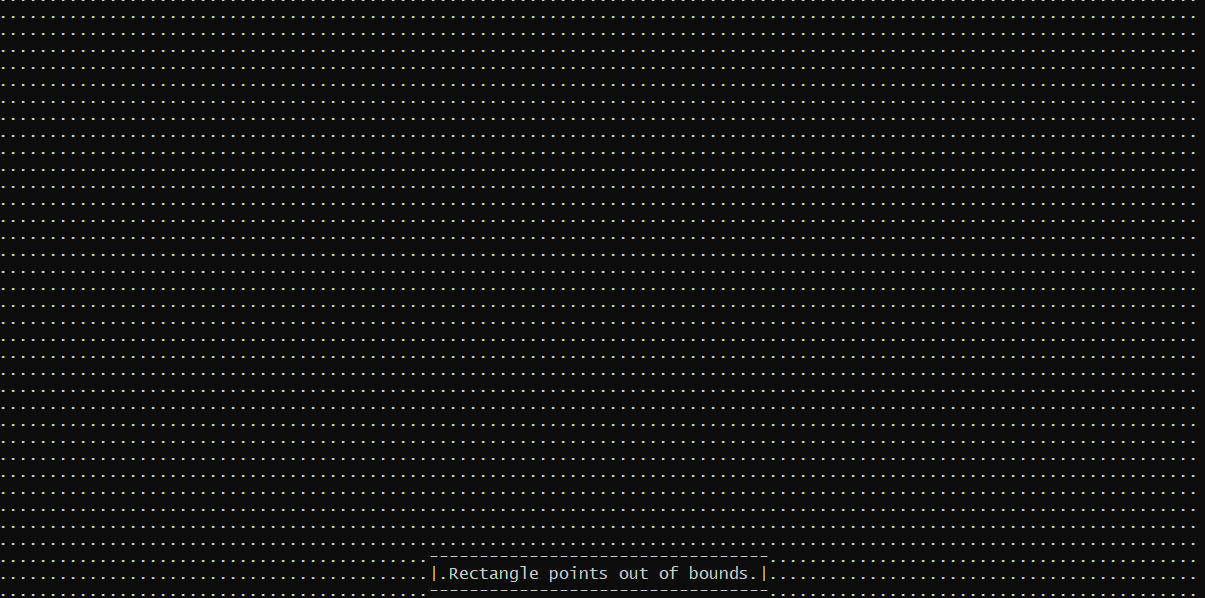
**}**

**screen\_destroy();**

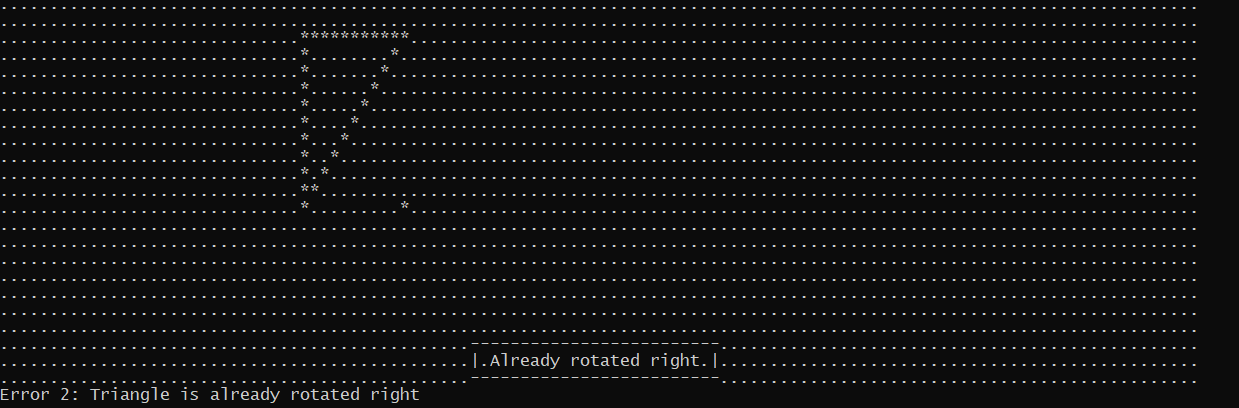
**return 0;**

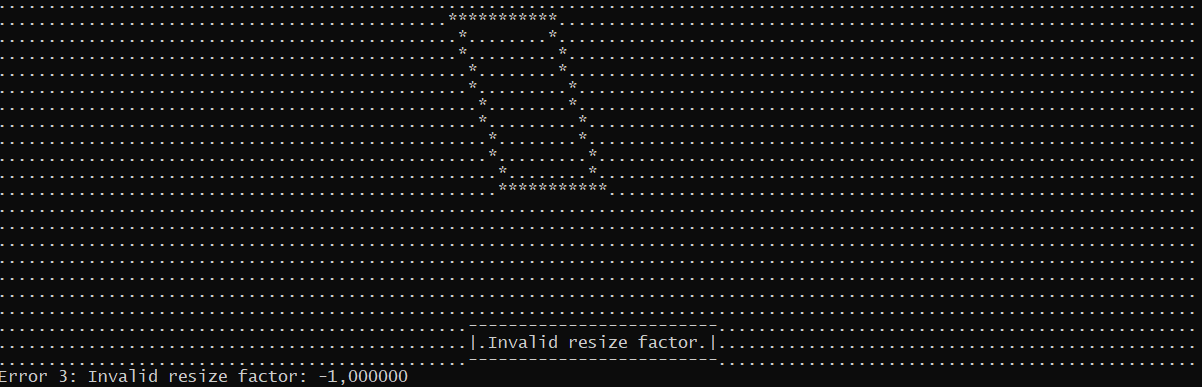
**}**

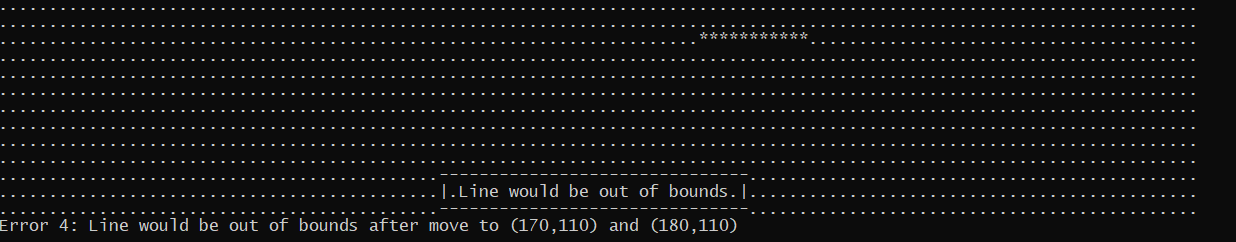
**Пример:**

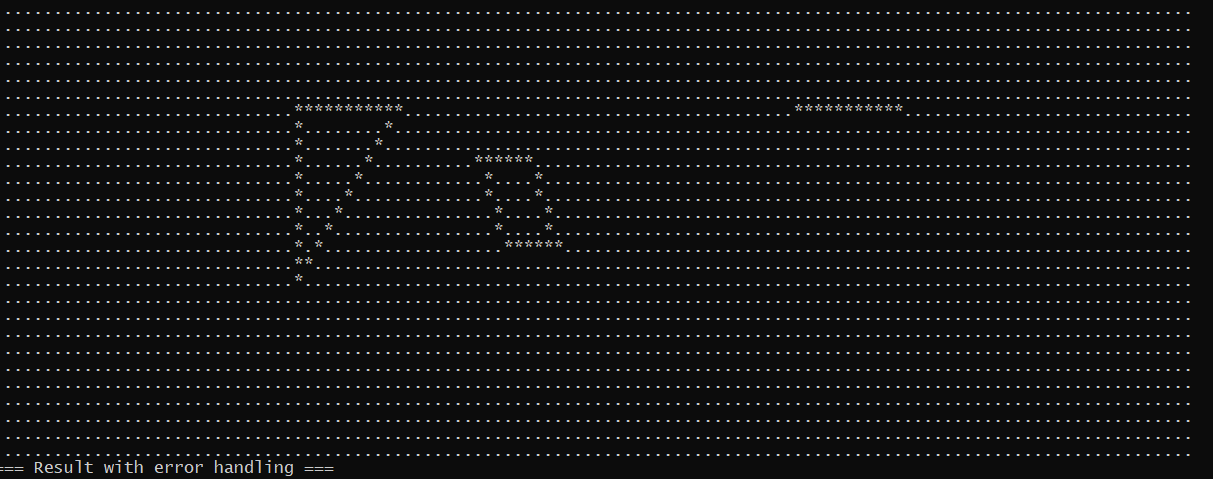












**Контрольные вопросы**

1. **1. Что такое исключительная ситуация?**

Это нештатное состояние программы, которое может привести к её аварийному завершению (например, деление на ноль, выход за границы массива).

**2. Как выявить исключительную ситуацию?**

С помощью:

* Проверок условий (if).
* Механизма исключений (try/catch).
* Встроенных исключений языка (например, std::out\_of\_range).

**3. Что делать при выявлении?**

* Обработать локально (исправить данные).
* Передать ошибку выше по стеку вызовов (throw).
* Завершить работу с сообщением (для критических ошибок).

1. **4. Можно ли передать информацию обработчику?**

Да, через параметры исключения:

throw std::runtime\_error("Файл не найден: " + filename);

1. **5. Можно ли обработать неизвестную ситуацию?**

Да, с помощью catch (...) (перехватывает любые исключения).

1. **6. Можно ли сделать обработчик пустым?**

Да, но это плохо

**7. Что делать, если обработчику не хватает данных?**

* Передать исключение выше (throw; внутри catch).
* Добавить контекст в сообщение об ошибке.

1. **8. Порядок нескольких обработчиков?**

От более конкретных к более общим:

catch (const std::invalid\_argument& e) { ... } // Сначала частные случаи

catch (const std::exception& e) { ... } // Затем общие

catch (...) { ... } // В конце — всё остальное

1. **9. Можно ли перехватывать несколько исключений одним обработчиком?**

Да, если они имеют общего предка

1. **10. Как работают обработчики, если ошибки не было?**

Блок catch игнорируется, программа выполняется дальше.

**11. Как размещать блоки контроля для безопасности?**

* Обрамлять опасные операции try/catch.
* Проверять входные данные до их использования.
* Использовать RAII (умные указатели, контейнеры).

**12. Можно ли продолжить выполнение после исправления ошибки?**

Да, если:

* Ошибка не критична (например, повторная попытка чтения файла).
* Исправлены данные, вызвавшие ошибку.

**Заключение**

В ходе выполнения работы была разработана библиотека для работы с графическими фигурами, дополненная механизмом обработки исключительных ситуаций. Реализована система контроля ошибок, охватывающая все ключевые аспекты работы с фигурами:

Контроль корректности параметров

* Проверка выхода точек за границы экрана.
* Обнаружение некорректных параметров фигур (нулевые размеры, вырожденные случаи).

Обработка динамических операций

* Поворот, отражение и перемещение с проверкой на допустимость.
* Защита от повторных трансформаций (например, двойной поворот).

 Визуализация ошибок

* Фигуры в ошибочном состоянии выделяются на экране (например, помечаются крестиками).
* Сообщения об ошибках выводятся в читаемом формате без перекрытия основной картинки.

Минимизация искажений

* Некорректные фигуры либо удаляются из списка рисования, либо заменяются знаком ошибки.
* Программа продолжает работу даже при возникновении исключений.
* Ключевые достижения

1. Гибкость
   * Система исключений легко расширяется для новых типов ошибок.
2. Наглядность
   * Ошибки визуализируются, что упрощает отладку.
3. Отказоустойчивость
   * Итоговое изображение остаётся читаемым даже при наличии ошибок.

**Литература:**

[**C++ | Обработка исключений - Введение в обработку исключений в C++,**](https://metanit.com/cpp/tutorial/6.1.php) **включая использование конструкций try, catch и throw 1.**

[**Обработка исключений (throw, try и catch) в С++ / Ravesli**](https://ravesli.com/urok-182-obrabotka-isklyuchenij/) **- Подробное объяснение механизма исключений в C++ с примерами использования 2.**

[**Обработка исключительных ситуаций**](https://prog-cpp.ru/cpp-exception/) **- Обзор типов исключений и их обработки в C++ 3.**