МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное учреждения высшего образования

«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт компьютерных технологий и информационной безопасности

Кафедра математического обеспечения и применения ЭВМ

**ЛАБОРАТОРНАЯ РАБОТА № 3**

по дисциплине

**«Динамический выбор типа объектов»**

на тему:

**«Наследование в С++»**

*Вариант № 2*

Выполнил:

Студент группы

КТбо1-8

Макаров С. В.

Проверил:

Тарасов С. А.

Оценка

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2020 г.

Таганрог 2020

# **1 ФОРМУЛИРОВКА ЗАДАНИЯ**

Необходимо создать абстрактный класс Shape для создания от него объектов-фигур. Определить методы вычисления центра, площади, поворота фигуры. Создать класс операций над фигурами, в нем определить методы определения пересечения и включения фигур.

# **2 СПЕЦИФИКАЦИЯ КЛАССОВ**

class Shape

{

public:

Shape();

Shape(Point\* points, int length, const std::string& \_name);

Shape(const Shape& other);

Shape& operator=(const Shape& other);

Point& operator[](int index);

virtual ~Shape();

virtual float FindArea() const = 0;

virtual Point FindCenter() const = 0;

virtual bool IsExisting() const = 0;

void Rotate(int degrees);

void Move(const Point& offset);

Point\* GetVerticesCopy() const;

std::string GetName() const;

int GetCount() const;

private:

std::string name;

void \_CopyArray(Point\* points, int length);

protected:

Point\* vertices;

int count;

void \_ConstructShape(Point\* points, int length, const std::string& \_name);

};

class Triangle : public Shape

{

public:

Triangle() = delete;

Triangle(const Point& a, const Point& b, const Point& c);

float FindArea() const override;

Point FindCenter() const override;

bool IsExisting() const override;

private:

static bool \_IsTriangleExisting(float A, float B, float C);

};

class Rect : public Shape

{

public:

Rect() = delete;

Rect(const Point& point, float \_height, float \_width);

float FindArea() const override;

Point FindCenter() const override;

bool IsExisting() const override;

float GetHeight() const;

float GetWidth() const;

void SetHeight(float value);

void SetWidth(float value);

private:

float height;

float width;

static bool \_IsRectExisting(float width, float height);

};

class ConsoleInteractor

{

public:

void Start();

};

// class with only static methods

class GeometryMath

{

public:

GeometryMath() = delete;

GeometryMath(const GeometryMath& other) = delete;

GeometryMath(GeometryMath&& rvalue) = delete;

static constexpr float pi = 3.1415926535897932385f;

static constexpr float epsilon = 0.00001f;

static float FindMagnitude(const Line& line);

static float CalculateScalar(const Line& v1, const Line& v2);

static bool IsIntersected(const Shape& first, const Shape& second);

static bool IsIntersected(const Line& first, const Line& second);

static bool IsIncluding(const Shape& dest, const Shape& incl);

static bool IsIncluding(const Point& p, const Shape& figure);

static float DegreeToRadian(int degree);

private:

static void \_ValidateShape(const Shape& shape);

static bool \_IsIncludingDontValidate(const Point& p, const Shape& figure);

};

struct Line

{

Point begin;

Point end;

Line();

Line(Point from, Point to);

};

struct Point

{

float x;

float y;

Point()

: x(0), y(0)

{}

Point(float \_x, float \_y)

: x(\_x), y(\_y)

{}

bool operator==(const Point& other);

Point& operator+=(const Point& offset);

};

class ShapeException : public std::exception

{

public:

ShapeException() = delete;

ShapeException(const Shape& shape);

const char\* what() const noexcept;

private:

std::string what\_arg;

};

class TriangleFactory

{

public:

static Shape\* CreateShape(const Point& a, const Point& b, const Point& c);

};

class RectFactory

{

public:

static Shape\* CreateShape(const Point& a, float width, float height);

};

# **3 ИСПОЛЬЗУЕМЫЕ МАТЕМАТИЧЕСКИЕ ЗАВИСИМОСТИ И АЛГОРИТМЫ**

В данной работе использовались следующие зависимости:

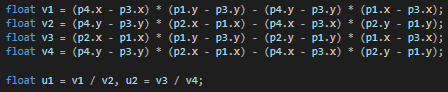
Вычисление длины вектора по формуле:

Вычисление угла между двумя векторами по формуле:

Принадлежность точки фигуре вычисляется по алгоритму суммы углов: из данной точки проводятся отрезки во все вершины фигуры, сумма углов между всеми соседними отрезками должна быть равна 2\*Pi.

Включение одной фигуры в другой определяется с помощью последовательной проверки всех точек на принадлежность.

Пересечение двух отрезков определяется по формуле, показанной на рисунке 1, которая была выведена из уравнения прямых.



Где, p1 и p2 – точки первого вектора, остальные – второго. Если коэффициенты u1 и u2 лежат на промежутке от 0 до 1 включительно, то отрезки пересекаются.

Для проверки, пересекаются ли фигуры, выполняется проверка на пересечения всех ребер со всеми возможными.

Для поворота фигур использовалась матрица поворота, которая поворачивает точку на заданный угол в радианах.

# **4 ДИАГРАММА КЛАССОВ**

Uml диаграмма классов изображена на рисунке 1.

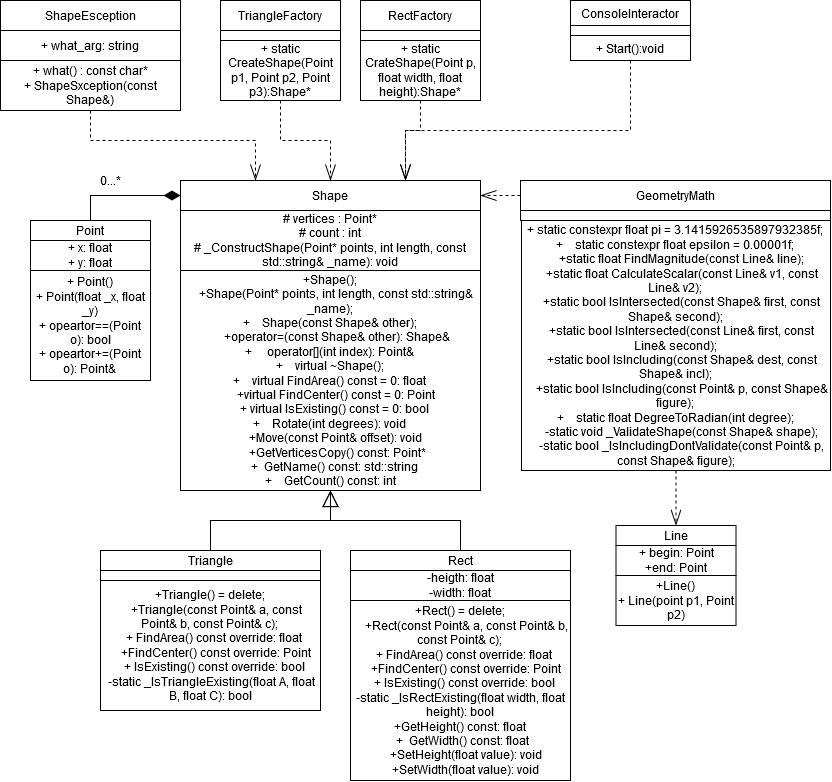


Рисунок 1 – Диаграмма классов

**ЛИСТИНГ ПРОГРАММЫ**

Triangle::Triangle(const Point& a, const Point& b, const Point& c)

{

int count = 3;

Point\* pts = new Point[count];

pts[0] = a;

pts[1] = b;

pts[2] = c;

\_ConstructShape(pts, count, "Triangle");

delete[] pts;

}

float Triangle::FindArea() const

{

Point a = vertices[0], b = vertices[1], c = vertices[2];

float A = GeometryMath::FindMagnitude({ a, b });

float B = GeometryMath::FindMagnitude({ b, c });

float C = GeometryMath::FindMagnitude({ b, c });

if (!\_IsTriangleExisting(A, B, C))

{

throw new ShapeException(\*this);

}

float p = (A + B + C) / 2;

return sqrt(p \* (p - A) \* (p - B) \* (p - C));

}

Point Triangle::FindCenter() const

{

Point a = vertices[0], b = vertices[1], c = vertices[2];

float A = GeometryMath::FindMagnitude({ a, b });

float B = GeometryMath::FindMagnitude({ b, c });

float C = GeometryMath::FindMagnitude({ b, c });

if (!\_IsTriangleExisting(A, B, C))

{

throw new ShapeException(\*this);

}

float Ox = 0, Oy = 0;

for (int i = 0; i < count; i++)

{

Ox += vertices[i].x;

Oy += vertices[i].y;

}

return Point(Ox / count, Oy / count);

return Point(Ox / count, Oy / count);

}

bool Triangle::IsExisting() const

{

Point a = vertices[0], b = vertices[1], c = vertices[2];

float A = GeometryMath::FindMagnitude({ a, b });

float B = GeometryMath::FindMagnitude({ b, c });

float C = GeometryMath::FindMagnitude({ b, c });

return \_IsTriangleExisting(A, B, C);

}

bool Triangle::\_IsTriangleExisting(float A, float B, float C)

{

if (A + B > C&& A + C > B&& B + C > A)

{

return true;

}

return false;

}

Rect::Rect(const Point& point, float \_height, float \_width)

: width(\_width), height(\_height)

{

int count = 4;

Point\* pts = new Point[count];

pts[0] = point;

pts[1] = Point(point.x, point.y + height);

pts[2] = Point(point.x + width, point.y + height);

pts[3] = Point(point.x + width, point.y);

\_ConstructShape(pts, count, "Rect");

delete[] pts;

}

float Rect::FindArea() const

{

if (!\_IsRectExisting(width, height))

{

throw new ShapeException(\*this);

}

return height \* width;

}

Point Rect::FindCenter() const

{

if (!\_IsRectExisting(width, height))

{

throw new ShapeException(\*this);

}

float Ox = 0, Oy = 0;

for (int i = 0; i < count; i++)

{

Ox += vertices[i].x;

Oy += vertices[i].y;

}

return Point(Ox / count, Oy / count);

}

bool Rect::IsExisting() const

{

return \_IsRectExisting(width, height);

}

float Rect::GetHeight() const

{

return width;

}

float Rect::GetWidth() const

{

return height;

}

void Rect::SetHeight(float value)

{

height = value;

Point p = vertices[0];

vertices[1] = Point(p.x, p.y + height);

vertices[2] = Point(p.x + width, p.y + height);

vertices[3] = Point(p.x + width, p.y);

}

void Rect::SetWidth(float value)

{

width = value;

Point p = vertices[0];

vertices[1] = Point(p.x, p.y + height);

vertices[2] = Point(p.x + width, p.y + height);

vertices[3] = Point(p.x + width, p.y);

}

bool Rect::\_IsRectExisting(float width, float height)

{

return height != 0 && width != 0;

}

void ConsoleInteractor::Start()

{

Shape\*\* figures;

int index = 0;

int count = 10;

figures = new Shape \* [count];

cout << "Commands: \n";

cout << " 1: create shape, 2: delete last shape, 3: print shape by index\n";

cout << " 4: find center, 5: find area, 6 rotate shape\n";

cout << " 7: is shapes intersected, 8: is shapes included\n";

cout << " 0: exit\n\n";

int var1, var2, switch\_on;

std::string name;

while (true)

{

cout << ">>> ";

cin >> switch\_on;

try

{

switch (switch\_on)

{

case 0:

goto end;

break;

case 1:

if (index == 9)

{

cout << "Out of place for figures (max is " << count << ")\n";

}

cout << "Enter shape's name\n>>> ";

cin >> name;

for (auto& i : name)

{

i = toupper(i);

}

if (name == "TRIANGLE")

{

float x, y;

Point p[3];

for (int i = 0; i < 3; i++)

{

cout << "Enter a point in format [x y]\n>>> ";

cin >> x >> y;

p[i] = Point(x, y);

}

cout << "\n";

figures[index++] = TriangleFactory::CreateShape(p[0], p[1], p[2]);

}

else if (name == "RECT")

{

float x, y;

cout << "Enter a point in format [x y]\n>>> ";

cin >> x >> y;

Point p(x, y);

cout << "Enter a width and height in format [w h]\n>>> ";

cin >> x >> y;

figures[index++] = RectFactory::CreateShape(p, x, y);

}

else

{

cout << "Invalid name\n";

}

break;

case 2:

if (index != 0)

{

delete figures[index - 1];

index--;

}

break;

case 3:

cout << "Enter index\n>>> ";

cin >> var1;

if (var1 < 0 || var1 >= index)

{

cout << "Invalid index\n";

break;

}

else

{

Shape\* s = figures[var1];

cout << "\nShape's name: " << s->GetName() << "\n";

for (int i = 0; i < s->GetCount(); i++)

{

cout << i << " vertex: x = " << (\*s)[i].x << " y = " << (\*s)[i].y << "\n";

}

cout << "\n";

}

break;

case 4:

cout << "Enter index\n>>> ";

cin >> var1;

if (var1 < 0 || var1 >= index)

{

cout << "Invalid index\n";

break;

}

else

{

Point c = figures[var1]->FindCenter();;

cout << "Center: x - " << c.x << " y - " << c.y << "\n";

}

break;

case 5:

cout << "Enter index\n>>> ";

cin >> var1;

if (var1 < 0 || var1 >= index)

{

cout << "Invalid index\n";

break;

}

else

{

cout << "Area = " << figures[var1]->FindArea() << "\n";

}

break;

case 6:

cout << "Enter index\n>>> ";

cin >> var1;

if (var1 < 0 || var1 >= index)

{

cout << "Invalid index\n";

break;

}

else

{

cout << "Enter a degrees\n>>> ";

cin >> var2;

figures[var1]->Rotate(var2);

}

break;

case 7:

cout << "Enter indexes\n>>> ";

cin >> var1 >> var2;

if (var1 < 0 || var1 >= index || var2 < 0 || var2 >= index)

{

cout << "Invalid index\n";

break;

}

else

{

cout << "answer is: " << (GeometryMath::IsIntersected(\*figures[var1], \*figures[var2]) ? "true\n" : "false\n");

}

break;

case 8:

cout << "Enter indexes\n>>> ";

cin >> var1 >> var2;

if (var1 < 0 || var1 >= index || var2 < 0 || var2 >= index)

{

cout << "Invalid index\n";

break;

}

else

{

cout << "first figure including second? " << (GeometryMath::IsIncluding(\*figures[var1], \*figures[var2]) ? "true\n" : "false\n");

}

break;

default:

cout << "Try again\n";

break;

}

}

catch (const std::exception& e)

{

cout << e.what() << "\n";

}

}

end:

for (int i = 0; i < index; i++)

{

delete figures[i];

}

delete[] figures;

}

float GeometryMath::FindMagnitude(const Line& line)

{

float Vx = line.end.x - line.begin.x;

float Vy = line.end.y - line.begin.y;

return sqrt(Vx \* Vx + Vy \* Vy);

}

float GeometryMath::CalculateScalar(const Line& v1, const Line& v2)

{

float v1\_x = v1.end.x - v1.begin.x;

float v1\_y = v1.end.y - v1.begin.y;

float v2\_x = v2.end.x - v2.begin.x;

float v2\_y = v2.end.y - v2.begin.y;

return v1\_x \* v2\_x + v1\_y \* v2\_y;

}

// O(n\*n) ;(

bool GeometryMath::IsIntersected(const Shape& first, const Shape& second)

{

\_ValidateShape(first);

\_ValidateShape(second);

if (first.GetCount() == 0 || second.GetCount() == 0)

{

return false;

}

bool ans = false;

Point\* first\_vert = first.GetVerticesCopy();

Point\* second\_vert = second.GetVerticesCopy();

// Filling to arrays with lines

int lines\_c = first.GetCount();

Line\* lines = new Line[lines\_c];

int check\_c = second.GetCount();

Line\* check = new Line[check\_c];

for (int i = 0; i < lines\_c - 1; i++)

{

lines[i] = Line(first\_vert[i], first\_vert[i + 1]);

}

lines[lines\_c - 1] = Line(first\_vert[lines\_c - 1], first\_vert[0]);

for (int i = 0; i < check\_c - 1; i++)

{

check[i] = Line(second\_vert[i], second\_vert[i + 1]);

}

check[check\_c - 1] = Line(second\_vert[check\_c - 1], second\_vert[0]);

// checking intersection of lines array with check array

for (int i = 0; i < lines\_c && !ans; i++)

{

for (int j = 0; j < check\_c; j++)

{

if (GeometryMath::IsIntersected(lines[i], check[j]))

{

ans = true;

break;

}

}

}

delete[] check;

delete[] lines;

delete[] first\_vert;

delete[] second\_vert;

return ans;

}

bool GeometryMath::IsIntersected(const Line& first, const Line& second)

{

Point p1 = first.begin;

Point p2 = first.end;

Point p3 = second.begin;

Point p4 = second.end;

if (p1 == p3 || p2 == p4)

{

return true;

}

float v1 = (p4.x - p3.x) \* (p1.y - p3.y) - (p4.y - p3.y) \* (p1.x - p3.x);

float v2 = (p4.y - p3.y) \* (p2.x - p1.x) - (p4.x - p3.x) \* (p2.y - p1.y);

float v3 = (p2.x - p1.x) \* (p1.y - p3.y) - (p2.y - p1.y) \* (p1.x - p3.x);

float v4 = (p4.y - p3.y) \* (p2.x - p1.x) - (p4.x - p3.x) \* (p2.y - p1.y);

float u1 = v1 / v2, u2 = v3 / v4;

return u1 >= 0 && u1 <= 1 && u2 >= 0 && u2 <= 1;

}

bool GeometryMath::IsIncluding(const Shape& dest, const Shape& incl)

{

\_ValidateShape(dest);

\_ValidateShape(incl);

bool ans = true;

Point\* pts = incl.GetVerticesCopy();

for (int i = 0; i < incl.GetCount(); i++)

{

if (\_IsIncludingDontValidate(pts[i], dest) == false)

{

ans = false;

break;

}

}

delete[] pts;

return ans;

}

bool GeometryMath::IsIncluding(const Point& p, const Shape& figure)

{

\_ValidateShape(figure);

return \_IsIncludingDontValidate(p, figure);

}

float GeometryMath::DegreeToRadian(int degree)

{

return degree \* pi / 180;

}

void GeometryMath::\_ValidateShape(const Shape& shape)

{

if (shape.IsExisting() == false)

{

throw new ShapeException(shape);

}

}

bool GeometryMath::\_IsIncludingDontValidate(const Point& p, const Shape& figure)

{

bool ans = false;

int count = figure.GetCount() + 1;

Line\* lines = new Line[count];

Point\* figure\_points = figure.GetVerticesCopy();

for (int i = 0; i < count - 1; i++)

{

lines[i] = Line(p, figure\_points[i]);

}

lines[count - 1] = Line(lines[0]);

float cos\_angle = 0.f;

float scalar, v1, v2;

for (int i = 0; i < count - 1; i++)

{

scalar = CalculateScalar(lines[i], lines[i + 1]);

v1 = FindMagnitude(lines[i]);

v2 = FindMagnitude(lines[i + 1]);

cos\_angle += acos(scalar / (v1 \* v2));

}

// Sum of angles must be equal 2\*pi

if (abs(cos\_angle - 2 \* pi) <= epsilon)

{

ans = true;

}

delete[] lines;

delete[] figure\_points;

return ans;

}

Line::Line()

: begin(), end()

{}

Line::Line(Point from, Point to)

: begin(from), end(to)

{}

bool Point::operator==(const Point& other)

{

return x == other.x && y == other.y;

}

Point& Point::operator+=(const Point& offset)

{

x += offset.x;

y += offset.y;

return \*this;

}

void Shape::Rotate(int degrees)

{

if (!IsExisting())

{

throw new ShapeException(\*this);

}

float a = GeometryMath::DegreeToRadian(degrees % 360);

for (int i = 0; i < count; i++)

{

float x = vertices[i].x;;

float y = vertices[i].y;

vertices[i] = Point(x \* cos(a) - y \* sin(a), x \* sin(a) + y \* cos(a));

}

}

void Shape::Move(const Point& offset)

{

if (!IsExisting())

{

throw new ShapeException(\*this);

}

for (int i = 0; i < count; i++)

{

vertices[i] += offset;

}

}

Point\* Shape::GetVerticesCopy() const

{

if (count == 0)

{

return nullptr;

}

Point\* arr = new Point[count];

std::copy(vertices, vertices + count, arr);

return arr;

}

std::string Shape::GetName() const

{

return name;

}

int Shape::GetCount() const

{

return count;

}

void Shape::\_CopyArray(Point\* points, int length)

{

vertices = new Point[length];

if (points != nullptr)

{

std::copy(points, points + length, vertices);

}

}

void Shape::\_ConstructShape(Point\* points, int length, const std::string& \_name)

{

name = \_name;

count = length;

\_CopyArray(points, count);

}

Shape::Shape()

: count(0), vertices(nullptr), name("Shape")

{}

Shape::Shape(Point\* points, int length, const std::string& \_name)

: vertices(nullptr), count(length), name(\_name)

{

if (count <= 0)

{

throw new std::invalid\_argument("Invalid array length");

}

\_CopyArray(points, count);

}

Shape::Shape(const Shape& other)

: count(other.count), name(other.name)

{

\_CopyArray(other.vertices, count);

}

Shape& Shape::operator=(const Shape& other)

{

if (this == &other)

{

return \*this;

}

if (vertices != nullptr)

{

delete[] vertices;

}

count = other.count;

\_CopyArray(other.vertices, count);

return \*this;

}

Point& Shape::operator[](int index)

{

if (index >= count || index < 0)

{

throw new std::out\_of\_range("invalid index uses in Shape::operator[]");

}

return vertices[index];

}

Shape::~Shape()

{

if (vertices != nullptr)

{

delete[] vertices;

}

}

ShapeException::ShapeException(const Shape& shape)

{

std::ostringstream sstr;

sstr << "Shape " << shape.GetName() << " with " << shape.GetCount() << " vertices doesn't exist";

what\_arg = sstr.str();

}

const char\* ShapeException::what() const noexcept

{

return what\_arg.c\_str();

}

Shape\* TriangleFactory::CreateShape(const Point& a, const Point& b, const Point& c)

{

return new Triangle(a, b, c);

}

Shape\* RectFactory::CreateShape(const Point& a, float width, float height)

{

return new Rect(a, height, width);

}