|  |  |
| --- | --- |
| Определение принадлежности точки невыпуклому многоугольнику | |
| Внутренняя спецификация | |
| Студент | Нечта Б.О. |
| Преподаватель | Кулюкин К.С. |
| Сдано |  |

1. Назначение

Программа предназначена для определения принадлежности точки невыпуклому многоугольнику.

1. Описание структур данных

Перечисление ErrorType содержит значения:  
noError, inputFileNotExist, outputFileCreateFail, invalidVertexCount, coordinateOutOfRange, pointOutOfRange, pointNotInteger, duplicateVertex, invalidPolygon, invalidCharacters, wrongElementCountInLine, emptyLineFound, verticesMismatch, wrongOrder, emptyFile

Класс Error содержит поля:  
Поля:

ErrorType type — тип ошибки.

std::string errorMessage — текст сообщения об ошибке.

int errorLineNumber — номер строки, где обнаружена ошибка.

std::string errorLineContent — содержимое строки с ошибкой.

std::string errorInputFileWay — путь к входному файлу, в котором произошла ошибка.

std::string errorOutputFileWay — путь к выходному файлу.

Методы:

**std::string generateErrorMessage()** — формирует полное описание ошибки. Этот метод создает строку, которая включает в себя тип ошибки, сообщение об ошибке, номер строки с ошибкой, содержание строки, а также пути к входному и выходному файлам.

Структура Point содержит поля:  
float x – координата X  
float y – координата Y

**Point(int x = 0, int y = 0)** — конструктор, инициализирующий координаты точки значениями по умолчанию: x = 0 и y = 0.

Операторы ==, < — для сравнения и использования в std::set или std::map

Класс Polygon

Поля:

std::vector<Point> vertices — список вершин многоугольника.

Методы:

bool isValidPolygon() — метод, который проверяет корректность многоугольника. Он проверяет, например, правильность количества вершин, отсутствие самопересечений и другие условия для допустимого многоугольника.

bool containsPoint(Point p) — метод, который проверяет, принадлежит ли точка p многоугольнику. Алгоритм обычно использует метод с пересечением луча или другой подход для определения принадлежности.

3. Описание алгоритмов функций

1. Главная функция программы:

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

Входные данные:

Параметры командной строки:

argc — количество аргументов командной строки.

argv[] — массив строковых аргументов командной строки:

argv[1] — путь к входному файлу с данными о многоугольнике и точке.

argv[2] — путь к выходному файлу для записи результата.

Входные аргументы командной строки:

Программа должна принимать два аргумента: путь к входному файлу и путь к выходному файлу.

Выходные данные:

Программа возвращает 0, если выполнение прошло успешно.

В случае ошибок программа завершится с ненулевым кодом, соответствующим типу ошибки.

Алгоритм функции:

// Считываем путь к входному файлу и выходному файлу из аргументов командной строки

// Программа должна принимать два аргумента: путь к входному и выходному файлам.

// Проверка на количество аргументов

// Если количество аргументов меньше двух, выводим ошибку и завершаем программу с кодом ошибки.

// Открытие входного файла

// Программа пытается открыть файл по пути, переданному в argv[1].

// Если файл не открылся, устанавливаем ошибку типа inputFileNotExist в объект Error и завершаем программу с соответствующим кодом ошибки. (если > 2 аргументов)

// Считывание данных из входного файла

// Используем метод parseFromFile для чтения данных о многоугольнике и точке из файла.

// Если при чтении данных возникает ошибка (например, файл пуст или данные некорректны), записываем ошибку в выходной файл и завершаем программу.

// Валидация данных

// После того как данные считаны, вызываем метод validate для проверки их корректности.

// Проверяем на пустые строки, некорректное количество элементов, дублирующиеся вершины и другие ошибки.

// Если данные невалидны, записываем ошибку в выходной файл и завершаем выполнение программы с кодом ошибки.

// Проверка корректности многоугольника

// Если данные валидны, проверяем сам многоугольник с помощью метода isValidPolygon.

// Проверяем количество вершин, порядок точек, на наличие коллинеарных точек, самопересечений и выпуклость.

// Если многоугольник некорректен, записываем ошибку в выходной файл и завершаем выполнение программы с кодом ошибки.

// Проверка принадлежности точки многоугольнику

// Если многоугольник корректен, вызываем метод containsPoint для проверки, принадлежит ли точка многоугольнику.

// Если точка не принадлежит многоугольнику, записываем ошибку в выходной файл и завершаем выполнение программы с кодом ошибки.

// Запись результата в выходной файл

// Если все проверки прошли успешно, записываем результат "Success" в выходной файл с помощью метода writeResult.

// Завершение работы программы

// Если все шаги выполнены успешно, программа завершает работу с кодом 0.

// В случае ошибок программа завершится с ненулевым кодом, соответствующим типу ошибки.

}

1. readFromFile(std::string fileName, Error& err) отвечает за чтение данных из входного файла и их предварительную обработку.

Входные данные:

fileName — строка, содержащая путь к входному файлу.

err — объект типа Error, в котором будут храниться ошибки, если они возникнут.

Выходные данные:

Возвращает true, если данные успешно считаны и подготовлены для дальнейшей обработки. В случае ошибок — false.

Алгоритм функции:

// Открыть файл с именем fileName

// Если файл не открылся:

// Установить ошибку типа inputFileNotExist в объект err

// Вернуть false

// Если файл открыт успешно:

// Для каждой строки файла:

// Проверить на пустую строку с использованием checkEmptyLine

// Проверить на некорректное количество элементов в строке с использованием checkInvalidElementCount

// Проверить на некорректные символы в строке с использованием checkInvalidCharacters

// Проверить на целые числа в координатах с использованием checkNonIntegerCoordinates

// Считать координаты и проверить на выход за пределы с использованием checkOutOfRangeCoordinates

// Добавить вершину в список

// Проверить на дублирующиеся вершины с использованием checkDuplicateVertices

// Проверить на корректный порядок точек с использованием checkCorrectVertexOrder

// Проверить на допустимое количество вершин с использованием checkVertexCount

// Закрыть файл

// Вернуть true

1. writeResult(std::string fileName, bool result) – метод записи результата в выходной файл.

Входные данные:

fileName – путь к выходному файлу

result – булевое значение, указывающее на успешность выполнения (true - успех, false - ошибка)

Выходные данные:

Записывает в файл результат выполнения программы

Алгоритм функции:

// Открыть файл с именем fileName для записи

// Если файл не открылся

// Установить ошибку типа outputFileNotExist в объект err

// Завершить выполнение функции

// Если файл открыт успешно

// Если результат успешный (result == true)

// Записать "Success" в файл

// Иначе

// Записать "Failure" в файл

// Закрыть файл

1. writeErrorToConsole(Error err) – метод вывода сообщения об ошибке в консоль

Входные данные:

err – объект класса Error, содержащий информацию об ошибке

Выходные данные:

Выводит сообщение об ошибке в консоль

Алгоритм функции:

// Проверить, есть ли ошибка в объекте err

// Если ошибка присутствует

// Вывести сообщение об ошибке на консоль:

// Ошибка: (err.getErrorMessage())

// Иначе

// Вывести сообщение, что ошибки не обнаружены

1. bool validate(Error& err) – метод валидации данных

Входные данные:  
err — объект типа Error, который будет хранить информацию о найденных ошибках во время валидации данных.

Выходные данные:

Возвращает true, если данные валидны, и false, если обнаружены ошибки.

err — объект типа Error, в котором будут сохранены сведения о возникших ошибках (если таковые имеются).

Алгоритм функции:

// Проверить на пустой файл

// Если файл пустой

// Установить ошибку "пустой файл" в err

// Вернуть false

// Пройти по строкам файла

// Для каждой строки:

// Проверить на пустоту (checkEmptyLine)

// Проверить на некорректное количество элементов (checkInvalidElementCount)

// Проверить на некорректные символы (checkInvalidCharacters)

// Проверить на целые числа в координатах (checkNonIntegerCoordinates)

// Проверить на выход за диапазон (checkOutOfRangeCoordinates)

// Проверить на дублирующиеся вершины (checkDuplicateVertices)

// Проверить на правильный порядок точек (checkCorrectVertexOrder)

// Проверить на корректное количество вершин (checkVertexCount)

// Если все проверки прошли успешно

// Вернуть true

1. isValidPolygon(std::vector <Point> vertices). – метод проверки корректности многоугольника

Входные данные:

Многоугольник, представленный набором вершин (std::vector <Point> vertices)

Выходные данные:

true - если многоугольник корректен

false - если многоугольник некорректен

Алгоритм:

// Если количество вершин меньше 3

// Вызвать checkVertexCount() для проверки допустимого количества вершин

// Если количество вершин некорректно

// Вернуть false

// Проверить на дублирующиеся вершины

// Вызвать checkDuplicateVertices() для проверки дублирующихся точек

// Если дублирующиеся вершины найдены

// Вернуть false

// Проверить на правильность порядка вершин

// Вызвать checkCorrectVertexOrder() для проверки порядка точек

// Если порядок точек некорректен

// Вернуть false

// Проверить, чтобы все вершины не были коллинеарными (нельзя чтобы три точки на одном прямом отрезке)

// Для каждой тройки последовательных точек (P[i], P[i+1], P[i+2]):

// Вызвать checkCollinearity(P[i], P[i+1], P[i+2]) для проверки коллинеарности

// Если коллинеарность обнаружена

// Вернуть false

// Проверить на корректность координат (например, чтобы не выходили за пределы допустимого диапазона)

// Для каждой вершины (P[i]):

// Вызвать checkOutOfRangeCoordinates(P[i].x, P[i].y, err, i+1) для проверки координат

// Если координаты выходят за пределы

// Вернуть false

// Проверить, чтобы точки не располагались внутри полигона (это можно сделать через containsPoint)

// Для каждой вершины (P[i]):

// Вызвать containsPoint(P[i]) для проверки, принадлежит ли точка многоугольнику

// Если точка лежит внутри

// Вернуть false

// Проверить на самопересечения

// Для каждой пары рёбер (P[i], P[i+1]) и (P[j], P[j+1]) проверить, пересекаются ли они

// Вызвать checkIntersection(P[i], P[i+1], P[j], P[j+1]) для проверки пересечений

// Если пересечение найдено

// Вернуть false

// Проверить, чтобы многоугольник был невыпуклым

// Для каждого рёберного сегмента (P[i], P[i+1]) и следующего сегмента (P[i+1], P[i+2])

// Проверить знак поворота, используя векторное произведение

// Если все повороты одинаковые (все положительные или все отрицательные), то многоугольник выпуклый

// В таком случае вернуть false

// Если все проверки пройдены успешно

// Вернуть true

1. containsPoint(Point p) – метод проверки, принадлежит ли точка многоугольнику

Входные данные:

Точка p, которую нужно проверить на принадлежность многоугольнику

Выходные данные:

true - если точка принадлежит многоугольнику

false - если точка не принадлежит многоугольнику

Алгоритм:

// Инициализируем переменную для подсчёта количества пересечений горизонтального луча с рёбрами многоугольника

// В этой переменной будем хранить количество пересечений луча с рёбрами

// Изначально количество пересечений равно нулю

// Для каждого ребра многоугольника (состоящего из двух точек, P[i] и P[i+1])

// Перебираем все рёбра многоугольника

// Каждый отрезок ребра многоугольника определяется двумя точками: P[i] и P[i+1]

// Если мы дошли до последней вершины, то берём первую точку, чтобы замкнуть многоугольник (P[(i + 1) % vertices.size()])

// Проверяем, пересекает ли горизонтальный луч, проходящий через точку p, это ребро

// Проверяем, находятся ли точки P1 и P2 (концы текущего ребра) по разные стороны от горизонтальной оси, через точку p

// Если одна точка выше точки p, а другая ниже, то предполагаем, что луч может пересечь это ребро

// Далее проверяем, не лежит ли точка p на линии, соединяющей P1 и P2

// Для этого вычисляем координату X точки пересечения луча с ребром и проверяем, находится ли точка p левее этой координаты

// Если пересечение найдено, увеличиваем счётчик пересечений

// Увеличиваем переменную intersections на единицу, если пересечение обнаружено

// После того, как мы перебрали все рёбра, подсчитываем количество пересечений

// Если количество пересечений нечётное, точка находится внутри многоугольника

// Если количество пересечений чётное, точка находится снаружи многоугольника

// Завершаем выполнение функции

// Возвращаем результат, который зависит от количества пересечений:

// Если количество пересечений нечётное, то точка принадлежит многоугольнику (внутри)

// Если количество пересечений чётное, то точка находится снаружи многоугольника

1. checkEmptyLine(std::string line, Error& err, int lineNumber) – Проверка на пустую строку.

Входные данные:

line — строка для проверки.

err — объект для записи ошибки.

lineNumber — номер строки в файле.

Выходные данные:

Если строка пуста, возвращаем ошибку.

Алгоритм:

// Если строка line пуста (не содержит символов)

// Создать описание ошибки в объекте err, указывая, что строка пустая

// В описании ошибки добавить номер строки, в которой обнаружена пустая строка

// В сообщении ошибки указать информацию о том, что строка не содержит данных

// Вернуть объект ошибки с сообщением об ошибке

1. checkInvalidElementCount(std::string line, Error& err, int lineNumber) – Проверка на некорректное количество элементов в строке.

Входные данные:

line — строка, содержащая данные для проверки.

err — объект для записи ошибки.

lineNumber — номер строки в исходном файле.

Выходные данные:

Если количество элементов в строке некорректно, возвращаем ошибку.

Алгоритм:

// Разделить строку line на элементы (например, по пробелам или запятым)

// Подсчитать количество элементов, полученных после разделения строки

// Если количество элементов не соответствует ожидаемому (например, меньше или больше нужного)

// Создать описание ошибки в объекте err, указывая, что количество элементов некорректно

// В сообщении ошибки указать номер строки и информацию о некорректном количестве элементов

// Вернуть объект ошибки с соответствующим сообщением

1. checkInvalidCharacters(std::string line, Error& err, int lineNumber) – Проверка на некорректные символы в строке.

Входные данные:

line — строка для проверки.

err — объект для записи ошибки.

lineNumber — номер строки в файле.

Выходные данные:

Если строка содержит некорректные символы, возвращаем ошибку.

Алгоритм:

// Для каждой строки line

// Пройти по всем символам строки

// Проверить каждый символ на допустимость (например, можно использовать регулярные выражения для проверки)

// Если в строке встречается символ, который не является допустимым

// Создать описание ошибки в объекте err, указывая, что строка содержит некорректные символы

// В сообщении ошибки указать номер строки и описание некорректных символов

// Вернуть объект ошибки с сообщением о найденной ошибке

11**.** checkNonIntegerCoordinates(std::string line, Error& err, int lineNumber) – Проверка на целые числа в координатах.

Входные данные:  
line — строка, содержащая координаты.  
err — объект ошибки для записи результата.  
lineNumber — номер текущей строки.

Выходные данные:

Если координаты не являются целыми числами, устанавливается ошибка pointNotInteger и функция возвращает false.

// Разделить строку на элементы (по пробелу)

// Для каждого элемента строки:

// Попробовать преобразовать элемент в целое число (используя toInt())

// Если преобразование не удалось:

// Установить тип ошибки pointNotInteger

// Записать номер строки и её содержимое в err

// Вернуть false

// Если все элементы успешно преобразованы в целые числа

// Вернуть true

12. checkOutOfRangeCoordinates(float x, float y, Error& err, int lineNumber) – Проверка на выход за пределы диапазона координат**.**

Входные данные:

x, y — координаты точки.

err — объект ошибки.

lineNumber — номер строки, в которой эти координаты считаны.

Выходные данные:

Если хотя бы одна координата выходит за границы [-999; 999], устанавливается ошибка coordinateOutOfRange и функция возвращает false.

В случае, если координаты в пределах допустимого диапазона, функция возвращает true.

Алгоритм функции:

// Проверить, лежат ли x и y в пределах диапазона [-999; 999]

// Если x < -999 или x > 999 или y < -999 или y > 999:

// Установить ошибку типа coordinateOutOfRange в объект err

// Указать номер строки и конкретные координаты (x, y)

// Вернуть false

// Иначе:

// Вернуть true

13. checkDuplicateVertices(std::vector <Point>& vertices, Error& err, int lineNumber) – Проверка на дублирующиеся вершины.

Входные данные:

vertices — вектор точек (вершин).

err — объект ошибки.

lineNumber — строка, где был добавлен дубликат.

Выходные данные:

Если обнаружены одинаковые вершины, устанавливается ошибка duplicateVertex.

Алгоритм функции:

// Создать пустой std::set для хранения уникальных точек

// Для каждой точки в vertices:

// Если точка уже есть в множестве:

// Установить тип ошибки duplicateVertex

// Указать номер строки и дубликатную вершину

// Вернуть false

// Иначе:

// Добавить точку в множество

// Вернуть true, если дубликатов не найдено

14. checkCorrectVertexOrder(std::vector <Point>& vertices, Error& err) – Проверка на корректность порядка точек.

Входные данные:

vertices — список точек полигона.

err — объект ошибки.

Выходные данные:

Если порядок точек не соответствует направлению обхода (например, нарушена ориентация или порядок хаотичен), устанавливается ошибка wrongOrder.

Алгоритм функции:

// Вычислить ориентированную площадь многоугольника по формуле Гаусса (shoelace formula):

// Площадь = 0.5 \* сумма по всем i (x[i] \* y[i+1] - x[i+1] \* y[i])

// Если результат отрицателен — порядок обхода по часовой стрелке

// Если результат положителен — против часовой

// Если нужен конкретный порядок (например, против часовой стрелки):

// Если порядок не совпадает:

// Установить ошибку wrongOrder

// Вернуть false

// Вернуть true, если порядок корректен

15. checkVertexCount(int vertexCount, Error& err) – Проверка на допустимое количество вершин.

Входные данные:

vertexCount — количество вершин.

err — объект ошибки.

Выходные данные:

Если количество вершин < 3 или слишком большое (> 1000, например), устанавливается ошибка invalidVertexCount.

Алгоритм:

// Если vertexCount < 3 или vertexCount > 1000:

// Установить ошибку invalidVertexCount

// Записать сообщение с числом вершин

// Вернуть false

// Вернуть true, если количество допустимо

16. checkCollinearity(Point a, Point b, Point c) – Проверка на коллинеарность трёх точек.

Входные данные:

a, b, c — три точки.

Выходные данные:

true, если точки коллинеарны (лежат на одной прямой); false в противном случае.

Алгоритм:

// Вычислить векторное произведение векторов AB и AC

// cross = (b.x - a.x) \* (c.y - a.y) - (b.y - a.y) \* (c.x - a.x)

// Если cross == 0:

// Точки лежат на одной прямой => вернуть true

// Иначе:

// Вернуть false

17. checkIntersection(Point a1, Point a2, Point b1, Point b2) – Проверка на пересечение двух отрезков

Входные данные:

a1, a2 — точки первого отрезка

b1, b2 — точки второго отрезка

Выходные данные:

true, если отрезки пересекаются

false, если не пересекаются

Алгоритм:

// Вычислить ориентацию троек точек:

// orientation1 = знак ((a2.y - a1.y) \* (b1.x - a2.x) - (a2.x - a1.x) \* (b1.y - a2.y))

// orientation2 = знак ((a2.y - a1.y) \* (b2.x - a2.x) - (a2.x - a1.x) \* (b2.y - a2.y))

// orientation3 = знак ((b2.y - b1.y) \* (a1.x - b2.x) - (b2.x - b1.x) \* (a1.y - b2.y))

// orientation4 = знак ((b2.y - b1.y) \* (a2.x - b2.x) - (b2.x - b1.x) \* (a2.y - b2.y))

// Если orientation1 != orientation2 и orientation3 != orientation4:

// Отрезки пересекаются → вернуть true

// Проверить коллинеарные случаи:

// Если orientation1 == 0 и точка b1 лежит между a1 и a2:

// Вернуть true

// Если orientation2 == 0 и точка b2 лежит между a1 и a2:

// Вернуть true

// Если orientation3 == 0 и точка a1 лежит между b1 и b2:

// Вернуть true

// Если orientation4 == 0 и точка a2 лежит между b1 и b2:

// Вернуть true

// Для проверки «точка лежит между двумя другими»:

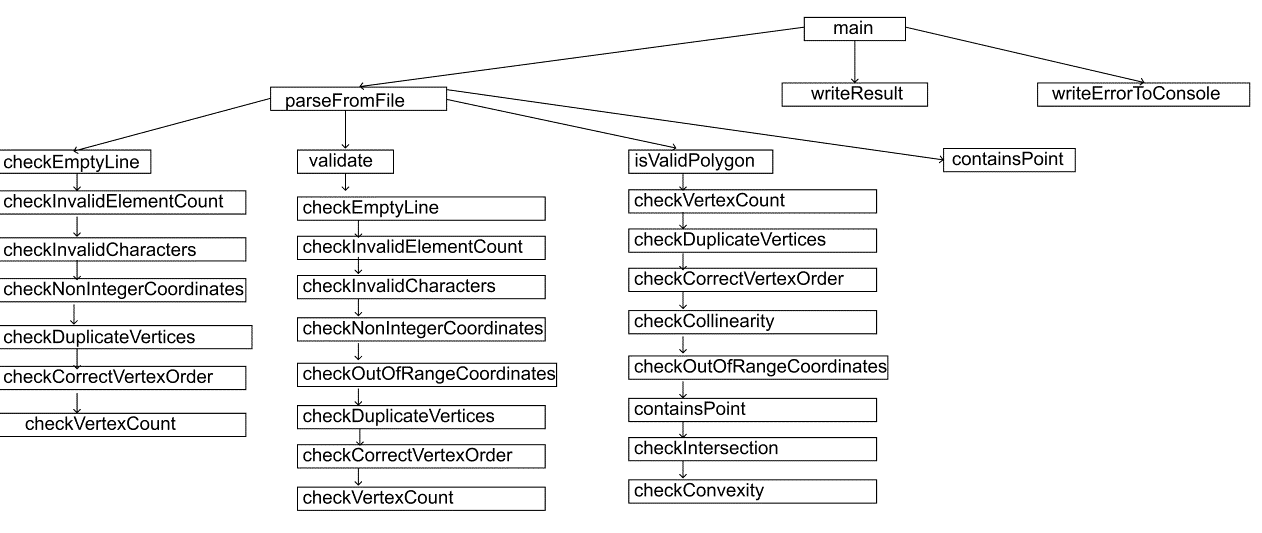
// Проверить, что min(p.x, r.x) <= q.x <= max(p.x, r.x)

// и одновременно min(p.y, r.y) <= q.y <= max(p.y, r.y)

// Если ни одно условие не выполнено:

// Вернуть false

Диаграмма вызовов функций приведена в приложении №1.



UML-диаграмма классов приведена в приложении №2.

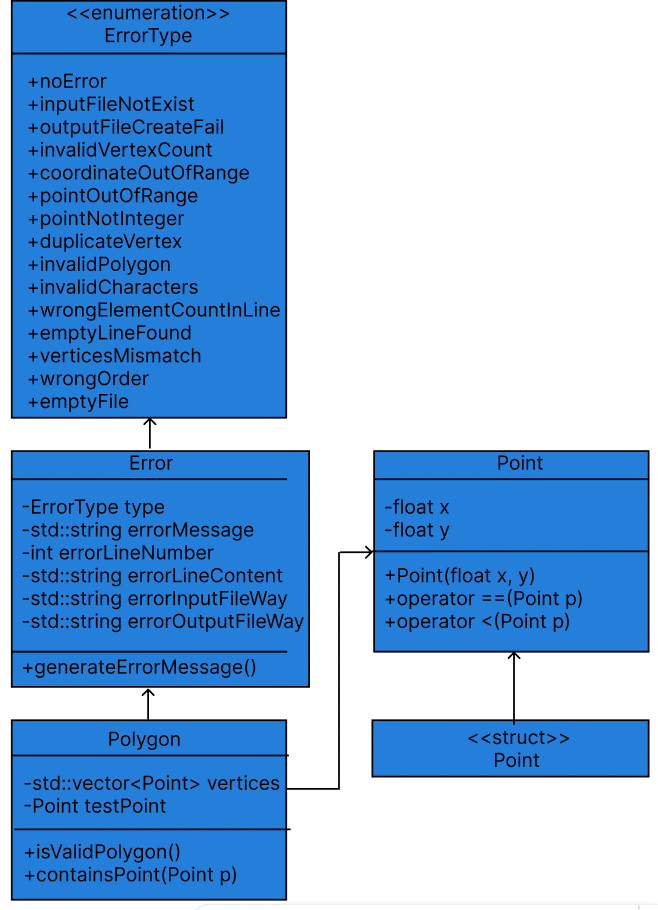


Диаграмма потоков данных приведена в приложении №3.

