**Московский авиационный институт**

**(Национальный исследовательский университет)**

Институт: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Объектно-ориентированное программирование»

**Лабораторная работа № 3**

Тема: Механизмы наследования в С++

Студент: Шараковский Юрий Дмитриевич

Группа: 80-206

Преподаватель: Чернышов Л.Н.

Дата:

Оценка:

Москва, 2020

1. Постановка задачи

Разработать классы согласно варианту задания, классы должны наследоваться от базового класса Figure. Фигуры являются фигурами вращения. Все классы должны поддерживать набор общих методов:

1. Вычисление геометрического центра фигуры;

2. Вывод в стандартный поток вывода std::cout координат вершин фигуры;

3. Вычисление площади фигуры;

Создать программу, которая позволяет:

• Вводить из стандартного ввода std::cin фигуры, согласно варианту задания.

• Сохранять созданные фигуры в динамический массив std::vector<Figure\*>

• Вызывать для всего массива общие функции (1-3 см. выше).

• Необходимо уметь вычислять общую площадь фигур в массиве.

• Удалять из массива фигуру по индексу;

Вариант 25: треугольник, квадрат, прямоугольник

1. Описание программы

Программа описывает простую иерархию классов - в основании которой находится абстрактный класс Figure, от него наследуются три класса-потомка: Triangle, Square и Rectangle. Каждый из классов-потомков имеет собственную реализацию методов поиска площади и центра фигуры. В функции main со стандартного потока считывается название фигуры и координаты точек для ее задания. Программа хранит все объекты в контейнере указателей на базовый класс Figure (полиморфизм). Во время работы программы находятся центры и площади фигур, в конце программы выводится общая площадь всех фигур.

1. Набор тестов

testcase01

Тест проверяет базовую работу программы и ее способность справляться с ошибками ввода

triangle 0 0 1 1

square 0 0 1 1

rectangle 0 0 1 1

square 0 0 10000 10000

rectangle 0 0 10000 10000

square asdasdaw

triangle 1 1 0 0

triangle -1 -1 0 0

triangle 9 9 10 10

testcase02

Тест проверяет возможность удаления фигур

square 0 0 10000 10000

rectangle 0 0 10000 10000

triangle 0 1 2 4

delete 1

delete 0

square 0 0 1 2

1. Результаты выполнения тестов

Правильность построения фигур и нахождения площадей можно проверить на геометрическом калькуляторе, иногда при выполнении программы происходят ошибки из за неточного представления вещественных чисел в компьютере, рекомендуется округлять double до нескольких знаков после запятой

testcase01

Bad input!

Triangle: [x: 1 y: 1] [x: -1.36603 y: 0.366025] [x: 0.366025 y: -1.36603]

GetCentroid() = [x: -0.000000 y: 0.000000]

GetArea() = 2.598076

Square: [x: 1.000000 y: 1.000000] [x: -1.000000 y: -1.000000]

GetCentroid() = [x: -0.000000 y: 0.000000]

GetArea() = 4.000000

Rectangle: [x: 1.000000 y: 1.000000] [x: -1.000000 y: -1.000000]

GetCentroid() = [x: -0.000000 y: 0.000000]

GetArea() = 4.000000

Square: [x: 10000.000000 y: 10000.000000] [x: -10000.000036 y: -9999.999964]

GetCentroid() = [x: -0.000018 y: 0.000018]

GetArea() = 400000000.000000

Rectangle: [x: 10000.000000 y: 10000.000000] [x: -10000.000036 y: -9999.999964]

GetCentroid() = [x: -0.000018 y: 0.000018]

GetArea() = 400000000.000000

Square: [x: 0.000000 y: 0.000000] [x: 0.000000 y: 0.000000]

GetCentroid() = [x: 0.000000 y: 0.000000]

GetArea() = 0.000000

Total Area: 800000010.598076

Goodbye!

testcase02

Triangle: [x: 2 y: 4] [x: -3.59808 y: 1.23205] [x: 1.59808 y: -2.23205]

GetCentroid() = [x: -0.000000 y: 1.000000]

GetArea() = 16.887495

Square: [x: 1.000000 y: 2.000000] [x: -1.000000 y: -2.000000]

GetCentroid() = [x: -0.000000 y: 0.000000]

GetArea() = 8.000000

Total Area: 24.887495

Goodbye!

1. Листинг программы

#include <iostream>

#include <vector>

#include <cmath>

#include <iomanip>

#define PI 3.14159265

constexpr bool DEBUG = false;

class Point2D {

public:

double x;

double y;

explicit Point2D(double x = 0, double y = 0) : x(x), y(y) {};

friend std::ostream& operator<<(std::ostream& os, const Point2D& d) {

os << "[x: " << d.x << " y: " << d.y << ']';

return os;

}

friend std::istream& operator>>(std::istream& is, Point2D& d) {

is >> d.x >> d.y;

return is;

}

};

class Figure {

public:

static uint64\_t counter;

Figure() : id(counter++) {

if (DEBUG) std::cout << "constructor called for Figure with ID of " << this->id << std::endl;

};

virtual ~Figure() {

if (DEBUG) std::cout << "destructor called for Figure with ID of " << this->id << std::endl;

};

[[nodiscard]] virtual Point2D GetCentroid() const = 0;

[[nodiscard]] virtual double GetArea() const = 0;

[[nodiscard]] std::string GetName() const {

return this->name;

}

virtual void Write(std::ostream& os) const {

os << this->name << ':';

for (auto vertex : this->vertices) {

os << ' ' << vertex;

}

}

friend std::ostream& operator<<(std::ostream& os, Figure& figure) {

figure.Write(os);

return os;

}

virtual void Read(std::istream& is) = 0;

friend std::istream& operator>>(std::istream& is, Figure& mc) {

mc.Read(is);

return is;

}

protected:

std::vector<Point2D> vertices;

std::string name;

private:

const uint64\_t id;

};

uint64\_t Figure::counter = 0;

class Triangle : public Figure {

public:

explicit Triangle(const Point2D& a = Point2D(), const Point2D& b = Point2D(), const Point2D& c = Point2D()) {

this->vertices.assign({ a,b,c });

this->name = "Triangle";

if (DEBUG) std::cout << "constructor called for " << this->name << " with vertices: "

<< vertices.at(0) << ' '

<< vertices.at(1) << ' '

<< vertices.at(2) << std::endl;

}

~Triangle() override {

if (DEBUG) std::cout << "destructor called for " << this->name << " with vertices: "

<< vertices.at(0) << ' '

<< vertices.at(1) << ' '

<< vertices.at(2) << std::endl;

}

void Read(std::istream& is) override {

this->vertices.resize(3);

Point2D center;

is >> center >> vertices[0];

double alpha = 120.0 / 180.0 \* PI;

for (int i = 1; i < 3; ++i) {

vertices[i].x = -sin(alpha) \* (vertices[i - 1].y - center.y) + cos(alpha) \* (vertices[i - 1].x - center.x) + center.x;

vertices[i].y = cos(alpha) \* (vertices[i - 1].y - center.y) + sin(alpha) \* (vertices[i - 1].x - center.x) + center.y;

}

//is >> this->vertices[0] >> this->vertices[1] >> this->vertices[2];

if (is.fail()) {

std::cout << "Bad input!" << std::endl;

}

}

[[nodiscard]] Point2D GetCentroid() const override {

return Point2D((vertices[0].x + vertices[1].x + vertices[2].x) / 3,

(vertices[0].y + vertices[1].y + vertices[2].y) / 3);

}

[[nodiscard]] double GetArea() const override {

double a = (vertices[0].x - vertices[2].x) \* (vertices[1].y - vertices[2].y);

double b = (vertices[0].y - vertices[2].y) \* (vertices[1].x - vertices[2].x);

double c = (a - b) / 2.0;

if (c >= 0) {

return c;

}

else {

return -c;

}

// (0 - 0.866) \* (-0.5 - -0.5) - (1 - -0.5) \* (-0.866 - 0.866);

}

};

class Square : public Figure {

public:

explicit Square(const Point2D& a = Point2D(), const Point2D& b = Point2D()) {

this->vertices.assign({ a,b });

this->name = "Square";

if (DEBUG) std::cout << "constructor called for " << this->name << " with vertices: "

<< vertices.at(0) << ' '

<< vertices.at(1) << std::endl;

}

~Square() override {

if (DEBUG) std::cout << "destructor called for " << this->name << " with vertices: "

<< vertices.at(0) << ' '

<< vertices.at(1) << std::endl;

}

void Read(std::istream& is) override {

this->vertices.resize(2);

Point2D center;

is >> center >> vertices[0];

double alpha = PI;

vertices[1].x = -sin(alpha) \* (vertices[0].y - center.y) + cos(alpha) \* (vertices[0].x - center.x) + center.x;

vertices[1].y = cos(alpha) \* (vertices[0].y - center.y) + sin(alpha) \* (vertices[0].x - center.x) + center.y;

if (is.fail()) {

std::cout << "Bad input!" << std::endl;

}

}

[[nodiscard]] Point2D GetCentroid() const override {

return Point2D(

(vertices[0].x + vertices[1].x) / 2,

(vertices[0].y + vertices[1].y) / 2);

}

[[nodiscard]] double GetArea() const override {

return abs((vertices[0].x - vertices[1].x) \* (vertices[0].y - vertices[1].y));

}

};

class Rectangle : public Figure {

public:

explicit Rectangle(const Point2D& a = Point2D(), const Point2D& b = Point2D()) {

this->vertices.assign({ a,b });

this->name = "Rectangle";

if (DEBUG) std::cout << "constructor called for Rectangle with vertices: "

<< vertices.at(0) << ' '

<< vertices.at(1) << std::endl;

}

~Rectangle() override {

if (DEBUG) std::cout << "destructor called for Rectangle with vertices: "

<< vertices.at(0) << ' '

<< vertices.at(1) << std::endl;

}

void Read(std::istream& is) override {

this->vertices.resize(2);

Point2D center;

is >> center >> vertices[0];

double alpha = PI;

vertices[1].x = -sin(alpha) \* (vertices[0].y - center.y) + cos(alpha) \* (vertices[0].x - center.x) + center.x;

vertices[1].y = cos(alpha) \* (vertices[0].y - center.y) + sin(alpha) \* (vertices[0].x - center.x) + center.y;

if (is.fail()) {

std::cout << "Bad input!" << std::endl;

}

}

[[nodiscard]] Point2D GetCentroid() const override {

return Point2D(

(vertices[0].x + vertices[1].x) / 2,

(vertices[0].y + vertices[1].y) / 2);

}

[[nodiscard]] double GetArea() const override {

return abs((vertices[0].x - vertices[1].x) \* (vertices[0].y - vertices[1].y));

}

};

int main() {

std::vector<Figure\*> v;

std::string figureString;

while (std::cin >> figureString) {

if (figureString == "triangle") {

v.push\_back(new Triangle());

}

else if (figureString == "square") {

v.push\_back(new Square());

}

else if (figureString == "rectangle") {

v.push\_back(new Rectangle());

}

else if (figureString == "delete") {

int index;

std::cin >> index;

delete v[index];

v.erase(v.begin() + index);

continue;

}

else {

std::cout << "I don\'t know this figure :(" << std::endl;

continue;

}

std::cin >> \*(v.back());

}

double totalArea = 0;

for (size\_t i = 0; i < v.size(); ++i) {

std::cout << \*(v[i]) << std::endl;

std::cout << "\tGetCentroid() = " << std::fixed << v[i]->GetCentroid() << std::endl;

double area = v[i]->GetArea();

totalArea += area;

std::cout << "\tGetArea() = " << area << std::endl;

}

std::cout << "Total Area: " << totalArea << std::endl;

for (auto item : v) {

if (item != nullptr) {

delete item;

}

}

std::cout << "Goodbye!" << std::endl;

return 0;

}

1. Выводы

Полиморфизм и Наследование - одни из основных принципов ООП, в языке C++ достигаются с помощью программирования иерархии классов, где одни классы могут наследоваться от других и разделять их поведение, при этом дочерние классы зачастую являются менее абстрактными и более точными. Благодаря полиморфизму в C++ мы можем обрабатывать объекты родственных классов в одном логическом блоке программы благодаря виртуальным методам, и можем реализовывать интерфейсы.

Список литературы

1. C++ Primer (5th Edition) Stanley B. Lippman, Josee Lajoie, E. Moo.

2. Effective Modern C++, Scott Meyers.