**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №2**

**по дисциплине «Объектно-ориентированное программирование»**

**Тема: «Наследование»**

|  |  |
| --- | --- |
| Студент гр. 7381 | Габов Е. С. |
| Преподаватель | Жангиров Т. Р. |

Санкт-Петербург

2019

**Цель работы.**

Ознакомиться с понятиями наследование, полиморфизм, абстрактный класс, изучить виртуальные функции, принцип их работы, способ организации в памяти, раннее и позднее связывания в языке C++. В соответствии с индивидуальным заданием разработать систему классов для представления геометрических фигур.

**Задание.**

Необходимо спроектировать систему классов для моделирования геометрических фигур (в соответствии с полученным индивидуальным заданием). Задание предполагает использование виртуальных функций в иерархии наследования, проектирование и использование абстрактного базового класса. Разработанные классы должны быть наследниками абстрактного класса Shape, содержащего методы для перемещения в указанные координаты, поворота на заданный угол, масштабирования на заданный коэффициент, установки и получения цвета, а также оператор вывода в поток.

Необходимо также обеспечить однозначную идентификацию каждого объекта.

Решение должно содержать:

1. Условие задания;
2. UML диаграмму разработанных классов;
3. Текстовое обоснование проектных решений;
4. Реализацию классов на языке С++.

**Индивидуализация.**

Вариант 13 – реализовать систему классов для фигур:

1. параллелограмм;
2. Эллипс;
3. Сектор эллипса.

**Обоснование проектных решений.**

Для решения данной задачи эллипс, параллелограмм и сектор эллипса задаются схожим образом – 2 диагонали и у параллелограмма дополнительно угол между ними, у сектора эллипса угол развертки. Реализован абстрактный класс Shape. Содержит информацию о 2-х диагоналях, а если некоторой фигуре нужно как-то добавить переменных это можно реализовать непосредственно в классе текущей фигуры. Для идентификации каждой фигуры введена статическая переменная в класса Shape – отслеживает сколько объектов классов создано и исходя из этого унифицирует каждый. Масштабирование, поворот и смена координат реализованы в классе Shape и не переопределяются в классах наследниках. Для добавления некоторой виртуальности – добавлены функции подсчета периметра и площади.

**UML диаграмма разработанных классов.**

UML диаграмма разработанных классов представлена в соседнем документе.

**Выводы.**

В ходе выполнения лабораторной работы была спроектирована система классов для работы с геометрическими фигурами в соответствии с индивидуальным заданием. В иерархии наследования были использованы виртуальные функции, базовый класс при этом является виртуальным (класс называется виртуальным, если содержит хотя бы одну виртуальную функцию). Были реализованы методы перемещения фигуры в заданные координаты, поворота на заданный угол, масштабирования на заданный коэффициент, была реализована однозначная идентификация объекта.

**ПРИЛОЖЕНИЕ А**

1. Shape.h

|  |  |
| --- | --- |
|  | #include <iostream> |
|  | #include <cmath> |
|  | #include <string> |
|  | #include <vector> |
|  |  |
|  | #define PI 3.14159 |
|  |  |
|  | class Point{ |
|  | public: |
|  | int x, y; |
|  | public: |
|  | Point(int x = 0, int y = 0): x(x), y(y){} |
|  | friend std::ostream& operator<<(std::ostream&, const Point&); |
|  | friend class Parallelogram; |
|  | Point& operator=(Point& point){this->x = point.x; this->y = point.y; return \*this;} |
|  | }; |
|  |  |
|  | class Shape{ |
|  | protected: |
|  | static unsigned id; |
|  | static std::string defaultColor; |
|  | Point centre; |
|  | unsigned D1, D2; |
|  | std::string color; |
|  | double angle; |
|  | public: |
|  | void rotate(double newAngle){angle += newAngle;} |
|  | void multiplicateCoordinate(unsigned k){D1\*=k; D2\*=k;} |
|  | void changeCoordinate(Point newCentre){centre = newCentre;} |
|  | void setColor(std::string newColor){color = newColor;} |
|  | virtual double calculatePerimeter() = 0; |
|  | virtual double calculateArea() = 0; |
|  | virtual void print() = 0; |
|  | friend std::ostream& operator<<(std::ostream&, const Shape&); |
|  | }; |
|  |  |
|  | class Parallelogram: public Shape{ |
|  | unsigned id; |
|  | double parAngle; |
|  | public: |
|  | Parallelogram(Point centre, unsigned D1, unsigned D2, std::string color, double newParAngle){ |
|  | angle = 0; // default angle |
|  | id = Shape::id; |
|  | this->centre = centre; |
|  | this->D1 = D1; |
|  | this->D2 = D2; |
|  | this->color = color; |
|  | parAngle = newParAngle; |
|  | } |
|  | void print() override{std::cout << \*this;} |
|  | double calculatePerimeter(); |
|  | double calculateArea(); |
|  | friend std::ostream& operator<<(std::ostream&, const Parallelogram&); |
|  | }; |
|  |  |
|  | class Ellipse: public Shape{ |
|  | unsigned id; |
|  | public: |
|  | Ellipse(Point centre, unsigned D1, unsigned D2, std::string color){ // инициализация через ":" ? |
|  | angle = 0; // default angle |
|  | id = Shape::id; |
|  | this->centre = centre; |
|  | this->D1 = D1; |
|  | this->D2 = D2; |
|  | this->color = color; |
|  | } |
|  | void print() override{std::cout << \*this;} |
|  | double calculatePerimeter(); |
|  | double calculateArea(); |
|  | friend std::ostream& operator<<(std::ostream&, const Ellipse&); |
|  | }; |
|  |  |
|  | class EllipseSector: public Ellipse{ |
|  | unsigned id; |
|  | double sectorAngle; |
|  | public: |
|  | EllipseSector(Point centre, unsigned D1, unsigned D2, std::string color, double newSectorAngle): |
|  | Ellipse(centre, D1, D2, color), sectorAngle(newSectorAngle){ |
|  | id = Shape::id; |
|  | } |
|  | void print() override{std::cout << \*this;} |
|  | double calculatePerimeter(); |
|  | double calculateArea(); |
|  | friend std::ostream& operator<<(std::ostream&, const EllipseSector&); |
|  | }; |
|  |  |
|  |  |

1. Shape.cpp

|  |  |
| --- | --- |
| #include "shape.h" |  |
|  |  |
|  | unsigned Shape::id = 0; |
|  | std::string Shape::defaultColor = "white"; |
|  |  |
|  | std::ostream& operator<<(std::ostream& out, const Point& point){ |
|  | out << "X: " << point.x << " Y: " << point.y; |
|  | return out; |
|  | } |
|  | std::ostream& operator<<(std::ostream& out, const Parallelogram& par){ |
|  | out << "PARALLELOGRAM" << std::endl; |
|  | out << "CENTRE: " << par.centre << std::endl << |
|  | "D1: " << par.D1 << std::endl << |
|  | "D2: " << par.D2 << std::endl << |
|  | "COLOR: " << par.color << std::endl << |
|  | "ANGLE: " << par.angle << std::endl << |
|  | "PAR ANGLE: " << par.parAngle << std::endl; |
|  | return out; |
|  | } |
|  |  |
|  | std::ostream& operator<<(std::ostream& out, const Ellipse& el){ |
|  | out << "ELLIPSE" << std::endl; |
|  | out << "CENTRE: " << el.centre << std::endl << |
|  | "D1: " << el.D1 << std::endl << |
|  | "D2: " << el.D2 << std::endl << |
|  | "COLOR: " << el.color << std::endl << |
|  | "ANGLE: " << el.angle << std::endl; |
|  | return out; |
|  | } |
|  |  |
|  | std::ostream& operator<<(std::ostream& out, const EllipseSector& elS){ |
|  | out << "ELLIPSESECTOR" << std::endl; |
|  | out << "CENTRE: " << elS.centre << std::endl << |
|  | "D1: " << elS.D1 << std::endl << |
|  | "D2: " << elS.D2 << std::endl << |
|  | "COLOR: " << elS.color << std::endl << |
|  | "ANGLE: " << elS.angle << std::endl << |
|  | "SECTOR ANGLE: " << elS.sectorAngle << std::endl; |
|  | return out; |
|  | } |
|  |  |
|  | double Parallelogram::calculatePerimeter(){ |
|  | return 2 \* sqrt(pow(D1, 2) + pow(D2, 2) - 2 \* D1 \* D2 \* std::cos(parAngle \* PI / 180)) + |
|  | 2 \* sqrt(pow(D1, 2) + pow(D2, 2) - 2 \* D1 \* D2 \* std::cos((180 - parAngle) \* PI / 180)); |
|  | } |
|  |  |
|  | double Parallelogram::calculateArea(){ |
|  | return 0.5 \* D1 \* D2 \* std::sin(parAngle \* PI / 180); |
|  | } |
|  |  |
|  | double Ellipse::calculatePerimeter(){ |
|  | return 4 \* ((PI \* D1 \* D2 + (D1 - D2)) / (D1 + D2)); |
|  | } |
|  |  |
|  | double Ellipse::calculateArea(){ |
|  | return D1 \* D2 \* PI; |
|  | } |
|  |  |
|  | double EllipseSector::calculateArea(){ |
|  | return Ellipse::calculateArea() \* sectorAngle / 180; |
|  | } |
|  |  |
|  | double EllipseSector::calculatePerimeter(){ |
|  | return Ellipse::calculatePerimeter() \* sectorAngle / 180; |
|  | } |