**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра Вычислительной техники**

отчет

**по лабораторной работе № 1**

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

Тема: **«РАБОТА С ИЕРАРХИЕЙ ОБЪЕКТОВ:  
НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3311 | Аршин А.Д |  |
| Преподаватель | Манирагена В. |  |

Са

Санкт-Петербург

2024

**Введение**

**Цель работы:**

демонстрация работы с геометрическими фигурами (лицо, шляпа, бакенбарды) с использованием библиотеки shape.h. Код создает составные объекты, выполняет их трансформацию (поворот, масштабирование, перемещение) и визуализирует результат на текстовом экране, показывая процесс сборки изображения из отдельных фигур. Особенности реализации:

1. На шляпе отображается косой крест (реализовано в классе hat\_with\_emblem)
2. Бакенбарды представлены в виде параллелограммов (реализовано в классе myshape)

**Постановка задачи:**

Разработать программу для работы с составными геометрическими фигурами, которая должна:

1. **Реализовать базовые примитивы (линии, прямоугольники) и составные фигуры:**

Лицо с глазами, ртом и бакенбардами

Шляпу с эмблемой в виде косого креста

Поля шляпы

1. **Обеспечить функциональность для манипуляции фигурами:**

Перемещение

Масштабирование

Поворот

Отражение

1. **Визуализировать фигуры на текстовом экране с учетом:**

Отображения косого креста на шляпе

Использования параллелограммов для бакенбард

1. **Организовать процесс сборки изображения из отдельных фигур:**

Позиционирование шляпы относительно лица

Выравнивание полей шляпы

Центрирование составных элементов

1. **Предоставить пошаговую демонстрацию:**

Исходного набора фигур

Результата трансформаций

Финального изображения

**Добавление новых классов**

**Для реализации требуемой функциональности были добавлены следующие классы:**

1. **Класс hat\_with\_emblem (шляпа с эмблемой):**

Наследует от класса rectangle

**Добавлен метод draw(), который:**

Рисует контур шляпы (вызов базового метода rectangle::draw())

**Добавляет косой крест внутри шляпы:**

Рассчитывает центр шляпы

Рисует две диагональные линии, образующие крест оптимизирует длину линий креста относительно размеров шляпы

1. **Класс parallelogram (параллелограмм):**

Наследует от базового класса shape

**Реализует все необходимые методы для работы с фигурой:**

Конструктор, принимающий три точки и рассчитывающий четвертую

Метод draw() для отрисовки четырех сторон

Методы для работы с якорными точками (north, south, east, west и др.)

Методы move() и resize() для трансформации фигуры

**Используется для реализации бакенбард в классе myshape**

**Эти классы были интегрированы в существующую систему:**

Параллелограммы используются для создания бакенбард в классе myshape

Шляпа с эмблемой добавляется как отдельный элемент композиции

Обе фигуры поддерживают все базовые операции (перемещение, масштабирование)

**Какие функции-члены пришлось переопределить и почему**

В добавленных классах были переопределены следующие функции-члены:

1. **В классе hat\_with\_emblem:**

**draw():**

Причина: необходимо добавить отрисовку косого креста поверх базового прямоугольника

Реализация: сначала вызывается rectangle::draw(), затем рисуются две диагональные линии

1. **В классе parallelogram:**

**draw():**

Причина: необходимо отрисовать четыре стороны параллелограмма вместо прямоугольника

Реализация: рисуются четыре линии, соединяющие вершины параллелограмма

**north(), south(), east(), west():**

Причина: Якорные точки для параллелограмма рассчитываются иначе, чем для прямоугольника

Реализация: используются координаты вершин и их средние значения

**move():**

Причина: необходимо перемещать все четыре вершины параллелограмма

Реализация: к координатам каждой вершины добавляются смещения

**resize():**

Причина: Масштабирование должно учитывать центр параллелограмма

Реализация: Вершины перемещаются относительно центра с учетом коэффициента масштабирования

Эти переопределения были необходимы для корректной работы с новыми типами фигур и поддержания полиморфного поведения в рамках существующей системы.

**Какие функции-члены сделаны недоступными и каким образом это осуществлено**

В коде были ограничены доступ к следующим функциям-членам:

1. **В классе hat\_with\_emblem:**

**rotate\_left() и rotate\_right() не реализованы (пустые функции)**

Причина: Поворот шляпы с эмблемой может нарушить правильное отображение косого креста

Реализация: Функции оставлены пустыми, чтобы предотвратить некорректное поведение

Альтернатива: Можно было бы сделать функции private, но оставлены public для совместимости с интерфейсом

1. **В классе myshape:**

**rotate\_left() и rotate\_right() не реализованы (пустые функции)**

Причина: Поворот составной фигуры (лица) может нарушить правильное расположение глаз, рта и бакенбард

Реализация: Функции оставлены пустыми

Альтернатива: можно было бы реализовать поворот всех компонентов, но это усложнило бы код

1. **В классе shape (базовый класс):**

**Конструктор копирования и оператор присваивания не реализованы**

Причина: предотвратить нежелательное копирование фигур, которое может привести к проблемам с управлением памятью

Реализация: используется правило "пяти" (C++11) - явное удаление этих функций

Пример: shape (const shape&) = delete;

**Эти ограничения были введены для:**

1. Предотвращения некорректного поведения при операциях, которые могут нарушить целостность составных фигур
2. Упрощения управления объектами и предотвращения ошибок, связанных с копированием
3. Сохранения семантики работы с фигурами в рамках заданной задачи

**Код добавлений в Shape.h**

class hat\_with\_emblem : public rectangle {

    public:

        // Конструктор шляпы: a - нижний левый угол, b - верхний правый угол

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

                : rectangle(a, b) { }

        // Отрисовка шляпы и её эмблема

        void draw() override {

            // Рисуем контур шляпы

            rectangle::draw();

            // Рисуем косой крест внутри шляпы с правильным наклоном

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

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

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

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

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

        }

    };

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

    point n = q.north(); // Получаем северную точку фигуры q

    point s = p.south(); // Получаем южную точку фигуры p

    p.move(0, n.y - s.y); // Перемещаем p над q

}

class parallelogram : public shape {

    point p1, p2, p3, p4; // Добавляем 4-ю точку

public:

    parallelogram(point a, point b, point c)

        : p1(a), p2(b), p3(c) {

        // Рассчитываем 4-ю точку для параллелограмма

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

    }

    void draw() override {

        put\_line(p1, p2);

        put\_line(p2, p3);

        put\_line(p3, p4);

        put\_line(p4, p1);

    }

    point north() const override {

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

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

        return point(x, y);

    }

    point south() const override {

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

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

        return point(x, y);

    }

    point east() const override {

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

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

        return point(x, y);

    }

    point west() const override {

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

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

        return point(x, y);

    }

    point neast() const override {

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

    }

    point nwest() const override {

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

    }

    point swest() const override {

        return point(std::min(p1.x, p4.x), std::max(p1.y, p4.y));

    }

    point seast() const override {

        return point(std::max(p2.x, p3.x), std::max(p2.y, p3.y));

    }

    void move(int dx, int dy) override {

        p1.x += dx; p1.y += dy;

        p2.x += dx; p2.y += dy;

        p3.x += dx; p3.y += dy;

        p4.x += dx; p4.y += dy;

    }

    void resize(double scale) override {

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

        p1.x = center.x + (p1.x - center.x) \* scale;

        p1.y = center.y + (p1.y - center.y) \* scale;

        p2.x = center.x + (p2.x - center.x) \* scale;

        p2.y = center.y + (p2.y - center.y) \* scale;

        p3.x = center.x + (p3.x - center.x) \* scale;

        p3.y = center.y + (p3.y - center.y) \* scale;

        p4.x = center.x + (p4.x - center.x) \* scale;

        p4.y = center.y + (p4.y - center.y) \* scale;

    }

};

**Контрольные вопросы**

1. **Какой базовый класс лучше всего использовать для производного класса «треугольник»?**

Ответ: лучше использовать класс shape, так как треугольник — это базовая геометрическая фигура.

1. **То же — для класса «кружок».**

Ответ: также класс shape, так как круг — это базовая геометрическая фигура.

1. **То же — для класса «крестик».**

Ответ: можно использовать класс shape или line, так как крестик состоит из пересекающихся линий.

1. **Какой тип наследования следует выбрать: private, public или protected?**

Ответ: для иерархии фигур следует использовать public наследование, чтобы сохранить интерфейс базового класса.

1. **Можно ли вообще не указывать тип наследования?**

Ответ: нет, в C++ тип наследования должен быть указан явно. По умолчанию для классов используется private наследование.

1. **В чём смысл объявления функций в базовом классе как виртуальных?**

Ответ: Виртуальные функции позволяют реализовать полиморфизм - возможность переопределения функций в производных классах.

1. **Нужно ли объявлять виртуальной функцию в производном классе, если в базовом она уже объявлена таковой?**

Ответ: нет, достаточно объявить функцию в базовом классе как виртуальную. В производных классах она автоматически будет виртуальной.

1. **Можно ли потребовать от компилятора проверить корректность объявления виртуальной функции в производном классе и как это сделать?**

Ответ: да, с помощью ключевого слова override. Оно указывает, что функция должна переопределять виртуальную функцию из базового класса.

1. **Можно ли запретить виртуальную функцию в классах-наследниках?**

Ответ: да, с помощью ключевого слова final. Оно запрещает дальнейшее переопределение функции в производных классах.

1. **Что такое «чисто виртуальная функция»?**

Ответ: это виртуальная функция, которая не имеет реализации в базовом классе и должна быть обязательно переопределена в производных классах. Обозначается как "= 0".

1. **Обязательно ли переопределять все функции-члены базового класса в производном классе?**

Ответ: нет, только те, которые необходимы для конкретного производного класса. Чисто виртуальные функции должны быть переопределены обязательно.

1. **Зачем может понадобиться создание набора (массива или списка) указателей на разные типы объектов в пределах некоторой иерархии?**

Ответ: для работы с объектами разных типов через единый интерфейс базового класса, что позволяет реализовать полиморфное поведение.

1. **Как запретить для объекта вызов конструктора по умолчанию?** Ответ: объявить конструктор по умолчанию как private или использовать ключевое слово delete: ClassName() = delete;
2. **Как запретить вызов конструктора для использования в качестве неявного преобразователя типа?**

Ответ: использовать ключевое слово explicit перед конструктором.

1. **Каким образом можно установить значение переменных объекта, объявленных с модификаторами const?**

Ответ: через список инициализации в конструкторе.

1. **Каким образом следует инициализировать объект базового класса в конструкторе производного класса? Всегда ли это нужно делать?** Ответ: через список инициализации. Это нужно делать, если базовый класс не имеет конструктора по умолчанию или требуется инициализация с параметрами.
2. **Каким требованиям должны удовлетворять классы, чтобы между ними можно было сформировать отношение «является»?**

Ответ: Производный класс должен расширять или специализировать функциональность базового класса, сохраняя его интерфейс.

1. **Как установить между двумя классами отношение «содержит»?** Ответ: включить объект одного класса как член данных в другой класс.
2. **Каким образом на экране появляются глаза и рот, если функция myshape ∷ draw () не содержит кода для выдачи этих элементов фигуры?**

Ответ: они отрисовываются через вызов draw() для соответствующих объектов-членов (l\_eye, r\_eye, mouth).

1. **Почему данные в классах line и rectangle сделаны protected?**

Ответ:чтобы производные классы могли напрямую обращаться к данным базового класса, сохраняя инкапсуляцию для внешнего кода.

**Заключение**

Данный Лабораторная представляет собой интересный пример применения объектно-ориентированного программирования для работы с графическими примитивами.

Он демонстрирует:

1. Гибкость наследования - возможность создавать сложные составные объекты (лицо, шляпа) из простых фигур (линии, прямоугольники, параллелограммы).
2. Мощь полиморфизма - единый интерфейс для работы с различными типами фигур, что позволяет легко расширять функциональность.
3. Практическое применение принципов ООП:

Инкапсуляция: скрытие деталей реализации

Наследование: создание иерархии классов

Полиморфизм: единый интерфейс для разных типов

1. Реализацию паттерна "Компоновщик" - работа с составными объектами как с единым целым.

Этот код может служить отличной базой для создания более сложных графических редакторов или игр, демонстрируя, как грамотное применение ООП позволяет создавать гибкие и расширяемые системы.