Министерство науки и высшего образования Российской Федерации Федеральное государственное автономное учреждение высшего образования «Пермский национальный исследовательский политехнический университет»

ПНИПУ

**Лабораторная работа №4**

**«Простое наследование. Принцип подстановки»**

**14 вариант**

Выполнил:

Студент группы РИС-23-1б

Шароглазов Егор Алексеевич

Проверила:

Доцент кафедры ИТАС

О.А. Полякова

1. г

**Постановка задачи:**

1. Определить пользовательский класс.

2.Определить в классе следующие конструкторы: без параметров, с параметрами, копирования.

3. Определить в классе деструктор.

4. Определить в классе компоненты-функции для просмотра и установки полей данных (селекторы и модификаторы).

5. Перегрузить операцию присваивания.

6. Перегрузить операции ввода и вывода объектов с помощью потоков.

7. Определить производный класс.

8. Написать программу, в которой продемонстрировать создание объектов и работу всех перегруженных операций.

9. Реализовать функции, получающие и возвращающие объект базового класса. Продемонстрировать принцип подстановки.

**Анализ задачи:**

**Создание объектов:**

Создаются объекты person1, person2, student1, student2 с разными способами инициализации.

**Ввод/вывод:** Используются операторы >> и << для ввода данных для person2 и student2 и вывода информации о всех объектах.

**Проверка оценки:** Вызывается метод checkMark() для проверки оценок студентов.

**Принцип подстановки:**

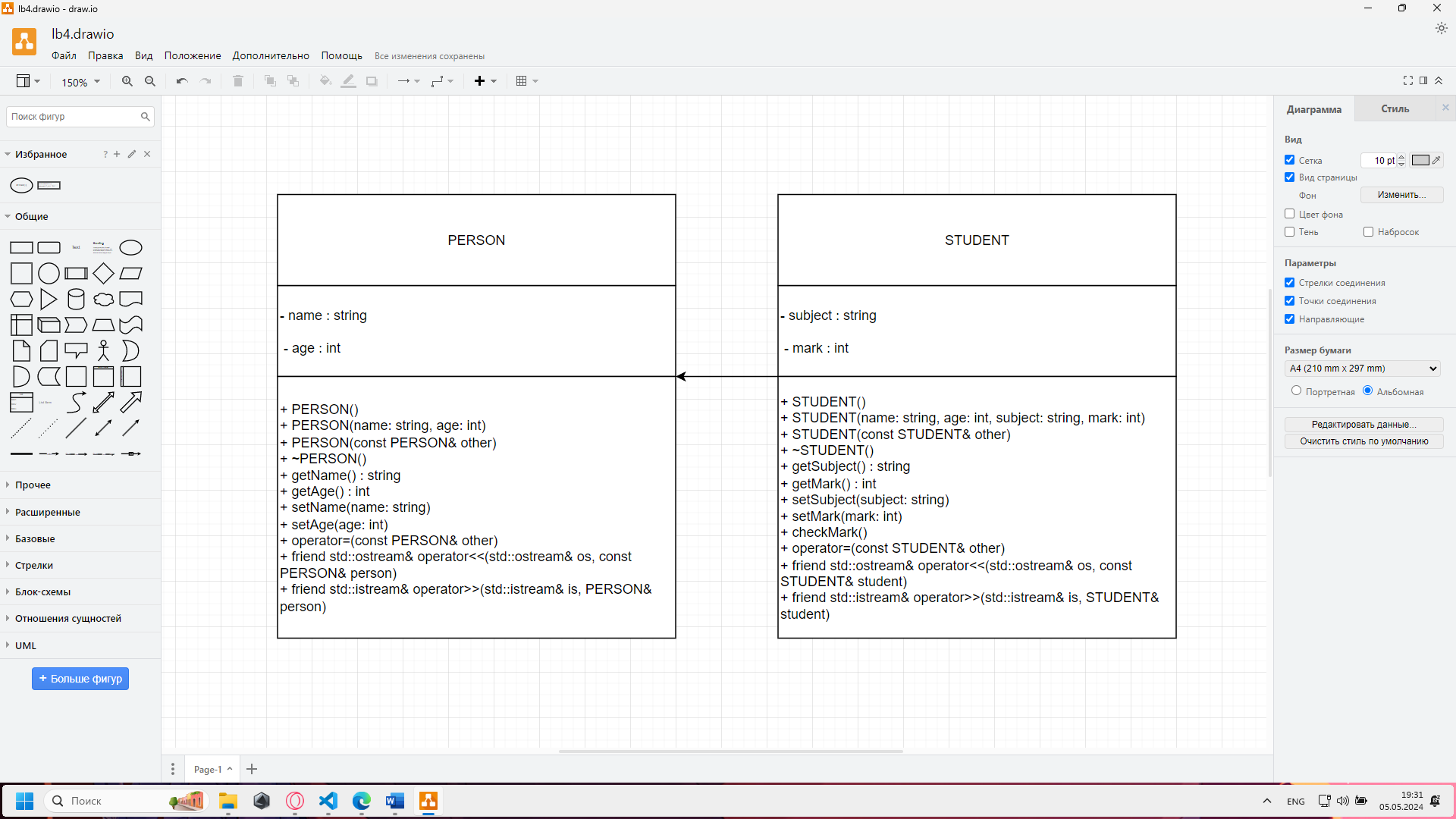
Функция printPersonInfo принимает объект PERSON и выводит его информацию.

В неё передаются как объект person1 (базовый класс), так и student1 (производный класс), демонстрируя принцип подстановки.

Функция createPerson создаёт и возвращает объект PERSON, демонстрируя возможность использовать возвращаемый объект производного класса в контексте базового.

Вывод: Программа демонстрирует основные принципы ООП, позволяющие создавать гибкий и расширяемый код.

**UML – Диаграмма**



**Код:**

#include <iostream>

#include <string>

class PERSON {

private:

    std::string name;

    int age;

public:

    // Конструкторы

    PERSON() : name(""), age(0) {}

    PERSON(const std::string& name, int age) : name(name), age(age) {}

    PERSON(const PERSON& other) : name(other.name), age(other.age) {}

    // Деструктор

    ~PERSON() {}

    // Геттеры и сеттеры

    std::string getName() const { return name; }

    int getAge() const { return age; }

    void setName(const std::string& name) { this->name = name; }

    void setAge(int age) { this->age = age; }

    // Перегрузка оператора присваивания

    PERSON& operator=(const PERSON& other) {

        if (this != &other) {

            name = other.name;

            age = other.age;

        }

        return \*this;

    }

    // Перегрузка операторов ввода/вывода

    friend std::ostream& operator<<(std::ostream& os, const PERSON& person) {

        os << "Имя: " << person.name << ", Возраст: " << person.age;

        return os;

    }

    friend std::istream& operator>>(std::istream& is, PERSON& person) {

        std::cout << "Введите имя: ";

        is >> person.name;

        std::cout << "Введите возраст: ";

        is >> person.age;

        return is;

    }

};

// Производный класс STUDENT

class STUDENT : public PERSON {

private:

    std::string subject;

    int mark;

public:

    // Конструкторы

    STUDENT() : PERSON(), subject(""), mark(0) {}

    STUDENT(const std::string& name, int age, const std::string& subject, int mark)

        : PERSON(name, age), subject(subject), mark(mark) {}

    STUDENT(const STUDENT& other) : PERSON(other), subject(other.subject), mark(other.mark) {}

    // Деструктор

    ~STUDENT() {}

    // Геттеры и сеттеры

    std::string getSubject() const { return subject; }

    int getMark() const { return mark; }

    void setSubject(const std::string& subject) { this->subject = subject; }

    void setMark(int mark) { this->mark = mark; }

    // Проверка оценки

    void checkMark() {

        if (mark < 4) {

            std::cout << "Неудовлетворительная оценка " << getName() << " по " << subject << std::endl;

        }

    }

    // Перегрузка оператора присваивания

    STUDENT& operator=(const STUDENT& other) {

        if (this != &other) {

            PERSON::operator=(other);

            subject = other.subject;

            mark = other.mark;

        }

        return \*this;

    }

    // Перегрузка операторов ввода/вывода

    friend std::ostream& operator<<(std::ostream& os, const STUDENT& student) {

        os << static\_cast<const PERSON&>(student) << ", Предмет: " << student.subject

           << ", Оценка: " << student.mark;

        return os;

    }

    friend std::istream& operator>>(std::istream& is, STUDENT& student) {

        is >> static\_cast<PERSON&>(student);

        std::cout << "Введите предмет: ";

        is >> student.subject;

        std::cout << "Введите оценку: ";

        is >> student.mark;

        return is;

    }

};

// Функции для демонстрации принципа подстановки

void printPersonInfo(const PERSON& person) {

    std::cout << person << std::endl;

}

PERSON createPerson() {

    std::string name;

    int age;

    std::cout << "Введите имя: ";

    std::cin >> name;

    std::cout << "Введите возраст: ";

    std::cin >> age;

    return PERSON(name, age);

}

int main() {

    PERSON person1("Иван", 30);

    PERSON person2;

    std::cin >> person2;

    STUDENT student1("Петр", 20, "Математика", 5);

    STUDENT student2;

    std::cin >> student2;

    std::cout << "Person 1: " << person1 << std::endl;

    std::cout << "Person 2: " << person2 << std::endl;

    std::cout << "Student 1: " << student1 << std::endl;

    std::cout << "Student 2: " << student2 << std::endl;

    student1.checkMark();

    student2.checkMark();

    // Принцип подстановки

    printPersonInfo(person1);

    printPersonInfo(student1);

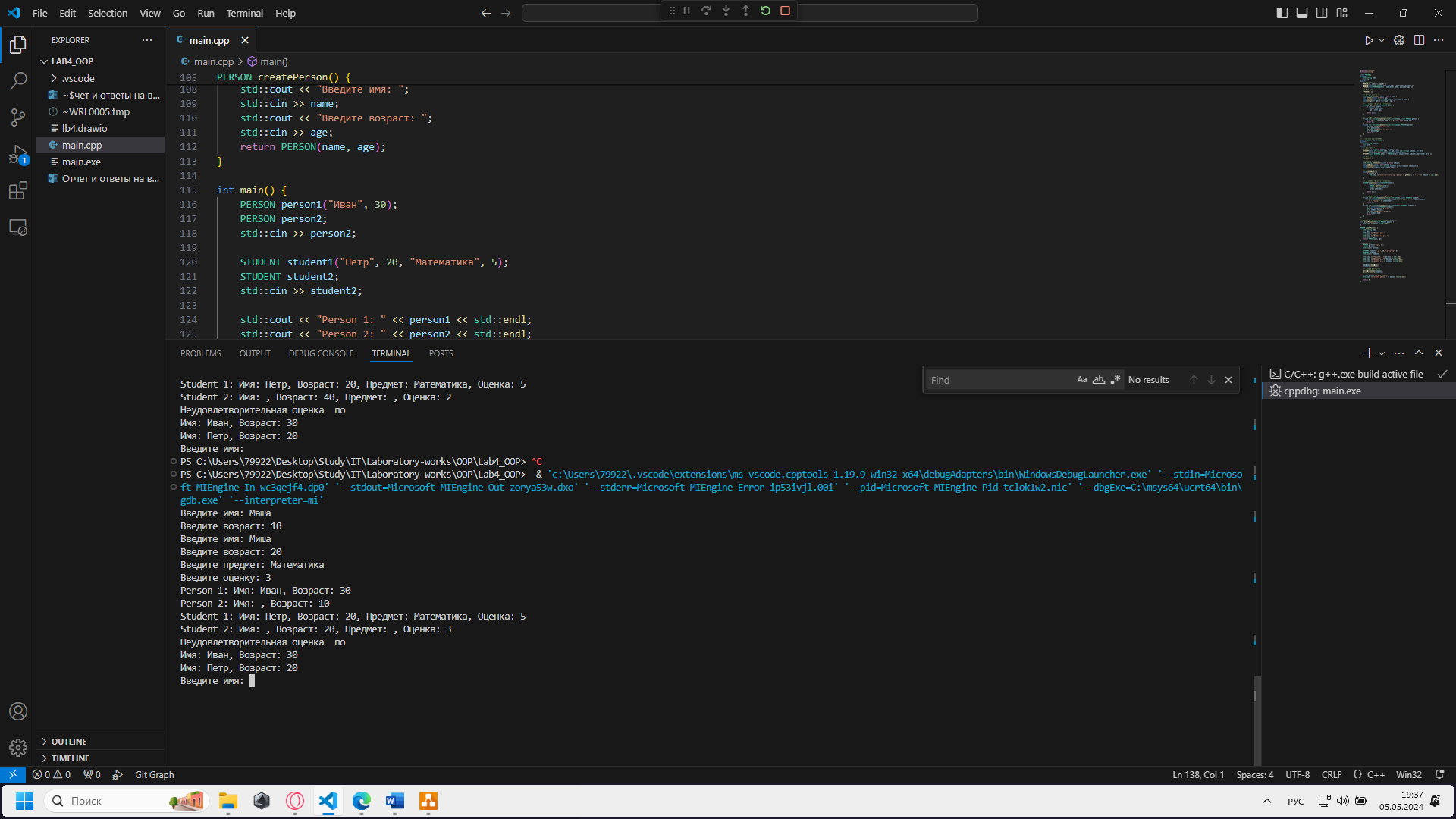
    PERSON person3 = createPerson();

    std::cout << "Created person: " << person3 << std::endl;

    return 0;

}

**Вывод программы:**



**Вопросы**

· **Использование механизма наследования**:

* Механизм наследования в языке программирования C++ позволяет создавать новые классы на основе уже существующих. Производный класс получает доступ к членам базового класса, что способствует повторному использованию кода и упрощает разработку и поддержку программ.

· **Наследование компонентов класса с public-спецификатором**:

* Компоненты класса, описанные со спецификатором public, наследуются в производном классе с тем же самым уровнем доступа.

· **Наследование компонентов класса с private-спецификатором**:

* Компоненты класса, описанные со спецификатором private, наследуются в производном классе как private, то есть они не доступны напрямую в производном классе.

· **Наследование компонентов класса с protected-спецификатором**:

* Компоненты класса, описанные со спецификатором protected, наследуются в производном классе с тем же самым уровнем доступа.

· **Описание производного класса**:

* Производный класс описывается с использованием ключевого слова class или struct, за которым следует имя класса, а затем двоеточие и ключевое слово public, за которым указывается имя базового класса.

· **Наследование конструкторов**:

* Конструкторы не наследуются автоматически. Они могут вызывать конструкторы базового класса в списке инициализации или через конструктор копирования.

· **Наследование деструкторов**:

* Деструкторы не наследуются автоматически. Они вызываются в обратном порядке, начиная с деструктора производного класса и заканчивая деструктором базового класса.

· **Порядок конструирования объектов производных классов**:

* При создании объекта производного класса сначала вызывается конструктор базового класса, затем конструктор самого производного класса.

· **Порядок уничтожения объектов производных классов**:

* При уничтожении объекта производного класса сначала вызывается деструктор производного класса, а затем деструктор базового класса.

· **Виртуальные функции и позднее связывание**:

* Виртуальные функции позволяют реализовывать полиморфизм, а механизм позднего связывания (также известный как динамическое связывание) обеспечивает вызов подходящей реализации виртуальной функции на основе типа объекта во время выполнения программы.

· **Виртуальные конструкторы и деструкторы**:

* В C++ конструкторы не могут быть виртуальными. Деструкторы могут быть виртуальными, что полезно при работе с полиморфными объектами, где используется оператор delete.

· **Наследуется ли спецификатор virtual**:

* Спецификатор virtual не наследуется.

· **Открытое наследование**:

* Открытое наследование устанавливает отношение "является" между классами, что означает, что производный класс является типом базового класса.

· **Закрытое наследование**:

* Закрытое наследование устанавливает отношение "реализуется в терминах" между классами, что означает, что производный класс реализует интерфейс базового класса.

· **Принцип подстановки**:

* Принцип подстановки (или принцип Лисков) гласит, что объекты производного класса могут быть использованы везде, где используются объекты базового класса, не нарушая функциональность программы.

· **Компонентные данные объекта x**:

* Объект x будет иметь все компонентные данные, которые определены в классах Student, Employee и Teacher, включая age, name, post и stage.

· **Конструкторы без параметров**:

· Student() : age(0), name("") {}

Employee() : Student(), post("") {}

Teacher() : Employee(), stage(0) {}

· **Конструкторы с параметрами**:

· Student(int a, const string& n) : age(a), name(n) {}

Employee(int a, const string& n, const string& p) : Student(a, n), post(p) {}

Teacher(int a, const string& n, const string& p, int s) : Employee(a, n, p), stage(s) {}

· **Конструкторы копирования**:

· Student(const Student& s) : age(s.age), name(s.name) {}

Employee(const Employee& e) : Student(e), post(e.post) {}

Teacher(const Teacher& t) : Employee(t), stage(t.stage) {}

· **Операции присваивания**:

Student& operator=(const Student& s) { age = s.age; name = s.name; return \*this; }

Employee& operator=(const Employee& e) { static\_cast<Student&>(\*this) = e; post = e.post; return \*this; }

Teacher& operator=(const Teacher& t) { static\_cast<Employee&>(\*this) = t; stage = t.st