Министерство науки и высшего образования Российской Федерации Федеральное государственное автономное образовательное учреждение высшего образования

**«Пермский национальный исследовательский политехнический университет»**

Электротехнический факультет

Кафедра «Информационные технологии и автоматизированные системы» направление подготовки: 09.03.04 – «Программная инженерия»

**Лабораторная работа №5.**

**«Наследование. Виртуальные функции. Полиморфизм.»**

Выполнил студент гр. РИС-24-1б

Чижов Денис Николаевич

Проверил:

Доц. Каф. ИТАС 

Ольга Андреевна Полякова

(оценка) (подпись)

(дата)

г. Пермь, 2025

**Задача:**

1. Определить абстрактный класс.

2. Определить иерархию классов, в основе которой будет находиться абстрактный класс (см. лабораторную работу №4).

3. Определить класс Вектор, элементами которого будут указатели на объекты иерархии классов.

4. Перегрузить для класса Вектор операцию вывода объектов с помощью потоков.

5. В основной функции продемонстрировать перегруженные операции и полиморфизм Вектора.

Базовый класс:

ТРОЙКА\_ЧИСЕЛ (TRIAD)

Первое\_число (first) - int Второе\_число (second) – int Третье\_число (third) - int

Определить методы изменения полей и сравнения триады.

Создать производный класс DATE с полями год, месяц и число. Определить полный набор операций сравнения дат.

**Анализ:**

**1. Абстрактный базовый класс Object**

* **Роль:**
  + Определяет интерфейс для всех производных классов.
  + Содержит **чисто виртуальную функцию** Show(), делая класс абстрактным.

## **2. Иерархия классов (**Triad **→** Date**)**

### ****Класс**** Triad ****(производный от**** Object****)****

* **Поля:**
  + first, second, third (целые числа).
* **Методы:**
  + Конструкторы (по умолчанию, с параметрами, копирования).
  + Селекторы/модификаторы (get\_first(), set\_second() и т. д.).
  + Перегрузка операторов (=, <, >, ==, >>, <<).
  + Виртуальный метод Show() (реализация абстрактного метода).

### ****Класс**** Date ****(производный от**** Triad****)****

* **Поля:**
  + day, month, year (дублируют first, second, third).
* **Методы:**
  + Конструкторы, деструктор.
  + Перегрузка операторов (<, >, ==, >>, <<).
  + Виртуальный метод Show() (выводит дату).

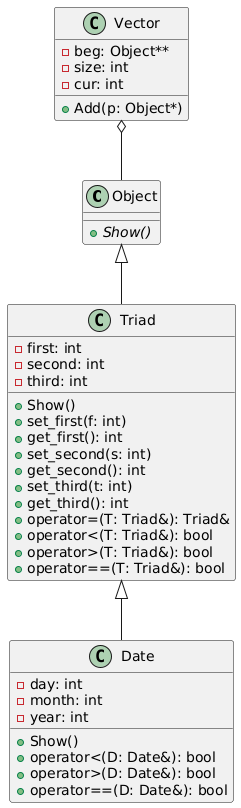
## **3. Класс** Vector **(контейнер объектов)**

* **Роль:**
  + Хранит массив указателей на Object (Object\*\* beg).
  + Поддерживает добавление (Add()) и вывод (operator<<).
* **Методы:**
  + Конструкторы (по умолчанию, с заданным размером).
  + Деструктор (освобождает память).
  + Перегрузка operator<< для вывода объектов.

## **4. Демонстрация в** main()

* **Что проверяется:**
  1. Создание объектов Triad и Date.
  2. Добавление их в Vector через указатель на Object.
  3. Вывод с использованием полиморфизма (Show() вызывается корректно).

**UML диаграмма:**



**Код:**

**Object.h:**

#pragma once

using namespace std;

class Object

{

public:

Object(void){};

~Object(void) {};

virtual void Show() = 0;//чисто виртуальная функция

};

**Triad.h:**

#pragma once

#include "Object.h"

#include <iostream>

using namespace std;

class Triad:public Object {

protected:

int first;

int second;

int third;

public:

//Конструкторы

Triad() { first = 0; second = 0; third = 0; }; //Конструктор без параметров

Triad(int f, int s, int t) { first = f; second = s; third = t; };//Конструктор с параметрами

Triad(const Triad& T) { first = T.first; second = T.second; third = T.third; }; //Конструктор копирования

//Деструктор

~Triad() {};

//

void Show();

//Селекторы и модификаторы

void set\_first(int f) { first = f; }; //модификатор first

int get\_first() { return first; }; //селектор first

void set\_second(int s) { second = s; }; //модификатор second

int get\_second() { return second; }; //селектор second

void set\_third(int t) { third = t; }; //модификатор second

int get\_third() { return third; }; //селектор second

//Перегрузки операторов

Triad& operator=(const Triad&);

bool operator<(const Triad&);

bool operator>(const Triad&);

bool operator==(const Triad&);

//Глобальные функции ввода-вывода

friend istream& operator>>(istream& in, Triad& T);

friend ostream& operator<<(ostream& out, const Triad& T);

};

**Triad.cpp:**

#include "Triad.h"

#include <iostream>

void Triad::Show()

{

cout << "\nFirst : " << first;

cout << "\nSecond : " << second;

cout << "\nThird : " << third;

cout << "\n";

}

Triad& Triad::operator=(const Triad& T)

{

//проверка на самоприсваивание

if (&T == this) return\*this;

first = T.first;

second = T.second;

third = T.third;

return \*this;

}

bool Triad::operator<(const Triad& T)

{

int f = first + second + third;

int s = T.first + T.second + T.third;

if (f < s) { return true; }

else { return false; }

}

bool Triad::operator>(const Triad& T)

{

int f = first + second + third;

int s = T.first + T.second + T.third;

if (f > s) { return true; }

else { return false; }

}

bool Triad::operator==(const Triad& T)

{

int f = first + second + third;

int s = T.first + T.second + T.third;

if (f == s) { return true; }

else { return false; }

}

istream& operator>>(istream& in, Triad& T) {

in >> T.first;

in >> T.second;

in >> T.third;

cout << endl;

return in;

}

ostream& operator<<(ostream& out, const Triad& T) {

return (out << T.first << " , " << T.second << " , " << T.third << endl);

}

**Date.h:**

#pragma once

#include "Triad.h"

class Date : public Triad{

public:

Date() { day = first; month = second; year = third; }; //Конструктор без параметров

Date(int f, int s, int t) { day = f; month = s; year = t; };

Date(const Date& D) { first = D.first; second = D.second; third = D.third; day = D.day; month = D.month; year = D.year;};

~Date() {}; //Деструктор

bool operator<(const Date&);

bool operator>(const Date&);

bool operator==(const Date&);

void Show();

friend istream& operator>>(istream& in, Date& D);

friend ostream& operator<<(ostream& out, const Date& D);

private:

int day;

int month;

int year;

};

**Date.cpp:**

#include "Date.h"

#include <iostream>

void Date::Show()

{

cout << "\nFirst : " << first;

cout << "\nSecond : " << second;

cout << "\nThird : " << third;

cout << "\nDay : " << day;

cout << "\nMonth : " << month;

cout << "\nYear : " << year;

cout << "\n";

}

istream& operator>>(istream& in, Date& D) {

cout << "Введите день:";

in >> D.first;

D.day= D.first;

cout << "Введите месяц:";

in >> D.second;

D.month = D.second;

cout << "Введите год:";

in >> D.third;

D.year = D.third;

cout << endl;

return in;

}

ostream& operator<<(ostream& out, const Date& D) {

return (out << D.day << '.' << D.month << '.' << D.year << endl);

}

bool Date::operator<(const Date& D) {

if (year < D.year) { return true; }

else {

if (month < D.month) { return true; }

else {

if (day < D.day) { return true; }

else { return false; }

}

}

}

bool Date::operator>(const Date& D) {

if (year > D.year) { return true; }

else {

if (month > D.month) { return true; }

else {

if (day > D.day) { return true; }

else { return false; }

}

}

}

bool Date::operator==(const Date& D) {

if (year == D.year) { return true; }

else {

if (month == D.month) { return true; }

else {

if (day == D.day) { return true; }

else { return false; }

}

}

}

**Vector.h:**

#pragma once

#include "object.h"

#include <string>

#include <iostream>

using namespace std; class Vector

{

public:

Vector(void);//конструктор без параметров

Vector(int);//конструктор копирования

public:

~Vector(void);//деструктор

void Add(Object\*);//добавление элемента в вектор

friend ostream& operator<<(ostream& out, const Vector&);//операция вывода

private:

Object\*\* beg;//указатель на первый элемент вектора

int size;//размер

int cur;//текущая позиция

};

**Vector.cpp:**

#include "Vector.h"

//конструктор без параметров

Vector::Vector(void)

{

beg = 0; size = 0; cur = 0;

}

//деструктор

Vector::~Vector(void)

{

if (beg != 0)delete[] beg; beg = 0;

}

//конструктор с параметрами

Vector::Vector(int n)

{

beg = new Object \* [n]; cur = 0;

size = n;

}

//добавление объекта, на который указывает указатель p в вектор

void Vector::Add(Object \*p)

{

if (cur < size)

{

beg[cur] = p; cur++;

}

}

//операция вывода

ostream& operator<<(ostream& out, const Vector& v)

{

if (v.size == 0) out << "Empty" << endl;

Object\*\* p = v.beg;//указатель на указатель типа Object

for(int i=0;i<v.cur;i++)

{

(\*p)->Show();//вызов метода Show() (позднее связывание)

p++;//передвигаем указатель на следующий объект

}

return out;

}

**Main.cpp:**

#include "Object.h"

#include "Triad.h"

#include "Date.h"

#include "Vector.h"

#include <string>

#include <iostream>

using namespace std;

void main()

{

setlocale(LC\_ALL, "Ru");

Vector v(5);//вектор из 5 элементов

Triad a;//объект класса Triad

cin >> a;

Date b;// объект класса Date

cin>>b;

Object\* p = &a;//ставим указатель на объект класса Car

v.Add(p);//добавляем объект в вектор

p = &b;//ставим указатель на объект класса Lorry

v.Add(p); //добавляем объект в вектор

cout<<v;//вывод вектора

}

**Вывод:**

Программа работает. Задача решена.

**Вопросы:**

### 1. Какой метод называется чисто виртуальным? Чем он отличается от виртуального метода?

**Чисто виртуальный метод** — это метод, который объявлен в базовом классе, но не имеет реализации в нём. Он обозначается синтаксисом = 0. Например, в C++:

**Отличие от виртуального метода:**

* Виртуальный метод имеет реализацию в базовом классе и может быть переопределён в производных.
* Чисто виртуальный метод **не имеет реализации** в базовом классе, и производные классы **обязаны** его переопределить (иначе они тоже станут абстрактными).

### 2. Какой класс называется абстрактным?

**Абстрактный класс** — это класс, содержащий хотя бы один чисто виртуальный метод. Создавать экземпляры такого класса **нельзя**, он предназначен только для наследования.

### 3. Для чего предназначены абстрактные классы?

* **Определение интерфейса:** Задание обязательных методов для производных классов.
* **Полиморфизм:** Обеспечение единого интерфейса для разных реализаций.
* **Запрет создания экземпляров:** Базовый класс описывает общую логику, но не конкретную реализацию.

### 4. Что такое полиморфные функции?

**Полиморфные функции** — это функции, которые могут работать с разными типами данных через единый интерфейс. В ООП это достигается через:

* **Переопределение виртуальных методов** (динамический полиморфизм).
* **Шаблоны/обобщённое программирование** (статический полиморфизм).

### 5. Чем полиморфизм отличается от принципа подстановки?

* **Полиморфизм** — это возможность использовать разные реализации методов через один интерфейс (например, через виртуальные функции).
* **Принцип подстановки (Liskov Substitution Principle, LSP)** — частный случай полиморфизма, требующий, чтобы объекты производных классов могли заменять объекты базовых без изменения корректности программы.

**Пример:**  
Если Квадрат наследуется от Прямоугольника, но меняет логику установки сторон, это нарушает LSP (хотя полиморфизм технически работает).

### 6. Примеры иерархий с абстрактными классами

**Животные:**

class Animal {

public:

virtual void makeSound() const = 0;

};

class Dog : public Animal {

public:

void makeSound() const override { std::cout << "Woof!"; }

};

### 7. Примеры полиморфных функций

void playSounds(const std::vector<Animal\*>& animals) {

for (auto animal : animals) {

animal->makeSound(); // вызовется нужная реализация

}

}

### 8. Когда используется позднее связывание?

**Позднее (динамическое) связывание** применяется:

1. При вызове **виртуальных методов** через указатель/ссылку на базовый класс.
2. Когда нужен **полиморфизм** (разные объекты ведут себя по-разному через один интерфейс).
3. В **шаблонах проектирования** (например, Стратегия, Фабрика).

Animal\* animal = new Dog();

animal->makeSound(); // Позднее связывание: вызовется Dog::makeSound()

**Раннее связывание** (на этапе компиляции) используется для невиртуальных методов и шаблонов.