Министерство образования и науки РФ

Пермский национальный исследовательский политехнический университет

Электротехнический факультет

Кафедра Информационные технологии и автоматизированные системы

Лабораторная работа № 5

Тема: «Наследование. Виртуальные функции. Полиморфизм»

Выполнил: студент группы РИС-22-2б

Мизёв В.А. ф

Проверил: доцент кафедры ИТАС

Полякова О.А.

г. Пермь – 2024

*Постановка задачи:*

1. Определить абстрактный класс.
2. Определить иерархию классов, в основе которой будет находиться абстрактный класс.
3. Определить класс Вектор, элементами которого будут указатели на объекты иерархии классов.
4. Перегрузка для класса Вектор операцию вывода объектов с помощью потоков.
5. В основной функции продемонстрировать перегруженные операции и полиморфизм Вектора.

*Задание вариант 1:*

Базовый класс: ПАРА\_ЧИСЕЛ(PAIR)

Первое\_число(first) – int

Второе\_число(second) – int

Определить методы изменения полей и сравнения пар (пара р1 больше пары р2, если (р1.first > p2.first) || (p.1first == p2.first && p1.second > p2.second)).

Создать производный класс ДРОБЬ (FRACTION), с полями Целая\_часть\_числа и Дробная\_часть\_числа. Определить полный набор методов сравнения.

*Анализ задачи:*

Основываясь на задании необходимо реализовать 4 класса “Pair”, “Fraction” и “Vector”. Где Fraction наследуется от Pair, также в классе “Pair”, есть два поля m\_first, m\_second. Целую и дробную часть чисел хранить не зачем, поэтому для них реализовать два метода getIntPart() и getFractionPart(). Также перегрузить все необходимые операторы (присваивания и сравнения).

Класс Vector будет с базовым пустым конструктором и с динамическим размером. Из функциональных методов это capacity и size, capacity возвращает сколько выделено памяти из кучи, а size кол-во элементов в этом векторе. Также перегружены все необходимые операторы.

*Код программы:*

Прикреплён в приложении 1.

*UML-диаграмма класса:*

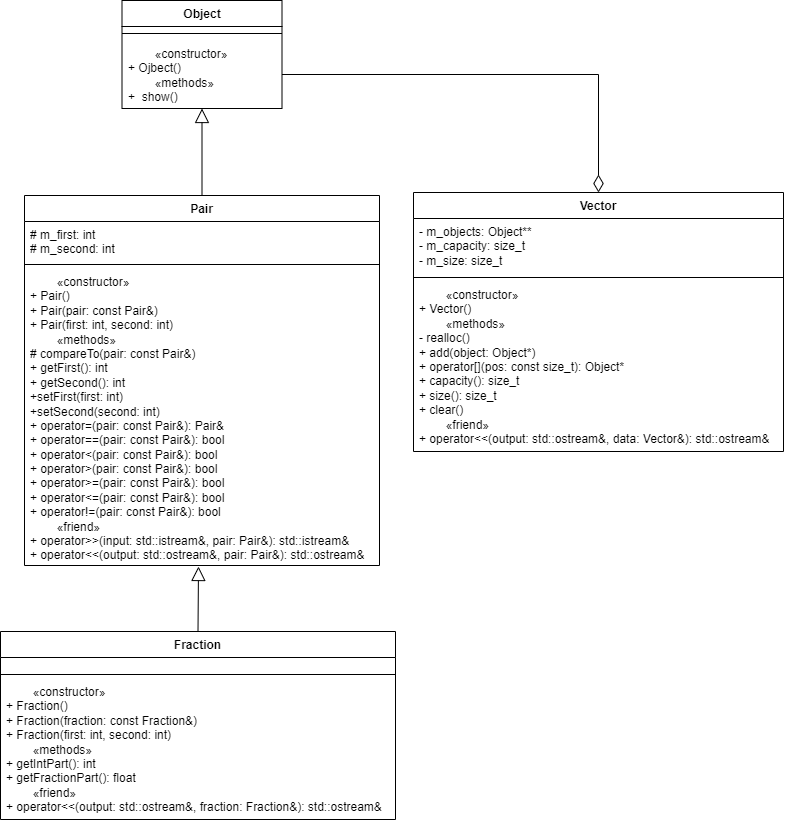


Рисунок 1 - UML-диаграмма классов

*Скриншот работы программы:*

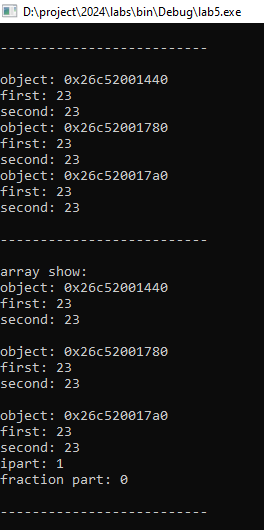


Рисунок 2 – Результат работы программы

*Контрольные вопросы:*

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

метода?

Чисто виртуальный метод — это метод, который не имеет реализации в базовом классе и объявлен с помощью = 0. Он обязательно должен быть реализован в любом наследуемом классе. Виртуальный метод может иметь реализацию в базовом классе.

class Base {

public:

virtual void regularMethod() {} // Виртуальный метод с реализацией

virtual void pureVirtualMethod() = 0; // Чисто виртуальный метод

};

2. Какой класс называется абстрактным?

Абстрактный класс — это класс, который содержит хотя бы один чисто виртуальный метод. Экземпляры абстрактных классов не могут быть созданы.

class Abstract {

public:

virtual void doSomething() = 0; // Чисто виртуальный метод

};

3. Для чего предназначены абстрактные классы?

Абстрактные классы предназначены для создания интерфейсов, которые должны быть реализованы в производных классах. Они обеспечивают общую базу для создания различных объектов с общим поведением.

4. Что такое полиморфные функции?

Полиморфные функции — это функции, которые могут принимать объекты разных типов из одной и той же иерархии классов и вызывать соответствующие реализации метода. Это позволяет использовать единый интерфейс для работы с различными типами объектов.

5. Чем полиморфизм отличается от принципа подстановки?

Полиморфизм — это способность функций или методов вести себя по-разному в зависимости от вызываемого объекта. Принцип подстановки подразумевает, что производный класс может быть использован вместо базового класса. То есть, объекты будут корректно заменены, что позволяет использовать их в одном контексте без изменения логики программы.

6. Привести примеры иерархий с использованием абстрактных классов.

class Shape {

public:

virtual void draw() = 0; // Чисто виртуальный метод

};

class Circle : public Shape {

public:

void draw() override { /\* реализация для круга \*/ }

};

class Rectangle : public Shape {

public:

void draw() override { /\* реализация для прямоугольника \*/ }

};

7. Привести примеры полиморфных функций.

void render(Shape& shape) {

shape.draw(); // Полиморфный вызов

}

class Shape { virtual void draw() =0; } class Rect : Shape { … }

Rect rect;

render(rect);

Где shape базовый класс, а класс rect, наследник

8. В каких случаях используется механизм позднего связывания?

Позднее связывание используется, когда необходимо вызывать методы в зависимости от динамического типа объекта, а не статического. Это позволяет создавать гибкие и расширяемые системы, где можно добавлять новые классы, не изменяя существующий код.

Shape\* shape = new Circle();

shape->draw(); // Метод будет вызван в зависимости от типа Circle, а не Shape

**ПРИЛОЖЕНИЕ 1**

Lab5.cpp

//-------------------------------------

#include <iostream>

#include <ctime>

#include "lab5\_vector.hpp"

#include "lab5\_object.hpp"

//-------------------------------------

#define IDENT\_PRINT printf("\n--------------------------\n\n")

//-------------------------------------

int main() {

srand(time(0));

Vector data;

IDENT\_PRINT;

for(int i = 0; i < 3; ++i) {

Object\* object;

int a = std::rand()%2;

switch(a){

case 0: object = new Pair; break;

case 1: object = new Fraction; break;

}

std::cin >> \*(a == 0 ? (Pair\*)object : (Fraction\*)object);

data.add(object);

}

IDENT\_PRINT;

std::cout << "array show: " << std::endl;

std::cout << data;

IDENT\_PRINT;

return 0;

}

//-------------------------------------

lab5\_object.hpp

//-------------------------------------

#ifndef LAB5\_OBJECT\_HPP\_INCLUDED

#define LAB5\_OBJECT\_HPP\_INCLUDED

//-------------------------------------

#include <iostream>

//-------------------------------------

class Object {

public:

Object() { }

virtual ~Object() { }

virtual void show() = 0;

};

//-------------------------------------

class Pair : public Object {

public:

Pair();

Pair(const Pair& pair);

Pair(int first, int second);

virtual ~Pair();

int getFirst() { return m\_first; }

int getSecond() { return m\_second; }

void setFirst(int first) { m\_first = first; }

void setSecond(int second) { m\_second = second; }

void show() { std::cout << \*this << std::endl; }

Pair& operator=(const Pair& pair);

bool operator==(const Pair& pair);

bool operator<(const Pair& pair);

bool operator>(const Pair& pair);

bool operator>=(const Pair& pair);

bool operator<=(const Pair& pair);

bool operator!=(const Pair& pair);

friend std::istream& operator>>(std::istream& input, Pair& pair);

friend std::ostream& operator<<(std::ostream& output, Pair& pair);

protected:

int m\_first,

m\_second;

int compareTo(const Pair& pair);

};

//-------------------------------------

class Fraction : public Pair {

public:

Fraction();

Fraction(const Fraction& fraction);

Fraction(int first, int second);

virtual ~Fraction();

int getIntPart() { return (int)(m\_first / m\_second); }

float getFractionPart() { return ((float)m\_first / (float)m\_second) - ((int)(m\_first / m\_second)); }

void show() { std::cout << \*this << std::endl; }

friend std::ostream& operator<<(std::ostream& output, Fraction& fraction);

protected:

};

//-------------------------------------

#endif // LAB5\_OBJECT\_HPP\_INCLUDED

//-------------------------------------

lab4\_object.cpp

//-------------------------------------

#include "lab5\_object.hpp"

//-------------------------------------

//-------------------------------------

// class Pair

//-------------------------------------

//-------------------------------------

Pair::Pair() {

m\_first = 0;

m\_second = 0;

}

//-------------------------------------

Pair::Pair(const Pair& pair) : Pair(pair.m\_first, pair.m\_second) {

//

}

//-------------------------------------

Pair::Pair(int first, int second) {

m\_first = first;

m\_second = second;

}

//-------------------------------------

Pair::~Pair() {

//

}

//-------------------------------------

int Pair::compareTo(const Pair& pair) {

#ifdef debug\_v

std::cout << "compare(" << this << "," << &pair << ")" << std::endl;

#endif // debug\_v

if(pair.m\_first == m\_first && pair.m\_second == m\_second)

return 0;

else if(m\_first > pair.m\_first || (m\_first == pair.m\_first && m\_second > pair.m\_second))

return 1;

return -1;

}

//-------------------------------------

Pair& Pair::operator=(const Pair& pair) {

if(this == &pair)

return \*this;

m\_first = pair.m\_first;

m\_second = pair.m\_second;

return \*this;

}

//-------------------------------------

bool Pair::operator==(const Pair& pair) {

#ifdef debug\_v

std::cout << "operator== ";

#endif // debug\_v

return compareTo(pair) == 0;

}

//-------------------------------------

bool Pair::operator!=(const Pair& pair) {

#ifdef debug\_v

std::cout << "operator!= ";

#endif // debug\_v

return compareTo(pair) != 0;

}

//-------------------------------------

bool Pair::operator<(const Pair& pair) {

#ifdef debug\_v

std::cout << "operator< ";

#endif // debug\_v

return compareTo(pair) == -1;

}

//-------------------------------------

bool Pair::operator>(const Pair& pair) {

#ifdef debug\_v

std::cout << "operator> ";

#endif // debug\_v

return compareTo(pair) == 1;

}

//-------------------------------------

bool Pair::operator>=(const Pair& pair) {

#ifdef debug\_v

std::cout << "operator>= ";

#endif // debug\_v

return compareTo(pair) != -1;

}

//-------------------------------------

bool Pair::operator<=(const Pair& pair) {

#ifdef debug\_v

std::cout << "operator<= ";

#endif // debug\_v

return compareTo(pair) != 1;

}

//-------------------------------------

std::istream& operator>>(std::istream& input, Pair& pair) {

#ifdef debug\_v

std::cout << "object: " << &pair << std::endl;

#endif // debug\_v

std::cout << "first: ";

input >> pair.m\_first;

std::cout << "second: ";

input >> pair.m\_second;

return input;

}

//-------------------------------------

std::ostream& operator<<(std::ostream& output, Pair& pair) {

#ifdef debug\_v

std::cout << "object: " << &pair << std::endl;

#endif // debug\_v

return output << "first: " << pair.m\_first

<< "\nsecond: " << pair.m\_second;

}

//-------------------------------------

//-------------------------------------

// class Fraction

//-------------------------------------

//-------------------------------------

Fraction::Fraction() : Pair() {

//

}

//-------------------------------------

Fraction::Fraction(const Fraction& fraction) : Pair(fraction.m\_first, fraction.m\_second) {

//

}

//-------------------------------------

Fraction::Fraction(int first, int second) : Pair(first, second) {

//

}

//-------------------------------------

Fraction::~Fraction() {

//

}

//-------------------------------------

std::ostream& operator<<(std::ostream& output, Fraction& fraction) {

#ifdef debug\_v

std::cout << "object: " << &fraction << std::endl;

#endif // debug\_v

return output << "first: " << fraction.m\_first

<< "\nsecond: " << fraction.m\_second

<< "\nipart: " << fraction.getIntPart()

<< "\nfraction part: " << fraction.getFractionPart();

}

//-------------------------------------

lab5\_vector.hpp

//-------------------------------------

#ifndef LAB5\_VECTOR\_HPP\_INCLUDED

#define LAB5\_VECTOR\_HPP\_INCLUDED

//-------------------------------------

#include <iostream>

//-------------------------------------

#define BASE\_SIZE 4

//-------------------------------------

class Object;

class Vector {

public:

Vector();

~Vector();

void add(Object\* object);

Object\* operator[](const size\_t pos) { return m\_size < pos ? nullptr : m\_objects[pos]; }

size\_t capacity() { return m\_capacity; }

size\_t size() { return m\_size; }

void clear();

friend std::ostream& operator<<(std::ostream& output, Vector& data);

private:

Object\*\* m\_objects;

size\_t m\_capacity,

m\_size;

void realloc();

};

//-------------------------------------

#endif // LAB5\_VECTOR\_HPP\_INCLUDED

//-------------------------------------

lab5\_vector.cpp

//-------------------------------------

#include "lab5\_vector.hpp"

#include "lab5\_object.hpp"

//-------------------------------------

Vector::Vector() {

m\_size = 0;

m\_capacity = 0;

this->realloc();

}

//-------------------------------------

Vector::~Vector() {

this->clear();

}

//-------------------------------------

void Vector::realloc() {

if(m\_size < m\_capacity)

return;

if(m\_capacity == 0) {

m\_capacity = BASE\_SIZE;

m\_objects = new Object\*[m\_capacity];

return;

}

m\_capacity\*=2;

Object\*\* array = new Object\*[m\_capacity];

for(size\_t i = 0; i < m\_size; ++i)

array[i] = m\_objects[i];

delete []m\_objects;

m\_objects = array;

}

//-------------------------------------

void Vector::add(Object\* object) {

this->realloc();

m\_objects[m\_size] = object;

m\_size++;

}

//-------------------------------------

void Vector::clear() {

for(size\_t i = 0; i < m\_size; ++i) {

if(m\_objects[i] != nullptr)

delete m\_objects[i];

}

delete []m\_objects;

m\_size = 0;

m\_capacity = 0;

}

//-------------------------------------

std::ostream& operator<<(std::ostream& output, Vector& data) {

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

data[i]->show();

if(i != data.size()-1)

std::cout << std::endl;

}

return output;

}

//-------------------------------------