Министерство образования и науки РФ

Пермский национальный исследовательский политехнический университет

Электротехнический факультет

Кафедра Информационные технологии и автоматизированные системы

Лабораторная работа № 4

Тема: «Простое наследование. Принципе подстановки»

Выполнил: студент группы РИС-22-2б

Мизёв В.А. ф

Проверил: доцент кафедры ИТАС

Полякова О.А.

г. Пермь – 2024

*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 принципе подстановки.

*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 несколько классов “Pair” и “Fraction”. 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).

*Код программы:*

Прикреплён в приложении 1.

*UML-диаграмма класса:*

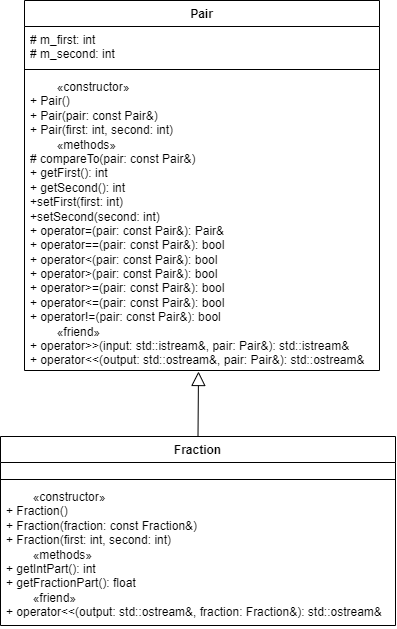


Рисунок 1 - UML-диаграмма классов

*Скриншот работы программы:*

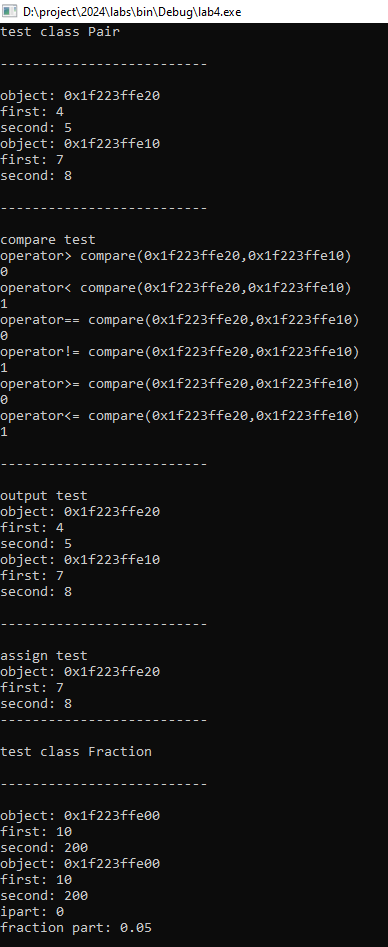


Рисунок 2 – Результат работы программы, первая часть с классом Pair

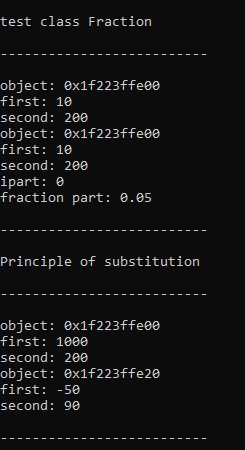


Рисунок 3 -Втора часть работы программы

*Контрольные вопросы:*

1. Для чего используется механизм наследования?

Наследование позволяет создавать новый класс на основе существующего, что способствует повторному использованию кода и расширяемости.

class Base {

public:

void show() { std::cout << "Base classn"; }

};

class Derived : public Base {};

2. Каким образом наследуются компоненты класса, описанные со спецификатором

public?

Компоненты класса с спецификатором public наследуются как public в производном классе.

class Base {

public:

int x;

};

class Derived : public Base {

public:

void setX(int val) { x = val; } // x доступен как public

};

3. Каким образом наследуются компоненты класса, описанные со спецификатором

private?

Компоненты класса с спецификатором private становятся private в производном классе, и не доступны напрямую из производного класса.

class Base {

private:

int y;

};

class Derived : public Base {

public:

void setY(int val) { /\* y недоступен напрямую \*/ }

};

4. Каким образом наследуются компоненты класса, описанные со спецификатором

protected?

Компоненты класса с спецификатором protected наследуются как protected в производном классе. Причём такое наследование позволяет классу наследнику обратиться к полям базового класса.

class Base {

protected:

int z;

};

class Derived : public Base {

public:

void setZ(int val) { z = val; } // z доступен как protected

};

5. Каким образом описывается производный класс?

Производный класс описывается с помощью ключевого слова :, за которым следуют спецификаторы доступа (public, protected или private) и имя базового класса.

class Derived : public Base {};

6. Наследуются ли конструкторы?

Конструкторы не наследуются. Однако производный класс может вызвать конструктор базового класса. Причём если явно не вызывать конструктор по умолчанию базового класса, то он вызовется с конструктором по умолчанию класса наследника.

class Base {

public:

Base() { std::cout << "Base constructorn"; }

};

class Derived : public Base {

public:

Derived() : Base() { std::cout << "Derived constructorn"; }

};

7. Наследуются ли деструкторы?

Деструкторы также не наследуются, но могут быть вызваны при уничтожении объектов производного класса.

class Base {

public:

~Base() { std::cout << "Base destructorn"; }

};

class Derived : public Base {

public:

~Derived() { std::cout << "Derived destructorn"; }

};

8. В каком порядке конструируются объекты производных классов?

Сначала конструируется базовый класс, затем производный.

9. В каком порядке уничтожаются объекты производных классов?

Сначала уничтожается производный класс, затем базовый.

10. Что представляют собой виртуальные функции и механизм позднего связывания?

Виртуальные функции позволяют переопределять функции в производных классах, использующие полиморфизм и позднее связывание.

class Base {

public:

virtual void show() { std::cout << "Base shown"; }

};

class Derived : public Base {

public:

void show() override { std::cout << "Derived shown"; }

};

11. Могут ли быть виртуальными конструкторы? Деструкторы?

Виртуальных конструкторов быть не может, но деструкторы могут быть виртуальными.

class Base {

public:

virtual ~Base() { std::cout << "Base destructorn"; }

};

12. Наследуется ли спецификатор virtual?

Да, если базовая функция объявлена как виртуальная, то производные функции, которые ее переопределяют, также будут виртуальными.

13. Какое отношение устанавливает между классами открытое наследование?

Открытое (public) наследование позволяет производному классу использовать public и protected члены базового класса как public и protected соответственно.

14. Какое отношение устанавливает между классами закрытое наследование?

Закрытое (private) наследование делает все члены базового класса private в производном классе.

class Base {

public:

int x;

};

class Derived : private Base { // закрытое наследование

public:

void setX(int val) { x = val; } // доступ к x внутри Derived

};

15. В чем заключается принцип подстановки?

Принцип подстановки (Liskov Substitution Principle) утверждает, что объекты производного класса должны быть заменяемы объектами базового класса без изменения корректности программы.

void draw(Base\* b) {

b->show(); // можно передать объект Base или Derived

}

Эти примеры кода иллюстрируют основные концепции и механизмы, связанные с наследованием в C++.

**ПРИЛОЖЕНИЕ 1**

lab4.cpp

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

#include <iostream>

#include "lab4\_object.hpp"

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

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

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

void output\_test(Pair& pair) {

pair.setFirst(1000);

std::cout << pair << std::endl;

}

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

Pair create\_fraction() {

return Fraction(-50, 90);

}

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

int main() {

///test class Pair

Pair pair,

pair1;

std::cout << "test class Pair" << std::endl;

IDENT\_PRINT;

//input test

std::cin >> pair;

std::cin >> pair1;

IDENT\_PRINT;

//compare test

std::cout << "compare test" << std::endl;

std::cout << (pair > pair1) << std::endl;

std::cout << (pair < pair1) << std::endl;

std::cout << (pair == pair1) << std::endl;

std::cout << (pair != pair1) << std::endl;

std::cout << (pair >= pair1) << std::endl;

std::cout << (pair <= pair1) << std::endl;

IDENT\_PRINT;

//output test

std::cout << "output test" << std::endl;

std::cout << pair << std::endl;

std::cout << pair1 << std::endl;

IDENT\_PRINT;

//assign test

std::cout << "assign test" << std::endl;

pair = pair1;

std::cout << pair;

IDENT\_PRINT;

std::cout << "test class Fraction" << std::endl;

IDENT\_PRINT;

Fraction fraction;

//input test

std::cin >> fraction;

std::cout << fraction << std::endl;

IDENT\_PRINT;

std::cout << "Principle of substitution" << std::endl;

IDENT\_PRINT;

output\_test(fraction);

pair = create\_fraction();

std::cout << pair << std::endl;

IDENT\_PRINT;

return 0;

}

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

lab4\_object.hpp

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

#ifndef LAB4\_OBJECT\_HPP\_INCLUDED

#define LAB4\_OBJECT\_HPP\_INCLUDED

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

#include <iostream>

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

class Pair {

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; }

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)); }

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

protected:

};

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

#endif // LAB4\_OBJECT\_HPP\_INCLUDED

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

lab4\_object.cpp

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

#include "lab4\_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();

}

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