Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

Пермский национальный исследовательский политехнический университет

Электротехнический факультет

Кафедра информационных технологий и автоматизированных систем

**ОТЧЕТ**

**о работе по информатике**

Семестр: 2

На тему: «Классы и объекты. Инкапсуляция»

Выполнил студент ИВТ-22-2б:

Захаров Дмитрий Сергеевич

Проверил доцент кафедры ИТАС:

Полякова Ольга Андреевна

Пермь 2023

**1 Постановка задачи**

1. Определить абстрактный класс.

2. Определить иерархию классов, в основе которой будет находиться абстрактный класс из лабораторной работы №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

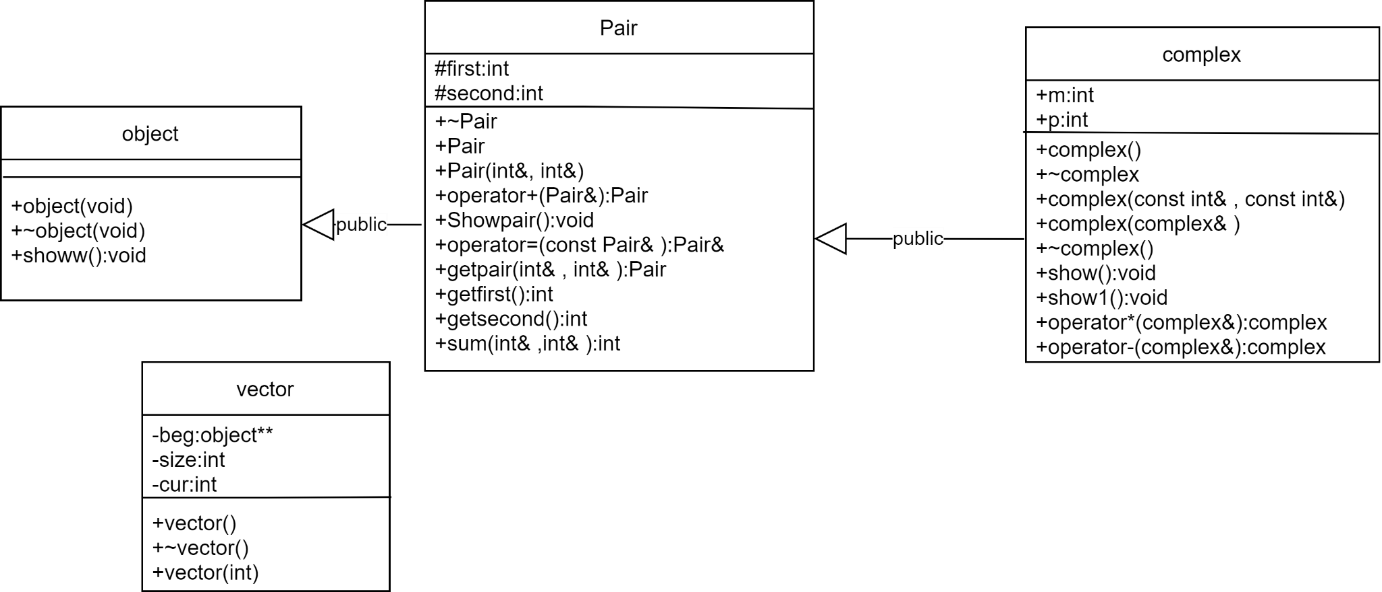
полиморфизм Вектора.

**2 Анализ задачи**

Под основу берется программа из 4 лабораторной, добавляется абстрактный класс object, который содержит метод virtual void output.

Создается класс vector, перегружаются операторы ввода и вывода для вектора. Полиморфизм заключается в том, что метод классов vector, person, student имеют одно имя, ошибки не возникает.

**3 UML-диаграмма**

****

**4 Код программы**

#include <iostream>

using namespace std;

class vector {

object\*\* beg;

int size;

int cur;

public:

vector();

vector(int);

~vector();

void add(object\*);

friend ostream& operator<<(ostream& out, const vector&);

};

vector::vector() {

beg = 0;

size = 0;

cur = 0;

};

vector::~vector() {

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

beg = 0;

};

vector::vector(int n) {

beg = new object \* [n];

cur = 0;

size = n;

};

void vector::add(object\* p) {

if (cur < size) {

beg[cur] = p;

cur++;

}

};

ostream& operator<<(ostream& out, const vector& v) {

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

object\*\* p = v.beg;

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

(\*p)->showw();

p++;

}

return out;

};

class object {

public:

object(void) {};

~object(void) {};

virtual void showw() = 0;

};

class Pair : public object {

protected:

int first;

int second;

public:

~Pair(void);

Pair(void);

Pair(int&, int&);

Pair(Pair&);

Pair operator+(Pair&);

void showw() {

cout << "Пара чисел: " << '(' << first << ',' << second << ')' << endl;

}

friend ostream& operator<<(ostream& stream, const Pair&);

friend istream& operator>>(istream&, Pair&);

Pair& operator=(const Pair& a);

Pair getpair(int& t, int& r) {

Pair a(t, r);

return a;

}

int getfirst() {

return this->first;

}

int getsecond() {

return this->second;

}

int sum(int& t, int& p) {

Pair a(t, p);

return a.first + a.second;

}

};

Pair::~Pair() {

}

Pair::Pair() {

first = 0;

second = 0;

}

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

this->first = first;

this->second = second;

}

Pair::Pair(Pair& tmp) {

this->first = tmp.first;

this->second = tmp.second;

}

Pair Pair:: operator+(Pair& p) {

int tmp = this->first + p.first;

int tmp1 = this->second + p.second;

Pair a(tmp, tmp1);

return a;

}

ostream& operator<<(ostream& stream, const Pair& tm) {

stream << '(' << tm.first << ',' << tm.second << ')';

return stream;

};

istream& operator>>(istream& stream1, Pair& tm) {

stream1 >> tm.first;

stream1 >> tm.second;

return stream1;

};

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

this->first = a.first;

this->second = a.second;

return \*this;

}

class complex : public Pair {

public:

int m;

int p;

complex();

complex(const int&, const int&);

complex(complex&);

~complex();

void showw() {

cout << "Комплексное число: " << '(' << m << ',' << p << "i)" << endl;

}

void show1() {

cout << "Пара без мнимой части: " << getpair(this->m, this->p) << "sum: " << sum(this->m, this->p) << endl;

}

complex operator\*(complex&);

complex operator-(complex&);

friend ostream& operator<<(ostream& stream, const complex& tm);

friend istream& operator>>(istream&, complex&);

};

complex::~complex() {

}

complex::complex() {

}

complex::complex(const int& m, const int& p) {

this->m = m;

this->p = p;

}

complex::complex(complex& tmp) {

this->m = tmp.m;

this->p = tmp.p;

}

complex complex::operator\*(complex& t) {

int tmp = this->m \* t.m - (this->p \* t.p);

int tmp1 = this->m \* t.p + (this->p \* t.m);

complex a(tmp, tmp1);

return a;

}

int main()

{

setlocale(LC\_ALL, "ru");

Pair a, b;

cout << "Введите первую пару чисел: ";

cin >> a;

cout << "Введите вторую пару чисел: ";

cin >> b;

cout << "Первая: " << a << "\t" << "Вторая: " << b << endl;

cout << "a+b " << a + b << endl;

complex d, r;

cout << "Введите комплексное число: ";

cin >> d;

cout << "Введите комплексное число: ";

cin >> r;

cout << r << d << endl;

cout << "r \* d = " << r \* d << endl;

cout << "r - d = " << r - d << endl;

d.show1();

r.show1();

a.showw();

b.showw();

d.showw();

r.showw();

vector v(5);

Pair t;

cin >> t;

complex y;

cin >> y;

object\* p = &t;

v.add(p);

p = &y;

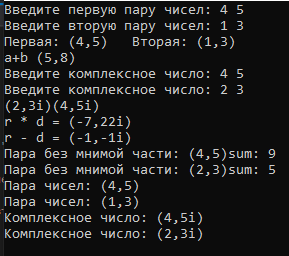
v.add(p);

cout << v;

return 0;

}

**5 Результаты работы программы**

****

**6 Ответы на вопросы**

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

Чисто виртуальный метод в базовом класса содержит признак равенства 0 вместо тела, например: virtual void f(int) = 0. Он должен переопределяться в производном классе (с помощью спецификатора override, который не является обязательным).

1. Какой класс называется абстрактным?

Абстрактный класс - класс, который содержит или наследует хотя бы одну чисто виртуальную функцию без переопределения.

1. Для чего предназначены абстрактные классы?

Абстрактные классы используются в качестве обобщения концепций, на основе которые можно создавать более конкретные производные классы.

Создание объектов абстрактного класса невозможно (компилятор выдаст ошибку), но можно использовать указатели и ссылки на типы абстрактных классов. Также, этот класс нельзя использовать при явном приведении типов (тип переменной, тип возвращаемого значения и т.п.)

class Object

{

public:

Object(void); public:

~Object(void);

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

};

1. Что такое полиморфные функции?

Полиморфизм - взаимозаменяемость объектов.

В контексте реализации программы полиморфизм - возможность использования экземпляра класса-наследника там, где требуется экземпляр базового класса.

class Base // Абстрактный класс

{

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

};

class Derived : Base

{

public:

virtual void print(Base\* p)

{

cout << "It's all right";

}

};

void main()

{

Derived\* t = new Derived;

t.print(t); // в функцию передается базовый файл в качестве аргумента.

}

Абстрактный класс становится здесь шаблоном.

1. Чем полиморфизм отличается от принципа подстановки?

Полиморфизм - это возможность классов иметь разную реализацию одного и того же функционала в основном с помощью наследования. Это механизм работы языка программирования.

Принцип подстановки является “правилом хорошего кода". В хорошем коде все методы класса-потомка должны быть применимы к объекту класса родителя.

1. Привести примеры иерархий с использованием абстрактных классов.

class Object // Класс объекта

{

public:

virtual double weight() const = 0; // вес

virtual double volume() const = 0; // объём

};

class Sphere : public Object // класс сферы

{

int radius;

double pi = 3.14;

int density;

public:

Sphere(int R, double pi);

double volume() const override // переопределение функции

{

return 1.333\*(pi\*pow(radius, 3));

}

double weight() const override // переопределение функции

{

return volume() \* density;

}

};

1. Привести примеры полиморфных функций.

class Book

{

protected:

string title;

string author;

public:

Book(string t, string a)

{

title = t;

author = a;

}

virtual void show()

{

cout << title << endl << author;

}

};

class Page : public Book

{

protected:

int charter;

int page;

public:

Page(string t, string a, int ch, int p) : Book(t, a)

{

charter = ch;

page = p;

}

void show()

{

cout << title << endl << author;

cout << charter << endl << page;

}

};

1. В каких случаях используется механизм позднего связывания?

Если в иерархии унаследованных классов объявляется виртуальный элемент, то выполняется позднее связывание.

При позднем связывании вызов метода происходит на основании типа объекта, а не типа ссылки на базовый класс. Позднее связывание используется, если нужно реализовать полиморфизм.