ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ УФИМСКИЙ УНИВЕРСИТЕТ НАУКИ И ТЕХНОЛОГИЙ

Кафедра ВМиК

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №5**

по предмету **«Объектно-ориентированное программирование»**

Выполнил: студент группы МО-203б

Ярошко Е. В.

Проверил:  
доцент каф. ВМиК

Котельников В. А.

**Уфа 2025 г.**

**Цель лабораторной работы**

Узнать определения и реализацию тестовых классов, написать программу, иллюстрирующую их использование.

**Задание**

Определение и реализация тестовых классов, и написание программы, иллюстрирующей их использование. Лабораторная работа должна включать несколько программ, по мере изучения соответствующих понятий из лекционного курса, не стоит пытаться писать все сразу в одну программу и строго последовательно.

1. Определения:

* перекрываемых методов (показать на примере случай, когда вызывается перекрываемый метод, а когда наследуемый)
* виртуальных методов, в т.ч. деструкторов (показать на примере, когда вызывается наследуемый виртуальный метод, а когда базовый)

1. Реализация:

* проверки на принадлежность некоторому классу
* безопасного приведения типов (dynamic\_cast)
* безопасного приведения типов (вручную)

1. Передача объектов как параметров в функции и возвращения объектов как результата из функции, контроль их жизненного цикла;
2. Умные указатели unique\_ptr и shared\_ptr, их влияние на жизненный цикл объектов.

**Ход выполнения лабораторной работы**

В качестве среды разработки для выполнения данной работы был использован редактор Visual Studio Code совместно с компилятором g++.

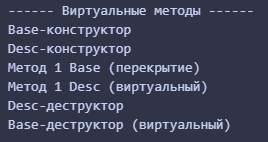
В ходе выполнения был реализован базовый класс Base, содержащий в себе конструкторы, а также виртуальный деструктор. Помимо этого, в нём реализованы два метода: один будет перекрываться в классах-наследниках, другой будет переопределён, т. к. является виртуальным. Также в нём созданы виртуальные методы для проверки на принадлежность к классу (classname, isA).

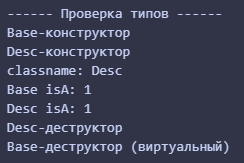
Создан класс-наследник Desc, в котором реализован перекрытый метод, а также переопределённый виртуальный метод из базового класса Base. Также в нём переопределены методы для проверки на принадлежность к классу.

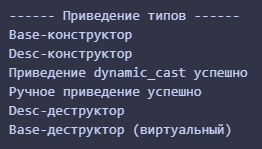
Созданы три функции для демонстрации передачи объектов как параметров в функцию тремя разными способами: как копия объекта, как указатель на объект и как ссылка на объект.

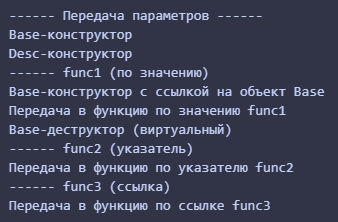
Созданы три функции для демонстрации возврата объектов из функции тремя способами: возврат статического объекта, возврат динамического объекта и возврат статической ссылки на объект.

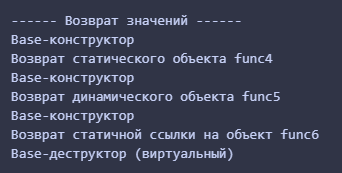
В главном цикле программы реализован удобный отладочный вывод и демонстрация работы виртуальных методов (Рисунок 1), проверки типов (Рисунок 2), приведения типов вручную и через dynamic\_cast (Рисунок 3), передачи параметров (Рисунок 4), возврата значений (Рисунок 5) и умных указателей (Рисунок 6).

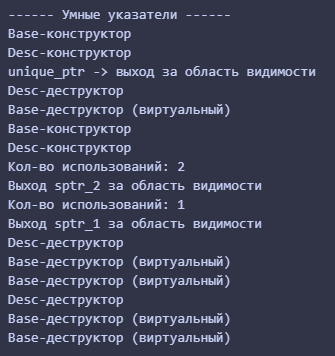
Рисунок 1 — Работа виртуальных методов

Рисунок 2 — Проверка типов

Рисунок 3 — Приведение типов

Рисунок 4 — Передача параметров

Рисунок 5 — Возврат значений

Рисунок 6 — Умные указатели

**Выводы по лабораторной работе**

Узнали определения и реализацию тестовых классов, написали программу, иллюстрирующую их использование.

**Приложение 1**

Исходные коды программ по заданиям

**Задание 1**

Файл lab5.cpp: <https://github.com/qvap/OOP--Lab-5/blob/main/lab5.cpp>

Исходный код:

#include <iostream>

#include <memory>

using namespace std;

class Base{

public:

// Конструкторы

Base(){

printf("Base-конструктор\n");

}

Base(Base\* obj){

printf("Base-конструктор с указателем на объект Base\n");

}

Base(Base& obj){

printf("Base-конструктор с ссылкой на объект Base\n");

}

// Деструктор

virtual ~Base(){

printf("Base-деструктор (виртуальный)\n");

}

// Методы 1 (перекрытие и виртуальность)

void method1(){

printf("Метод 1 Base (перекрытие)\n");

}

virtual void method1virtual(){

printf("Метод 1 Base (виртуальный)\n");

}

// Проверка типа

virtual string classname(){

return "Base";

}

virtual bool isA(string className){

return className == "Base";

}

};

class Desc : public Base{

public:

// Конструкторы

Desc(){

printf("Desc-конструктор\n");

}

Desc(Desc\* obj){

printf("Desc-конструктор с указателем на объект Desc\n");

}

Desc(Desc& obj){

printf("Desc-конструктор с ссылкой на объект Desc\n");

}

// Деструктор

~Desc(){

printf("Desc-деструктор\n");

}

// Перекрытие метода 1

void method1(){

printf("Метод 1 Desc (перекрытие)\n");

}

// Переопределение виртуального метода 1

void method1virtual() override{

printf("Метод 1 Desc (виртуальный)\n");

}

// Переопределение методов проверки типа

string classname() override{

return "Desc";

}

bool isA(string className) override{

return className == "Desc" || Base::isA(className);

}

};

// Передача параметров

void func1(Base obj){

printf("Передача в функцию по значению func1\n");

}

void func2(Base\* obj){

printf("Передача в функцию по указателю func2\n");

}

void func3(Base& obj){

printf("Передача в функцию по ссылке func3\n");

}

// Возврат значений

Base func4(){

Base obj;

printf("Возврат статического объекта func4\n");

return obj;

}

Base\* func5(){

Base\* obj = new Base();

printf("Возврат динамического объекта func5\n");

return obj;

}

Base& func6(){

static Base obj;

printf("Возврат статичной ссылки на объект func6\n");

return obj;

}

int main(){

cout << "------ Виртуальные методы ------\n";

Base\* baseptr = new Desc();

baseptr->method1(); // Вызывает метод Base (т. к. указатель на Base)

baseptr->method1virtual(); // Вызывает метод Desc (override)

delete baseptr;

cout << "\n------ Проверка типов ------\n";

Base\* obj = new Desc();

cout << "classname: " << obj->classname() << endl;

cout << "Base isA: " << obj->isA("Base") << endl;

cout << "Desc isA: " << obj->isA("Desc") << endl;

delete obj;

cout << "\n------ Приведение типов ------\n";

Base\* obj2 = new Desc();

if (dynamic\_cast<Desc\*>(obj2)){

cout << "Приведение dynamic\_cast успешно\n";

}

if (obj2->isA("Desc")){

cout << "Ручное приведение успешно\n";

}

delete obj2;

cout << "\n------ Передача параметров ------\n";

Desc obj3;

cout << "------ func1 (по значению)\n";

func1(obj3);

cout << "------ func2 (указатель)\n";

func2(&obj3);

cout << "------ func3 (ссылка)\n";

func3(obj3);

cout << "\n------ Возврат значений ------\n";

Base returnedobj = func4();

Base\* returnedptr = func5();

Base& returnedref = func6();

delete returnedptr;

cout << "\n------ Умные указатели ------\n";

{

unique\_ptr<Base> uptr = make\_unique<Desc>();

cout << "unique\_ptr -> выход за область видимости\n";

}

{

shared\_ptr<Base> sptr\_1 = make\_shared<Desc>();

{

shared\_ptr<Base> sptr\_2 = sptr\_1;

cout << "Кол-во использований: " << sptr\_1.use\_count() << endl;

cout << "Выход sptr\_2 за область видимости\n";

}

cout << "Кол-во использований: " << sptr\_1.use\_count() << endl;

cout << "Выход sptr\_1 за область видимости\n";

}

return 0;

}