2. Цель и задачи лабораторной работы

Цель работы — освоить механизмы объектно-ориентированного программирования (ООП), включая:

- Инкапсуляцию
- Наследование
- Полиморфизм
- Композицию объектов

Задачи:

1. Реализовать классы с отношением включения (композиции) и проверить корректное управление динамической памятью.
2. Добавить класс-наследник и продемонстрировать работу механизма наследования.
3. Использовать виртуальные функции для демонстрации полиморфизма.
4. Осуществить тестирование разработанных классов (включая проверку освобождения памяти).

При этом **строго исключить** использование контейнеров стандартной библиотеки (STL) — std::string, std::vector и т.д. Вместо них использовать динамически выделяемые массивы char[].

3. Описание структуры классов

В работе реализованы 4 класса:

1. **COne**
    - Не входит в иерархию наследования, но включён в класс CTwo (композиция).
    - Хранит число с плавающей точкой и динамическую строку (массив char).
2. **CTwo**
    - Содержит динамическую строку s и объект класса COne.
    - Имеет виртуальный метод print(), который выводит поля класса и данные объекта COne.
3. **CThree**
    - **Наследуется** от CTwo, добавляет дополнительное поле extraField (тоже динамическая строка).
    - Переопределяет виртуальный метод print().

4. **CFour**

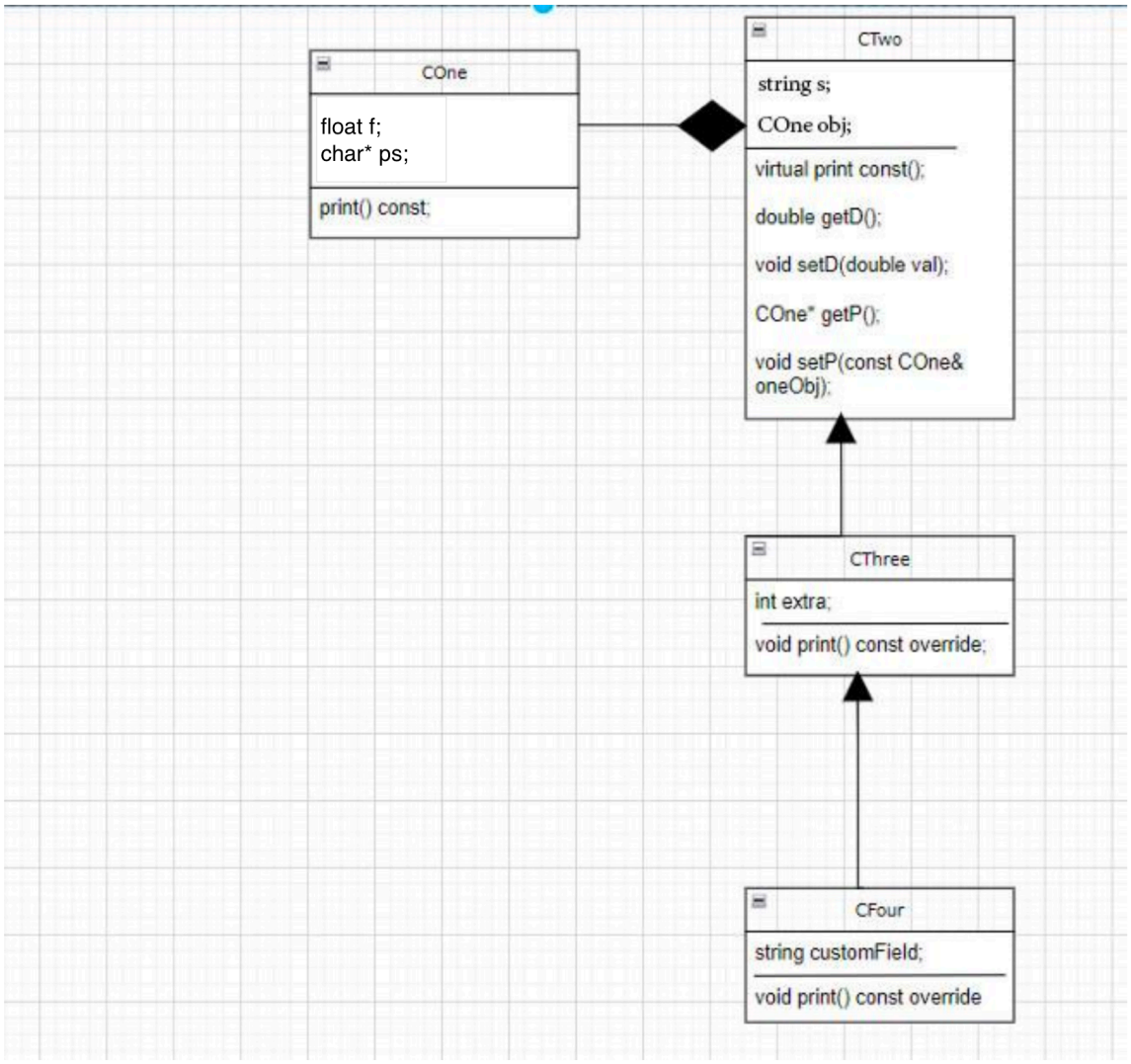
    - **Наследуется** от CThree, добавляет поле additionalData типа int.
    - Переопределяет виртуальный метод print().

Ниже приведена примерная UML-диаграмма, описывающая связи между классами (композиция и наследование).

```
classDiagram
   class COne {
      - float f
      - char* ps
      + COne()
      + COne(float, const char*)
      + COne(const COne&)
      + ~COne()
      + operator=(...)
      + print() : void
   }

   class CTwo {
      - char* s
      - COne obj
      + CTwo()
      + CTwo(const char*, COne)
      + CTwo(const CTwo&)
      + virtual ~CTwo()
      + operator=(...)
      + print() : void
   }

   class CThree {
      - char* extraField
      + CThree()
      + CThree(const char*, COne, const char*)
      + CThree(const CThree&)
      + virtual ~CThree()
      + operator=(...)
      + print() : void
   }

   class CFour {
```

- int additionalData
+ CFour()
+ CFour(const char*, COne, const char*, int)
+ CFour(const CFour&)
+ virtual ~CFour()
+ operator=(...)
+ print() : void
}

CTwo *-- COne : has-a
CThree --|> CTwo : inherits
CFour --|> CThree : inherits



4. Описание классов

4.1. Класс COne

- **Назначение**: хранит число с плавающей точкой и C-строку.
- **Поля**:
  - float f — вещественное число.
  - char* ps — динамическая C-строка.
- **Методы**:
  - **Конструктор по умолчанию**: инициализирует f = 0.0f, выделяет память под пустую строку ps.
  - **Конструктор с параметрами**: принимает float и const char*, копирует строку.
  - **Конструктор копирования**: выполняет глубокое копирование строки.
  - **Оператор присваивания**: освобождает старую память, копирует новую.
  - **Деструктор**: освобождает ps.
  - **print()**: выводит f и ps.

4.2. Класс CTwo

- **Назначение**: хранит строку s и объект COne. Продолжает работу с динамической памятью.
- **Поля**:
  - char* s — динамическая C-строка.
  - COne obj — объект класса COne (композиция).
- **Методы**:
  - Аналогичные конструкторы, оператор присваивания, деструктор.
  - **Виртуальный метод** print(), выводящий s и данные obj.print().

4.3. Класс CThree (наследник CTwo)

- **Назначение**: расширение CTwo, добавляет поле extraField (C-строка).
- **Методы**:
  - Переопределённый **виртуальный** print(), который дополнительно выводит extraField.

4.4. Класс CFour (наследник CThree)

- **Назначение**: расширение CThree, добавляет поле additionalData (int).
- **Методы**:

- Переопределённый **виртуальный** print(), выводящий поле additionalData помимо данных базового класса.

5. Реализация конструкций классов

Все классы используют динамические массивы char для хранения строк (вместо std::string).

При этом:

1. В конструкторах по умолчанию создаётся **пустая** строка (один символ '').
2. В операторах присваивания перед копированием новых данных освобождается старая память через delete[].
3. У каждого класса, содержащего динамическую строку, есть соответствующий деструктор.
4. **Виртуальные** деструкторы у CTwo, CThree, CFour обеспечивают корректное освобождение памяти при полиморфном удалении (delete basePtr;).

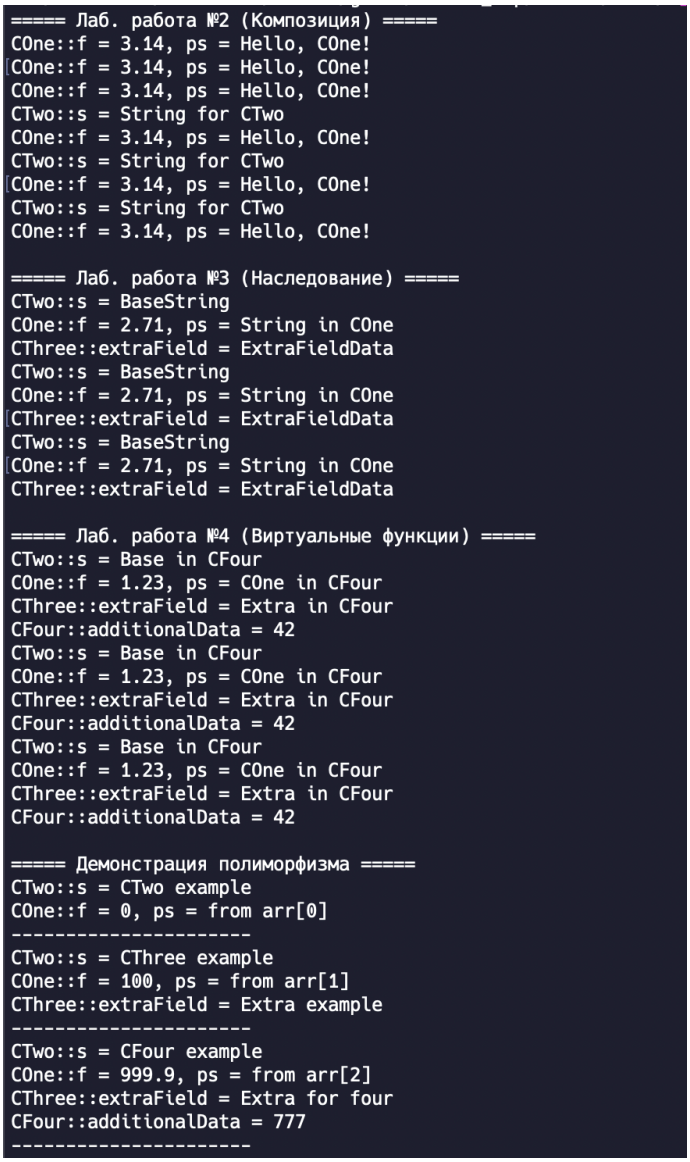
6. Проблемы реализации

1. **Необходимость конструктора по умолчанию** в COne. Без него при создании obj внутри других классов возникали ошибки.
2. **Исключение контейнеров STL**: вместо std::string или std::vector везде применяются динамические массивы char. Пришлось вручную управлять памятью (копировать строки, освобождать память).
3. **Осторожность при наследовании**:
   - При вызове конструктора CThree обязательно вызвать конструктор CTwo.
   - При вызове конструктора CFour обязательно вызвать конструктор CThree.
   - Проверка самоприсваивания в operator=.
4. **Проверка освобождения памяти** (Valgrind, ASan и т.п.) показала, что при корректной реализации утечек нет.

7. Тестирование программы

В main.cpp создаются объекты перечисленных классов, проверяются различные конструкторы, оператор присваивания и вывод через метод print().

Дополнительно продемонстрирован **полиморфизм**: массив указателей типа CTwo* с объектами CTwo, CThree и CFour; при вызове print() с помощью базового указателя вызывается **фактический** метод класса объекта (динамический полиморфизм).



Результаты тестов:

1. Все методы корректно вызываются и выводят ожидаемые значения.
2. Память освобождается без утечек (проверено инструментами).

8. Выводы

- Реализованы базовые механизмы ООП: **композиция**, **наследование**, **полиморфизм**.
- Показано управление динамической памятью без использования std::string (каждый класс работает со своими char*).
- Продемонстрировано, что **виртуальные методы** и **виртуальные деструкторы** необходимы для корректного полиморфного поведения.
- Лабораторная работа успешно выполнена в соответствии с заданием.

9. Приложения

9.1. Полный листинг кода

Cone.h:

```
#ifndef CONE_H
#define CONE_H
```

```
class COne {
private:
    float f;      // число с плавающей точкой
    char* ps;     // указатель на С-строку

public:
    // Конструктор по умолчанию
    COne();

    // Конструктор с параметрами
    COne(float value, const char* str);

    // Конструктор копирования
    COne(const COne& other);

    // Оператор присваивания
    COne& operator=(const COne& other);

    // Деструктор
    ~COne();

    // Методы доступа (геттеры и сеттеры)
    float getF() const;
    void setF(float value);

    const char* getPs() const;
    void setPs(const char* str);

    // Метод print()
    void print() const;
};
```

```
#endif // CONE_H
```

Cone.cpp:

```
#include "cone.h"
#include <cstring>   // для strlen, strcpy
```

```
#include <iostream>  // для std::cout

COne::COne() : f(0.0f) {
    ps = new char[1];
    ps[0] = '\0';
}

COne::COne(float value, const char* str) : f(value) {
    if (str) {
        ps = new char[strlen(str) + 1];
        strcpy(ps, str);
    } else {
        ps = new char[1];
        ps[0] = '\0';
    }
}

COne::COne(const COne& other) : f(other.f) {
    ps = new char[strlen(other.ps) + 1];
    strcpy(ps, other.ps);
}

COne& COne::operator=(const COne& other) {
    if (this == &other) {
        return *this; // защита от самоприсваивания
    }
    f = other.f;
    delete[] ps;
    ps = new char[strlen(other.ps) + 1];
    strcpy(ps, other.ps);
    return *this;
}
```

```
COne::~COne() {
    delete[] ps;
}

float COne::getF() const {
    return f;
}

void COne::setF(float value) {
    f = value;
}

const char* COne::getPs() const {
    return ps;
}

void COne::setPs(const char* str) {
    delete[] ps;
    if (str) {
        ps = new char[strlen(str) + 1];
        strcpy(ps, str);
    } else {
        ps = new char[1];
        ps[0] = '\0';
    }
}

void COne::print() const {
    std::cout << "COne::f = " << f
              << ", ps = " << ps << std::endl;
}
```

Two.h:

```
#ifndef CTWO_H
#define CTWO_H
```

```
#include "cone.h"

class CTwo {
protected:
    char* s;   // динамическая С-строка
    COne obj;  // вложенный объект класса COne

public:
    // Конструктор по умолчанию
    CTwo();

    // Конструктор с параметрами
    CTwo(const char* str, const COne& cOneObj);

    // Конструктор копирования
    CTwo(const CTwo& other);

    // Виртуальный деструктор
    virtual ~CTwo();

    // Оператор присваивания
    CTwo& operator=(const CTwo& other);

    // Методы доступа
    const char* getS() const;
    void setS(const char* str);

    COne getObj() const;
    void setObj(const COne& cOneObj);

    // Виртуальный метод print() для полиморфизма
```

```
    virtual void print() const;

};


#endif // CTWO_H
```

CTwo.cpp:

```
#include "ctwo.h"
#include <cstring>
```

```
#include <iostream>


CTwo::CTwo() : s(nullptr), obj() {
    s = new char[1];
    s[0] = '\0';
}


CTwo::CTwo(const char* str, const COne& cOneObj) : s(nullptr), obj(cOneObj) {
    if (str) {
        s = new char[strlen(str) + 1];
        strcpy(s, str);
    } else {
        s = new char[1];
        s[0] = '\0';
    }
}


CTwo::CTwo(const CTwo& other) : s(nullptr), obj(other.obj) {
    if (other.s) {
        s = new char[strlen(other.s) + 1];
        strcpy(s, other.s);
    } else {
        s = new char[1];
        s[0] = '\0';
    }
}


CTwo::~CTwo() {
    delete[] s;
}


CTwo& CTwo::operator=(const CTwo& other) {
```

```
    if (this == &other) {
        return *this;
    }
    obj = other.obj;
    delete[] s;
    if (other.s) {
        s = new char[strlen(other.s) + 1];
        strcpy(s, other.s);
    } else {
        s = new char[1];
        s[0] = '\0';
    }
    return *this;
}

const char* CTwo::getS() const {
    return s;
}

void CTwo::setS(const char* str) {
    delete[] s;
    if (str) {
        s = new char[strlen(str) + 1];
        strcpy(s, str);
    } else {
        s = new char[1];
        s[0] = '\0';
    }
}

COne CTwo::getObj() const {
    return obj;
```

```
}


void CTwo::setObj(const COne& cOneObj) {

    obj = cOneObj;

}


void CTwo::print() const {

    std::cout << "CTwo::s = " << s << std::endl;

    obj.print();

}
```

CThree.h:

```
#ifndef CTHREE_H
#define CTHREE_H
```

```
#include "ctwo.h"

class CThree : public CTwo {
private:
    char* extraField;

public:
    // Конструктор по умолчанию
    CThree();

    // Конструктор с параметрами
    CThree(const char* str, const COne& cOneObj, const char* extra);

    // Конструктор копирования
    CThree(const CThree& other);

    // Оператор присваивания
    CThree& operator=(const CThree& other);

    // Виртуальный деструктор
    virtual ~CThree();

    // Методы доступа к новому полю
    const char* getExtraField() const;
    void setExtraField(const char* extra);

    // Переопределяем виртуальный метод print()
    virtual void print() const override;
};

#endif // CTHREE_H
```

CThree.cpp:

```
#include "cthree.h"
#include <cstring>
```

```
#include <iostream>

CThree::CThree() : CTwo(), extraField(nullptr) {
    extraField = new char[1];
    extraField[0] = '\0';
}

CThree::CThree(const char* str, const COne& cOneObj, const char* extra)
    : CTwo(str, cOneObj), extraField(nullptr) {
    if (extra) {
        extraField = new char[strlen(extra) + 1];
        strcpy(extraField, extra);
    } else {
        extraField = new char[1];
        extraField[0] = '\0';
    }
}

CThree::CThree(const CThree& other)
    : CTwo(other), extraField(nullptr) {
    if (other.extraField) {
        extraField = new char[strlen(other.extraField) + 1];
        strcpy(extraField, other.extraField);
    } else {
        extraField = new char[1];
        extraField[0] = '\0';
    }
}

CThree& CThree::operator=(const CThree& other) {
    if (this == &other) {
        return *this;
```

```
    }
    CTwo::operator=(other);
    delete[] extraField;
    if (other.extraField) {
        extraField = new char[strlen(other.extraField) + 1];
        strcpy(extraField, other.extraField);
    } else {
        extraField = new char[1];
        extraField[0] = '\0';
    }
    return *this;
}

CThree::~CThree() {
    delete[] extraField;
}

const char* CThree::getExtraField() const {
    return extraField;
}

void CThree::setExtraField(const char* extra) {
    delete[] extraField;
    if (extra) {
        extraField = new char[strlen(extra) + 1];
        strcpy(extraField, extra);
    } else {
        extraField = new char[1];
        extraField[0] = '\0';
    }
}
```

```
void CThree::print() const {

    CTwo::print();

    std::cout << "CThree::extraField = " << extraField << std::endl;

}
```

CFour.h:

```
#ifndef CFOUR_H
#define CFOUR_H
```

```
#include "cthree.h"

class CFour : public CThree {
private:
    int additionalData;

public:
    // Конструктор по умолчанию
    CFour();

    // Конструктор с параметрами
    CFour(const char* str, const COne& cOneObj, const char* extra, int data);

    // Конструктор копирования
    CFour(const CFour& other);

    // Оператор присваивания
    CFour& operator=(const CFour& other);

    // Виртуальный деструктор
    virtual ~CFour();

    // Методы доступа к новому полю
    int getAdditionalData() const;
    void setAdditionalData(int data);

    // Переопределяем виртуальный метод print()
    virtual void print() const override;
};

#endif // CFOUR_H
```

CFour.cpp:

```
#include "cfour.h"
#include <iostream>
```

```
CFour::CFour() : CThree(), additionalData(0) {
}

CFour::CFour(const char* str, const COne& cOneObj, const char* extra, int data)
    : CThree(str, cOneObj, extra), additionalData(data) {
}

CFour::CFour(const CFour& other)
    : CThree(other), additionalData(other.additionalData) {
}

CFour& CFour::operator=(const CFour& other) {
    if (this == &other) {
        return *this;
    }
    CThree::operator=(other);
    additionalData = other.additionalData;
    return *this;
}

CFour::~CFour() {
}

int CFour::getAdditionalData() const {
    return additionalData;
}

void CFour::setAdditionalData(int data) {
    additionalData = data;
}
```

```
void CFour::print() const {

    CThree::print();

    std::cout << "CFour::additionalData = " << additionalData << std::endl;

}
```

Main.cpp:

```
#include <iostream>
#include "cone.h"
```

```
#include "ctwo.h"
#include "cthree.h"
#include "cfour.h"

// Глобальная функция для демонстрации полиморфизма
void printAll(CTwo* arr[], int n) {
    for (int i = 0; i < n; i++) {
        arr[i]->print();
        std::cout << "-----------------------" << std::endl;
    }
}

int main() {
    std::cout << "===== Лаб. работа №2 (Композиция) =====\n";
    {
        // COne
        COne oneDefault;
        COne oneParam(3.14f, "Hello, COne!");
        COne oneCopy(oneParam);
        oneDefault = oneParam;

        oneDefault.print();
        oneParam.print();
        oneCopy.print();

        // CTwo
        CTwo twoDefault;
        CTwo twoParam("String for CTwo", oneParam);
        CTwo twoCopy(twoParam);
        twoDefault = twoParam;

        twoDefault.print();
```

```
    twoParam.print();
    twoCopy.print();
}

std::cout << "\n===== Лаб. работа №3 (Наследование) =====\n";
{
    // CThree
    COne oneForThree(2.71f, "String in COne");
    CThree threeDefault;
    CThree threeParam("BaseString", oneForThree, "ExtraFieldData");
    CThree threeCopy(threeParam);
    threeDefault = threeParam;

    threeDefault.print();
    threeParam.print();
    threeCopy.print();
}

std::cout << "\n===== Лаб. работа №4 (Виртуальные функции) =====\n";
{
    // CFour
    COne oneForFour(1.23f, "COne in CFour");
    CFour fourDefault;
    CFour fourParam("Base in CFour", oneForFour, "Extra in CFour", 42);
    CFour fourCopy(fourParam);
    fourDefault = fourParam;

    fourDefault.print();
    fourParam.print();
    fourCopy.print();
}
```

```
    std::cout << "\n===== Демонстрация полиморфизма =====\n";

    {

        CTwo* arr[3];

        arr[0] = new CTwo("CTwo example", COne(0.0f, "from arr[0]"));

        arr[1] = new CThree("CThree example", COne(100.0f, "from arr[1]"), "Extra
example");

        arr[2] = new CFour("CFour example", COne(999.9f, "from arr[2]"), "Extra for
four", 777);



        printAll(arr, 3);



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

            delete arr[i];

        }

    }



    return 0;

}
```