Лабораторная работа №2
по курсу «Высокоуровневое программирование» (2 семестр)
«Наследование и иерархия классов»

Цель: приобретение практических навыков проектирования классов и иерархии структур данных.

Задачи:

1. Изучить понятие иерархии структур данных;
2. Познакомиться со способами наследования классов;
3. Изучить понятие интерфейса;
4. Научиться выполнять инициализацию объекта разными способами.

Содержание отчёта:

1. Титульный лист;
2. Цель, задачи работы;
3. Формулировка общего задания;
4. Листинги пользовательских функций, классов и основной программы;
5. Результаты работы;

6. Выводы по работе в целом.

В начало

Основные теоретические сведения

Композиция и Агрегация

Процесс построения сложных объектов из более простых называется композицией объекта. При композиции между двумя объектами представлен тип отношения «имеет» (ПК имеет процессор, материнскую плату и т.д.; стол состоит из столешницы и ножек; билет содержит информацию о сеансе в кино) Композиция и агрегация позволяют реализовать иерархию «целое-часть».

Для реализации композиции объект и часть должны иметь следующие отношения:

• Часть (член) является частью объекта (класса).

• Часть (член) может принадлежать только одному объекту (классу) в моменте.

• Часть (член) существует, управляемая объектом (классом).

• Часть (член) не знает о существовании объекта (класса).

В отношениях внутри композиции объект несет ответственность за существование частей. Часть создается при создании объекта и уничтожается при его уничтожении, то есть их время жизни совпадает.

Для реализации агрегации целое и его части должны соответствовать следующим отношениям:

• Часть (член) является частью целого (класса).

• Часть (член) может принадлежать более чем одному целому (классу) в моменте.

• Часть (член) существует, не управляемая целым (классом).

• Часть (член) не знает о существовании целого (класса).

В композиции:

• Используются обычные переменные-члены внутри класса.

• Используются указатели, если класс реализовывает собственное управление памятью (происходит динамическое выделение/освобождение памяти).

• Класс ответственен за создание/уничтожение своих частей.

В агрегации:

• Используются указатели/ссылки, которые указывают/ссылаются на части вне класса.

• Класс не несет ответственности за создание/уничтожение своих частей

Полиморфизм

Простой

Подобный подход позволяет строить более гибкие и совершенные иерархии классов, переопределяя в производных классах методы в соответствии с требованиями разрабатываемой программы или системы, использующей эти классы.

Сложный

Однако использование переопределенных методов не всегда безопасно.

Результаты явного и опосредованного вызовов метода print() различаются.

При явном вызове для переменных базового и производного класса, а также, если указатели и объекты совпадают по типу, никаких проблем не возникает: вызывается аспект метода класса, к которому принадлежит объект.

Опосредованный вызов может приводить к ошибкам. Объект, для которого вызывается метод, определяется адресом, хранящемся в указателе, который по типу может не совпадать с типом объекта. От этого появляется некорректное поведение.

Фиксация адреса метода на этапе компиляции является отличительной чертой раннего связывания. Поэтому для получения правильного результата необходимо использование сложного полиморфизма, который реализуется механизмом позднего связывания, позволяющего выбирать требуемый аспект полиморфного метода уже на этапе выполнения, когда тип (класс) объекта, для которого вызывается метод точно известен.

Полиморфными переменными (объектами) называются переменные, которым в процессе выполнения программы может быть присвоено значение, тип которой отличается от типа переменной.

В языках с жесткой типизацией такая ситуация может возникнуть при передаче объекта типа класса потомка в качестве фактического параметра подпрограммы, в которой этот параметр описан как параметр типа класса родителя неявно / явно в списке параметров или при работе с указателями, когда указателю на объект класса родителя присваивается адрес объекта класса потомка. Тип полиморфного объекта становится известным только на этапе

выполнения программы. Соответственно при вызове полиморфного метода для такого объекта нужный аспект также должен определяться на этапе выполнения.

Реализации сложного полиморфизма основана на применении виртуальных функций.

Приведение типов

В ООП при работе с объектами возможно временное изменение их типа. При этом различают:

• нисходящее приведение типа;

• восходящее приведение типа.

Приведение типа объекта называют нисходящим, если в его результате указатель или ссылка на объекты базового класса преобразуется в указатель или ссылку на объекты производного, и восходящим, если указатель или ссылка на объекты производного класса преобразуется в указатель или ссылку на объекты базового класса.

Восходящее приведение типа возможно всегда, поскольку все поля и методы предка унаследованы потомком, а, следовательно, ошибок при обращении к компонентам при восходящем приведении возникать не будет. Данный тип приведения типа объекта используют при необходимости вызвать переопределенный в потомке метод базового класса.

Нисходящее приведение допустимо, только для случаев, когда объект производного класса доступен через указатель или ссылку на объект базового, т.е. когда при приведении восстанавливается соответствие объекта своему классу (или при сложной иерархии – возможно его родителю, если указатель или ссылка принадлежали более «древним» предкам). Нисходящее приведение выполняют, чтобы получить через указатель или ссылку базового класса доступ к компонентам, добавленным в производном классе. Такая ситуация встречается сравнительно часто при работе с полиморфными объектами.

Динамическое приведение типа: dynamic_cast <T>(t).

Применяется для нисходящего приведения типов полиморфных объектов.

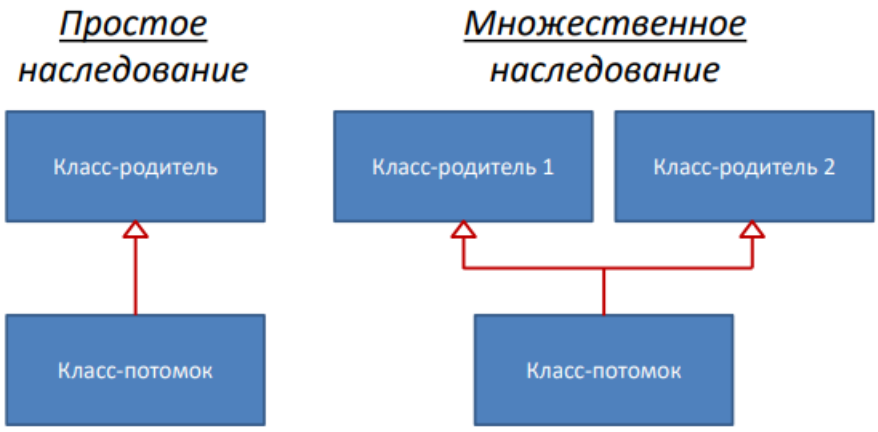
Операнды:

T – указатель или ссылка на класс или void*,

t – выражения типа указателя, причем оба операнда либо указатели, либо ссылки.

Приведение типа осуществляется во время выполнения программы.

Наследование



Наследование – это один из принципов ООП, который позволяет не писать много одинакового кода, а также построить иерархию “Общее-частное”. Концепцию наследования мы рассмотрим на примере класса банковского работника.

```
class BankWorker {

public:
	typedef char* string;
	typedef unsigned int score;

	enum class Gender {
		male,
		female
	};
private:
	string m_name{};
	score m_age{};
	Gender m_gender{};
	score m_salary{};
};
```

Банковский работник является частным случаем профессии человека. Если нам потребуется написать другую профессию, придется с чистого листа создавать второй класс, дублируя общие характеристики из первого класса. Избежать многократного повторения (принцип DRY) можно при помощи вынесения сущности человека за пределы работника профессии, как отдельного родительского класса.

```
class Human {

public:
	typedef char* string;
	typedef unsigned int score;

	enum class Gender {
		male,
		female
	};
```

```
private:
	string m_name{};
	score m_age{};
	Gender m_gender{};
};

class BankWorker : public Human {

private:
	score m_salary{};
};
```

После имени нашего класса мы поставили двоеточие и через модификатор доступа public написали имя класса, от которого хотим унаследоваться.

Модификаторы доступа при наследовании

Существует возможность при наследовании указывать модификаторы: public, private и protected.

Связи между модификаторами наследования и модификаторами членов объекта:

Если мы наследуемся через public, то нам доступны все члены суперкласса, кроме тех, которые помечены – private.

Если мы наследуемся через private, то мы закрываем для себя доступ ко всем членам класса, независимо от их модификатора.

Если мы наследуемся через модификатор protected, то все члены суперкласса, которые не являются private, становятся protected.

Если мы укажем при наследовании приватный модификатор, то мы закроем все члены родительского класса. Таким образом, он будет являться производным классом, но взаимодействовать с корневым классом не сможет.

Модификатор доступа protected служит для наследования. Поля и методы, помеченные этим модификатором, будут открыты, только для производных классов, для доступа через объекты они будут закрыты.

Виды инициализации объекта

Создадим простые классы разных типов и попробуем создать от них объекты разными способами:

```
class T_1 {
public:
	int m_a{}, m_b{}, m_c{};
};

class T_2 {
public:
	int m_a{}, m_b{}, m_c{};

	T_2() {
```

```
		m_a = m_b = m_c = 5;
	}
};

class T_3 {
public:
	int m_a{}, m_b{}, m_c{};

	T_3(int n) {
		m_a = m_b = m_c = n;
	}
};
```

У нас создано три класса, в – первом у нас нет конструктора, во – втором у нас создан конструктор без аргументов (конструктор по умолчанию), в – третьем, у нас конструктор принимает один аргумент. Классы будут предоставлять различные способы инициализации своих объектов.

Простая инициализация объекта – вызов конструктора по умолчанию – эффективна, если конструктор создан явно и не принимает аргументов.

```
T_2 obj_1;
```

Код ниже будет распознан словно мы хотим создать прототип функции, который возвращает тип T_1

```
T_1 obj_2();
```

Однако, если класс имеет конструктор, который принимает какие-то значения, то такая инициализация считается самой эффективной

Конструктор принимает один аргумент. При вызове конструктора происходит присваивание значений полям

```
T_3 obj_3(1);
```

Универсальная форма инициализации была введена в 11-ом стандарте языка и позволяет одинаково для всех объектов выполнить инициализацию. При таком подходе конструктор не вызывается, а объект инициализируется значениями списка, переданного ему.

```
T_1 obj_5{ 1, 2, 3 };
T_1 obj_6 = { 1, 2, 3 };
```

Копирующая инициализация самая затратная в плане ресурсов. В ней создаётся анонимный объект указанного типа, затем он копируется в созданный объект слева и анонимный объект удаляется. Создание двух объектов, их копирование и удаление очень затратная операция, поэтому такая инициализация применяется только в крайней необходимости.

```
T_1 obj_7 = T_1{ 1, 2, 3 };
T_2 obj_8 = T_2();
```

Создание ссылки на объект

```
T_2* p_obj = new T_2();
```

Виртуальные функции и абстрактные классы

Создадим программу «Рисование геометрических фигур».

Выделим абстракции сущностей. Например, квадрат – у него есть цвет и четыре стороны, каждая сторона имеет длину. Также, фигуры можно вращать, смещать и рисовать на экране – это поведения. Заметим, что все фигуры, с которыми мы будем работать, будут обязательно иметь одно поведение – отрисовываться на экране.

Получается, нам нужен какой – то базовый класс фигуры, в котором будут реализованы методы. Но невозможно выделить одну фигуру, от которой получится унаследовать все остальные, к тому же не все фигуры одинаково отрисовываются, для каждой нужна своя реализация метода.

Выход есть, нужно создать базовый класс, который не будет описывать конкретные свойства фигур, т.е. не будет иметь полей, но будет иметь базовые методы работы с этими полями. Такой класс называется – абстрактным классом.

```
class Shape {

public:
	char* m_type_shape{};
	int scale{};

	void draw() {
		cout << "Рисуем фигуру" << endl;
	}

	void rotation();

	void motion();
};
```

От абстрактного класса нельзя создать объект. Его обязательно нужно унаследовать и только от потомка создать объект. Нереализованные функции нужно переопределить в классах потомках, используя особый спецификатор virtual, который позволяет создавать виртуальные функции.

Виртуальная функция – это метод, который при вызове у объекта использует реализацию, которая по иерархии наследования ближе всего к вызываемому объекту. Иными словами, если мы такую функцию перепишем в дочернем классе, то при вызове её у объекта дочернего класса, будет выполнена реализация, которая прописана в этом объекте. Такое поведение реализует последний принцип ООП – полиморфизм.

```
virtual void draw();
```

Чтобы класс стал абстрактным, он должен иметь хотя бы одну чистую виртуальную функцию. Чистая виртуальная функция выглядит так:

```
virtual void draw() = 0;
```

Базовый класс:

```
class Shape {

public:
	char* m_type_shape{};
	int scale{};

	virtual void draw() = 0;

	virtual void rotation() = 0;

	virtual void motion() = 0;
};
```

Мы явно сообщили компилятору, что наш класс является абстрактным, и что все его методы должны быть переопределены в дочернем классе. Унаследуем от него класс фигуры.

```
class Square : public Shape {

public:
	virtual void draw() {
		cout << "Рисуем квадрат" << endl;
	}

	virtual void rotation() {
		cout << "Крутим квадрат" << endl;
	}

	virtual void motion() {
		cout << "Перемещаем квадрат" << endl;
	}

};
```

При создании объекта этого класса мы получим переопределённое поведение методов.

Реализация по умолчанию чистых виртуальных функций

В абстрактных классах можно определять чистые виртуальные функции, это значит, что мы можем задать реализацию по умолчанию, которую дочерний класс может и не менять. Для того, чтобы это сделать необходимо обязательно вынести реализацию из класса, рассмотрим пример:

```
class Shape {

public:
	char* m_type_shape{};
	int scale{};
```

```
	virtual void draw() = 0;

	virtual void rotation() = 0;

	virtual void motion() = 0;
};

void Shape::draw() {
	cout << "Рисуем фигуру" << endl;
}
```

Draw – по-прежнему чистая виртуальная функция, только теперь у неё есть реализация по умолчанию. В переопределении функции мы вызываем её у класса Shape через оператор разрешения области видимости.

```
class CertainShape : public Shape {

public:
	virtual void draw() {
		Shape::draw();
	}
};
```

Стоит отметить, что не следует объявлять все методы виртуальными, виртуальные функции имеют сложный механизм вызова, поэтому уступают по скорости обычным функциям. Делайте это только там, где это действительно нужно.

Модификаторы override и final

- **Override** – явно указывает на то, что виртуальная функция должна быть переопределена в классе родителе.
- **Final** – явно указывает на то, что виртуальная функция не должна быть переопределена в классе родителе. Если этот модификатор указать после имени класса, то такой класс нельзя будет наследовать.

Интерфейсы

Создадим функцию, которая будет определять тип фигуры и рисовать её на экране.

```
void function(Square &sq) {
	cout << sq.m_type_shape << endl;
	sq.draw();
}
```

Функция принимает объект класса квадрат, печатает этот тип и вызывает функцию отрисовки. Допустим, понадобится написать идентичную функцию для окружности, прямоугольника, ромба, овала, тогда придётся делать множество перегрузок функций с одинаковым телом.

Однако возможно создать функцию, которая принимает любой объект, обладающий интерфейсом, с которым эта функция умеет работать. Абстрактный класс фигуры в качестве интерфейса волне подойдёт. Добавим новый класс.

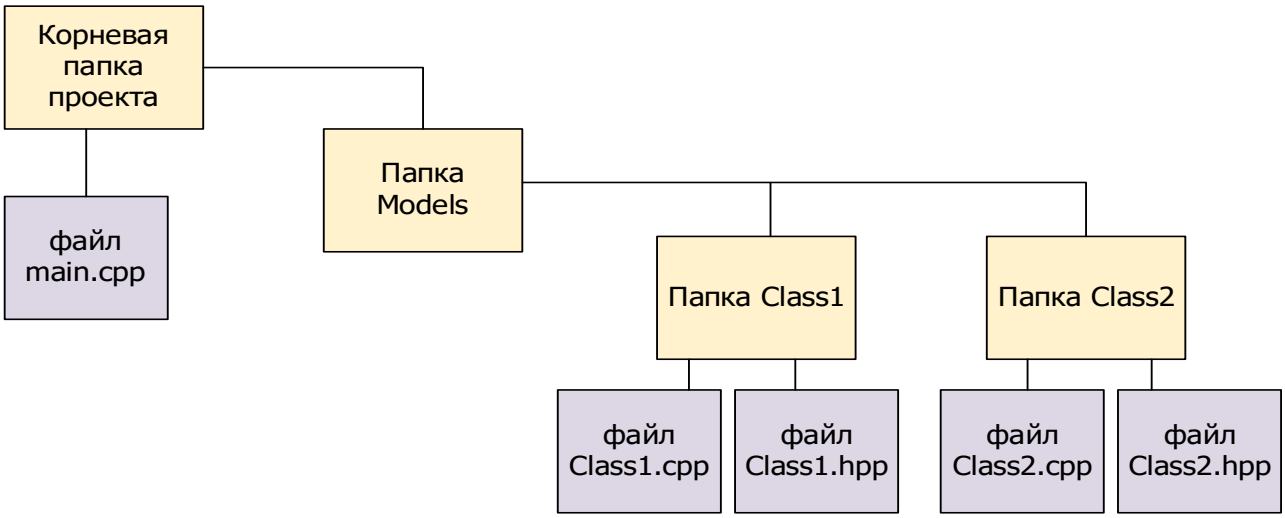
```
class Circle : public Shape {

public:
	virtual void draw() {
		cout << "Рисуем окружность" << endl;
	}

	virtual void rotation() {
		cout << "Крутим окружность" << endl;
	}

	virtual void motion() {
		cout << "Перемещаем окружность" << endl;
	}

};

int main() {

	Circle circle;
	Square square;

	function(circle);
	function(square);

	return 0;
}
```

Мы создали два объекта разных классов, но смогли передать их в одну функцию, которая, зная **интерфейс** (а именно виртуальные функции, которые обязаны быть в наследуемых классах) объектов умело их обработала.

Интерфейс в языке C++ это класс, у которого есть чистые виртуальные функции, но нет их реализации и полей. Интерфейсы созданы для того, чтобы обрабатывать общее поведение наследуемых объектов. Когда класс наследует интерфейс, говорят, что он реализует его. В отличие от наследования абстрактного класса, методы реализуемого интерфейса обязаны быть переопределены в дочернем классе.

Задание

Создайте в корне проекта папку: Models. В этой директории будут находиться все дальнейшие папки сущностей.



Задача 1

Создайте абстрактный класс “пользователь” с полями: имя, фамилия, возраст, логин и пароль. А также с чистыми виртуальными функциями, поведение которых определено по умолчанию.

Задача 2

Создайте папку для самостоятельной сущности (см. таблицу вариантов) и реализуйте в ней класс со следующими полями (см. столбец - Свойства).

Добавьте папки для наследуемых сущностей (см. таблицу вариантов). В соответствующих папках опишите классы сущностей, унаследовав их от абстрактного класса “пользователь”. Дополнительно (см. столбец - Дополнительно)

Реализуйте конструкторы классов, геттер/сеттер методы по необходимости.

Задача 3

Реализуйте тестовую логику классов в файле main, чтобы оценить работоспособность приложения. Добавьте эту функциональность в меню.

Таблица Вариантов

Вариант	Тема	Наследуемые сущности	Дополнительно	Самостоятельные сущности	Свойства
1.	Институт	Студент, Преподаватель	в классе студент опишите поля: курс и средний балл.	Курс	название курса, оценка за курс, описание курса
2.	Парикмахерская	Сотрудник, Клиент	в классе сотрудник опишите поле: должность, а в классе клиент: услуга.	Услуга	название услуги, цена, описание услуги
3.	Школа	Ученик, Учитель	в классе ученик опишите поля: оценка и класс	Класс	название класса, смена (утренняя / дневная), предметное направление
4.	Автосалон	Сотрудник, Клиент	в классе сотрудник опишите поле: должность, а в классе клиент: услуга.	Автомобиль	марка авто, цена, год выпуска, описание и характеристики
5.	Банк	Сотрудник, Клиент	Дополнительно в классе сотрудник опишите поле: должность, а в классе клиент: депозиты	Вклад	название вклада, клиент, сумма, процент
6.	Мастерская	Сотрудник, Клиент	в классе сотрудник опишите поле: должность, а в классе клиент: услуга.	Услуга	название услуги, цена, описание услуги
7.	Театр	Кассир, Зритель	в классе зритель опишите поле билет.	Сеанс, Билет	Сеанс: время начала, время окончания, название постановки и номер зала. Билет: сеанс с указанием места
8.	Больница	Врач, Пациент	в классе врач укажите поле специальность, а в классе пациент поле номера амбулаторной карты	Амбулаторная карта, Консультация	Карта: номер карты, история болезней (в виде массива). Консультация: время приёма, рекомендации и имя врача

9.	Почта	Сотрудник, Клиент	в классе сотрудник опишите поле: должность, а в классе клиент: бандероль	Бандероль	кодовый номер, дата отправления, адрес назначения.
10.	Завод	Сотрудник, Поставщик	в классе сотрудник опишите поле: должность, а в классе поставщик: тип поставляемой продукции	Продукт	название продукции, ее тип, цена за единицу и кол – во
11.	Почтовый клиент	Пользователь, Администратор	в классе пользователь опишите поле: список писем	Письмо	Текст сообщения, имя отправителя, имя получателя, дата отправления
12.	Магазин	Кассир, Покупатель	в классе кассир опишите поле: количество пробитых чеков, сумма их итогов	Товар, Чек	Товар: название товара, описание, цена. Чек: товары, количество позиций, итог, кассир, дата.
13.	Библиотека	Сотрудник, Читатель	в классе читатель опишите поле: читательский билет	Книга, Читательский билет	Книга: название, автор, описание, год выпуска Билет: количество книг, дата и срок выдачи

Контрольные вопросы

1. Что такое иерархия сущностей?
2. Приведите пример иерархии сущностей из реальной жизни?
3. Как воспроизводится иерархия классов?
4. Что такое наследование?
5. Как выполняется наследование класса?
6. Виды наследования, модификаторы наследования.
7. Связь модификаторов доступа и модификаторов наследования.
8. Назовите виды инициализации объектов класса.
9. Что такое виртуальная функция?
10. Что такое чистая виртуальная функция и как её создать?
11. Что такое абстрактный класс?
12. Чем абстрактный класс отличается от обычного?
13. Как создать абстрактный класс?
14. Что такое интерфейс?
15. Как использовать интерфейс?
16. Как переопределять методы класса?
17. Как устроено переопределение виртуальных функций?

Список литературы

1. Курс лекций доцента кафедры ФН1-КФ Пчелинцевой Н.И.

2. Программирование на языке высокого уровня C/C++ [Электронный ресурс]: конспект лекций / – Электрон. текстовые данные. – М.: Московский государственный строительный университет, Ай Пи Эр Медиа, ЭБС АСВ, 2016. – 140 с. – Режим доступа: http://www.iprbookshop.ru/48037.

В начало