Экзамен АиП

C++

1. Основные отношения между классами: наследование, композиция, наполнение, зависимость. Примеры диаграмм классов.

Наследование - создание производных классов на основе базового, класс-наследник получает свойства и методы базового класса, может переопределять и добавлять новые. На диаграмме классов обозначается незакрашенной стрелкой
Композиция - отношение между классами, когда один из них включает в себя объекты другого (1 или больше) и полностью управляет их жизненным циклом. Включаемый объект может существовать только как часть контейнера. Обозначается на диаграмме стрелочкой с закрашенным ромбиком. Пример: автомобиль и двигатель, без которого он не будет работать.
Наполнение(агрегация) - отношение между классами, при котором один из них содержит или не содержит объекты другого (0 или больше), в отличие от композиции объекты второго класса могут существовать без объекта первого. Пример: автомобиль и колёса, которые могут быть в количестве от 0 до 4.
Зависимость(ассоциация) - объекту одного класса ставится в соответствие некоторое количество объектов другого, на диаграмме обозначается простой стрелкой с указанием с каждой стороны количества объектов. В этом случае один класс содержит или использует объекты другого.

2. Конструкторы: инициализирующий, без параметров (инициализирующий и неинициализирующий), копирующий, перемещения. Примеры.

Конструкторы представляют специальную функцию, которая имеет то же имя, что и класс, которая не возвращает никакого значения и которая позволяет инициализировать объект класса во время его создания и таким образом гарантировать, что поля класса будут иметь определенные значения. При каждом создании нового объекта класса вызывается конструктор класса. Конструкторов может быть не сколько, каждый может иметь любые параметры и вызывать другие конструкторы.
По умолчанию компилятор при компиляции классов генерирует специальный конструктор - **конструктор копирования**, который позволяет создать объект на основе другого объекта (по сути копирует объект). Конструктор копирования по умолчанию копирует значения полей объекта, в новый объект. Конструктор копирования должен принимать в качестве параметра объект того же класса. Причем параметр лучше принимать по ссылке, потому что при передаче по значению компилятор будет создавать копию объекта. А для создания копия объекта будет вызываться конструктор копирования, что приведет бесконечной рекурсии.

```
Person(const Person &p)
{
    name = p.name;
    age = p.age + 1;
}
// если нужно удалить копирующий конструктор
cppPerson(const Person &p) = delete;
```

Конструктор перемещения (move constructor) представляет альтернативу конструктору копирования в тех ситуациях, когда надо сделать копию объекта, но копирование данных нежелательно - вместо

копирования данных они просто перемещаются из одной копии объекта в другую.

```
class Message
{
    public:
    Message(const char* data, unsigned count)
    {
        size = count;
        text = new char[size];  // выделяем память
        for(unsigned i{}; i < size; i++)    // копируем данные
        {
    text[i] = data[i];
}
id = ++counter;
std::cout << "Create Message " << id << std::endl;
    }
    // конструктор копирования
    Message(const Message& copy) : Message{copy.getText(), copy.size }  // обращаемся к
стандартному конструктору
    {
    std::cout << "Copy  Message " << copy.id << " to " << id << std::endl;
}
    ~Message() // деструктор
    {
        std::cout << "Delete Message " << id << std::endl;
        delete[] text;  // особождаем память
    }
    char* getText() const { return text; }
    unsigned getSize() const { return size; }
    unsigned getId() const {return id;}
private:
    char* text { };  // текст сообщения
    unsigned size { };    // размер сообщения
    unsigned id { };  // номер сообщения
    static inline unsigned counter{};   // статический счетчик для генерации номера объекта
};
```

3. Список инициализации. Примеры.

Поля класса, являющиеся константами нельзя инициализировать в конструкторе, потому что они являются неизменяемыми, поэтому используются списки инициализации:

```
class Person
{
    const std::string name;
    unsigned age;
    public:
    void print()
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
    Person(std::string p_name, unsigned p_age) : name(p_name), age(p_age)
    { }
};
```

В этом случае поля `name` и `age` получат значения параметров `p_name` и `p_age` ещё до того, как начнёт выполняться тело конструктора `Person`. При использовании списков инициализации важно учитывать, что передача значений должна идти в том порядке, в котором константы и переменные определены в классе.

4. Наследование. Особенности описания конструкторов производных классов. Примеры.

Наследование - создание производных классов на основе базового, класс-наследник получает свойства и методы базового класса, может переопределять и добавлять новые.
При наследовании конструкторы **не наследуются**, и если базовый класс содержит только конструкторы с параметрами, то производный класс должен вызывать в своем конструкторе один из конструкторов базового класса. После списка параметров конструктора производного класса через двоеточие идет вызов конструктора базового класса, в который передаются значения параметров.

```
# include <iostream>
class Person
{
public:
    Person(std::string name, unsigned age);
    void print();
private:
    std::string name;
    unsigned age;
};
class Employee : public Person
{
public:
    Employee(std::string name, unsigned age, std::string company): Person(name, age) //
вызов конструктора базового класса
    {
        this→company = company;
    }
private:
    std::string company;
};
```

Если конструктор полностью совпадает с конструктором базового класса его можно просто подключить:

```
class Employee : public Person
{
public:
    using Person::Person;
};
```

5. Композиция. Особенности описания конструкторов классов-агрегатов. Примеры.

Композиция - отношение между классами, когда один из них включает в себя объекты другого (1 или больше) и полностью управляет их жизненным циклом. Включаемый объект может существовать только как часть контейнера. Обозначается на диаграмме стрелочкой с закрашенным ромбиком. Пример: автомобиль и двигатель, без которого он не будет работать. Для включаемых классов обязательным условием является существование конструктора по умолчанию.

```
class Point2D
{
private:
    int m_x;
    int m_y;
public:
    // Конструктор по умолчанию
    Point2D() : m_x(0), m_y(0) {}
    // Специфический конструктор
    Point2D(int x, int y) : m_x(x), m_y(y) {}
    void setPoint(int x, int y)
    {
        m_x = x;
        m_y = y;
    }
};
class Line
{
private:
    std::string name;
    Point2D a, b;
}
```

6. Наполнение. Особенности описания конструкторов и деструкторов классов-агрегатов. Примеры.

Наполнение(агрегация) - отношение между классами, при котором один из них содержит или не содержит объекты другого (0 или больше), в отличие от композиции объекты второго класса могут существовать без объекта первого. Пример: автомобиль и колёса, которые могут быть в количестве от 0 до 4.

```
class Wheel {
private:
        int id;
        static int n;
public:
        Wheel() {
                id = ++n;
        }
        int getID() { return id; }
};

int Wheel::n = 0;

class Car {
private:
        Wheel* wheels;
public:
        Car(Wheel* w) : wheels(w) {};
        ~Car() {
                std::cout << "Объект Car уничтожается\n";
        }
};

int main()
{
```

```
        Wheel* wheels = new Wheel[4]; // массив колёс
        {
                Car car(wheels); // автомобиль с колёсами
        } // объект car удаляется при выходе из блока
        for (int i = 0; i < 4; ++i)
                std::cout << wheels[i].getID() << "\n"; // колёса продолжают существовать
        delete[] wheels;
}
```

7. Полиморфное наследование в языке C++. Раннее и позднее связывание: описание в программе и механизмы реализации. Примеры.

Полиморфизм в языке C++ позволяет работать с разными данными одинаковым образом. Несколько классов наследуют и переопределяют свойства одного общего, в результате чего получают общий интерфейс для работы с ними. В зависимости от способа переопределения происходит раннее или позднее связывание методов с объектами и типами, для которых они будут вызваны.
Раннее связывание - это когда обычный метод явно переопределяется для класса-наследника и вызывается для всех объектов соответствующего типа, в том числе по типу указателя на объект(если явно преобразовать тип указателя, для него будет вызван уже другой метод, независимо от того, объект какого типа реально находится по данному адресу).
Позднее связывание - это когда метод родительского класса объявляется виртуальным и классы-наследники его переопределяют, в этом случае для данной иерархии классов создаётся специальная таблица виртуальных методов и при каждом вызове метода для объекта по указателю происходит получение реального типа хранимого объекта из специального скрытого поля и вызов метода по таблице для него.
В данном примере для метода `PrintInfo()` используется раннее связывание, а для метода `Sound` - позднее связывание:

```
class Animal {
public:
        virtual void Sound()=0; // чистая виртуальная функция, требуется переопределение в
потомках
        void PrintInfo() { // обычный метод
                std::cout << "Class Animal\n";
        };
};

class Dog : public Animal {
public:
        void Sound() override {
                std::cout  << "Ruff-ruff\n";
        }
        void PrintInfo() {
                std::cout << "Class Dog\n";
        }
};

class Cat : public Animal {
public:
        void Sound() override {
                std::cout << "Meow-meow\n";
        }
        void PrintInfo() {
                std::cout << "Class Cat\n";
        }
```

```
};

int main() {
        Animal *animals[2];
        animals[0] = new Cat();
        animals[1] = new Dog();
        for (auto i : animals) {
                i->Sound();             // для каждого объекта вызовется свой метод
                i->PrintInfo(); // для всех объектов вызовется метода класса Animal
        }
        // при позднем связывании для любого преобразования типа указателя
        // всё равно вызовется метода, соответствующий реальному типу объекта
        ((Dog*)animals[0])->Sound();
        ((Dog*)animals[1])->Sound();
        // при раннем связывании и явном преобразовании вызовется метод типа указателя
        ((Cat*)animals[0])->PrintInfo();
        ((Dog*)animals[1])->PrintInfo();
}
```

8. Контейнерные классы. Примеры диаграмм классов.

9. Qt. Принципы создания графических интерфейсов. Использование C++. контейнеров.

10. Qt. Сигналы и слоты.

11. Qt. События и обработчики событий.

12. Полиморфное наследование. Сложный полиморфизм. Пример.