



ГНИУ ВШЭ 2020

# Одинокое наследование

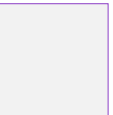
О, не знай сих страшных слов

# Что такое наследование?

- Наследование – это возможность создавать новый класс на основе уже существующего, с сохранением функциональности и компонентов данных базового класса. Уже существующий класс (на основе которого мы создаём) будем называть родительским, а класс, основанный на родительском – дочерним (классом-наследником).
- Наследование – это отношение **IS A** (является) между классами. Если мы говорим, что класс A – наследник класса B, то всякий объект класса A является объектом класса B.
- $A \text{ is a } B \leftrightarrow \forall a \in A: a \in B$
- Наиболее очевидные примеры наследования в реальной жизни можно привести из биологии, эта наука буквально пронизана отношением наследования между классами (видами животных).

## Пример наследования

- Тигр → Семейство кошачьих → Млекопитающее  
(всякий тигр принадлежит семейству кошачьих, а всякое кошачье – это млекопитающее)
- Человек → Примат → Обезьяна  
(всякий человек является приматом, а всякий примат – это обезьяна)
- Основная сложность в CPN№6 – правильно выстроить 3 класса и связать их в одну иерархию.





# Связь наследования и модификаторов доступа

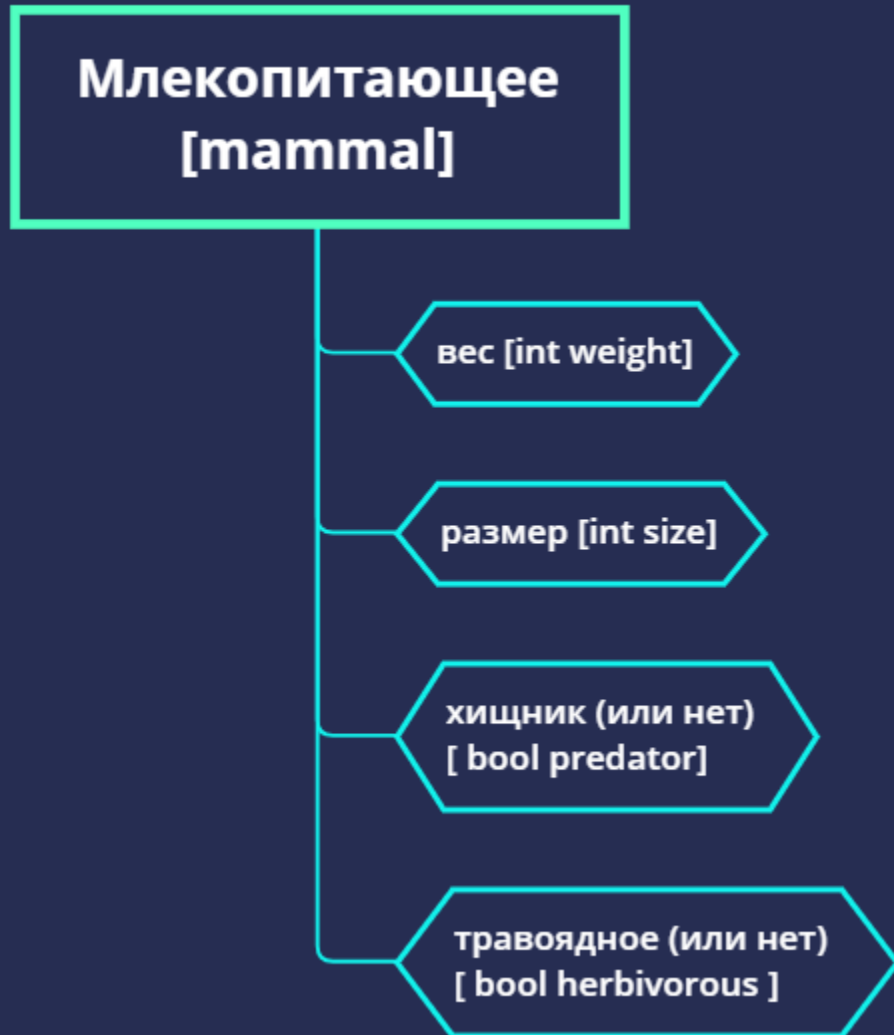
## Модификаторы доступа

- В C++ существует 3 вида модификаторов доступа:
- **public** (публичные) члены класса: можно вызвать извне класса, можно наследовать
- **private** (приватные) члены класса: нельзя вызывать извне класса, нельзя наследовать
- **Protected** (защищённые) члены класса: нельзя вызывать извне класса, можно наследовать

## Виды наследования

- В C++ существует 3 вида наследования:
- **Public** (публичное) наследование: модификаторы доступа членов родительского класса остаются неизменными в дочернем классе
- **Private** (приватное) наследование: все члены родительского класса становятся приватными в дочернем классе
- **Protected** (защищённое) наследование: все члены родительского класса становятся защищёнными (protected) в дочернем классе
- Замечание: приватные члены родительского класса (поля и методы) не наследуются в любом случае!

# Тигр. Иерархия одиночного наследования. Класс «Млекопитающее»



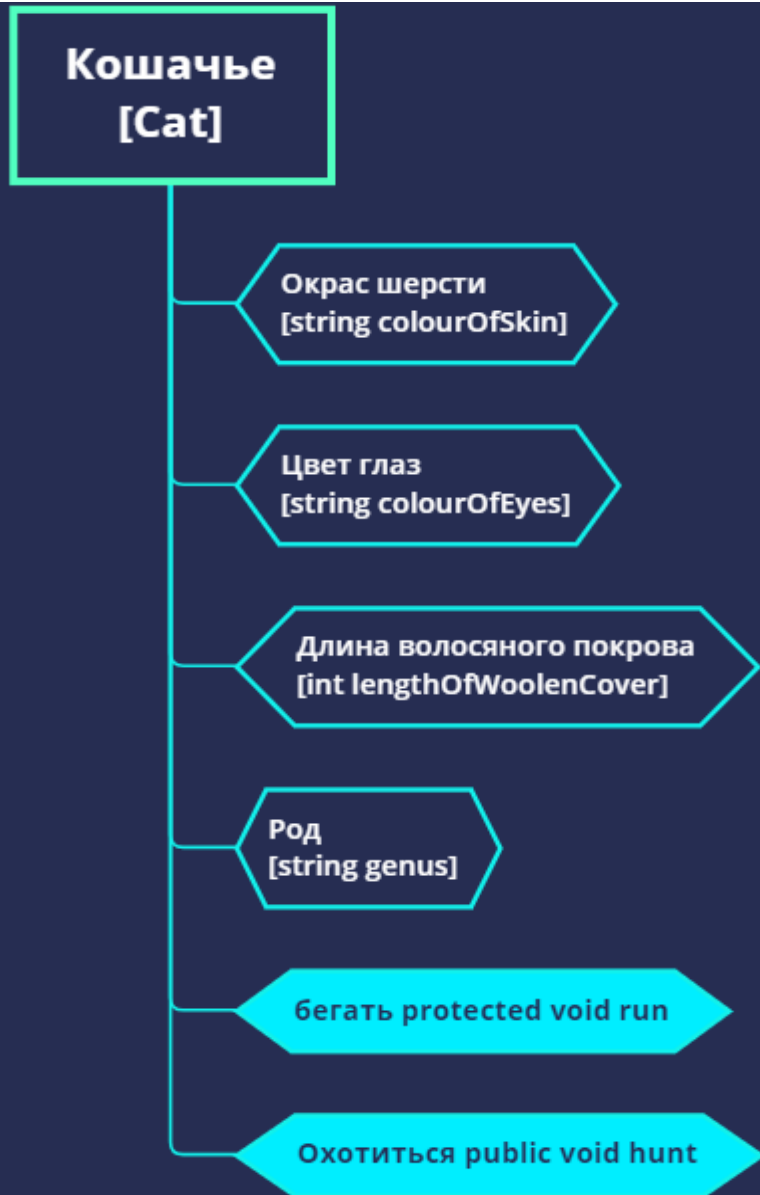
Примечание: в классе ***Mammal*** будут определены только 3 метода:

- Конструктор по умолчанию
- Конструктор копирования
- Виртуальный деструктор

Нет смысла определять какие-то другие методы внутри класса ***Mammal***, поскольку мы находимся на самом верхнем уровне абстракции.

# Тигр. Иерархия одиночного наследования.

## Класс «Кошачье»



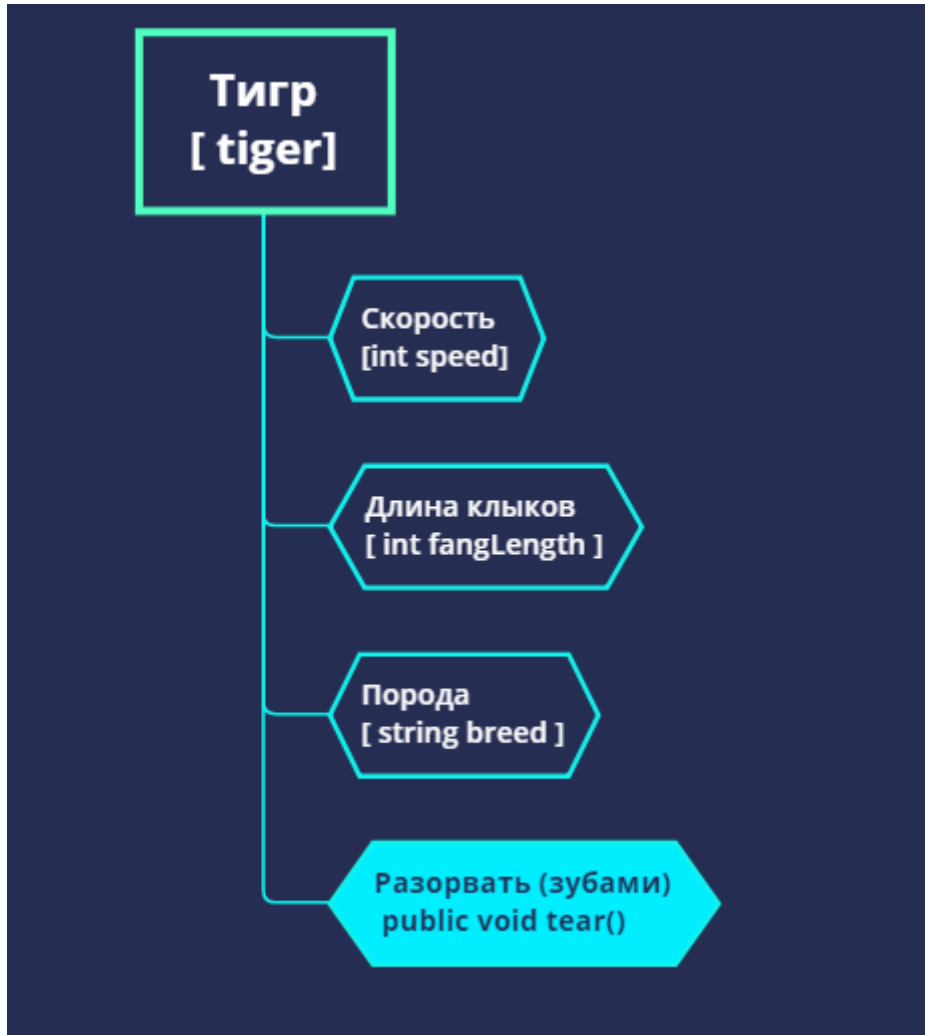
**Примечание:** методы базового класса должны быть либо публичными, либо защищёнными (приватным метод стоит делать лишь в том случае, если Вы точно знаете, что не будете его наследовать, но подобную ситуацию в рамках учебного проекта придумать очень сложно).

Методы **run** и **hunt** в классе **Cat** должны быть виртуальными. Если метод класса виртуальный, то его можно переопределить (написать альтернативную реализацию) в дочернем классе. Таким образом, виртуальные методы – это способ обеспечения полиморфизма методов класса.

В некоторых языках программирования все методы родительских классов можно переопределять в дочерних классах без каких-либо дополнительных усилий, т.е. они являются виртуальными по умолчанию. (например, в Python).

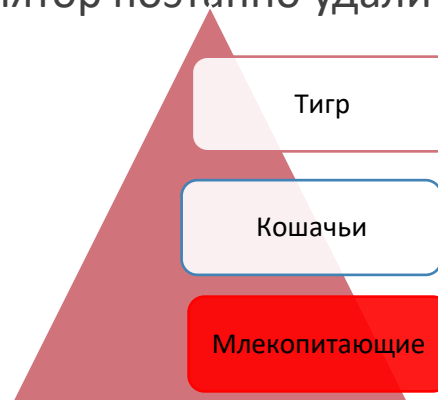
# Тигр. Иерархия одиночного наследования.

## Класс «Тигр»



# Виртуальный деструктор

- Деструктор – это особый метод класса, который вызывается при уничтожении объекта. Виртуальный деструктор должен быть определён для всех классов, кроме тигра (класса, у которого уровень абстракции ниже всех, самое узкое множество).  
Если мы не будем определять виртуальный деструктор (например, у класса «Кошачьи» и скажем, что `Mammal* m = new Tiger(); // возвращаем указатель на млекопитающее (тигра)`  
`delete m; // удаление указателя на объект`  
) то произойдёт следующее:  
Компилятор при удалении **указателя на объект класса** «Млекопитающее» удалит ту часть объекта, которая связана с классом «Млекопитающее», а остальное он трогать не будет, т.к. у класса `Mammal` нет виртуального деструктора.
- Все остальные поля класса (условно говоря, верхняя часть пирамидки) останутся висеть мёртвым грузом в оперативной памяти и возникнет неопределённое поведение (undefined behaviour).  
Для того, чтобы избежать этой ситуации, необходимо определить виртуальные деструкторы для классов «Кошачьи» и «Млекопитающие». Тогда компилятор поэтапно удалит каждую из трёх частей объекта, а не только ту, которая лежит в основании пирамиды.



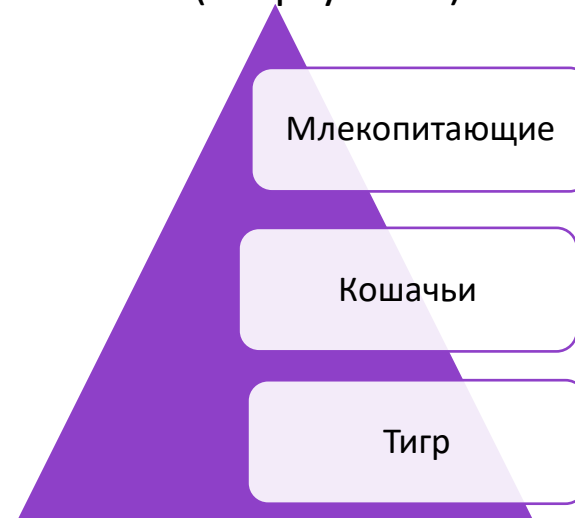
# Порядок вызовов конструктора и деструктора при одиночном наследовании

- При создании объекта класса «Tiger» первым будет вызван конструктор класса «Млекопитающие» (Mammal), затем – «Кошачьи» (Cats) и последним – конструктор класса «Тигр» (Tiger).
- Вызов деструкторов при уничтожении объекта класса Tiger происходит в обратном порядке – сначала вызывается деструктор класса «Tiger», затем – деструктор класса «Cats» и последний – деструктор класса «Млекопитающие» (при условии, что определены виртуальные деструкторы классов «Cats» и «Mammal»).
- Отсутствие объявления виртуального деструктора в базовых классах (родителях) – это прямой путь к undefined behaviour и утечкам памяти.

Порядок вызова деструкторов при уничтожении объекта (сверху вниз)



Порядок вызова конструкторов при создании объекта (сверху вниз)





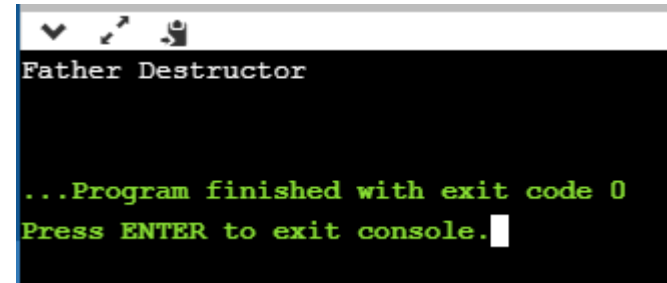
# Коварный виртуальный деструктор

- Ситуация с отсутствием вызова деструкторов производных классов возникает только при **полиморфном удалении**. Здесь, при попытке удалить указатель на отца (который, на самом деле, является указателем на сына) будет вызван только деструктор отца, а половинка сына так и останется жить где-то в оперативной памяти.

```
class Father
{
public:
    ~Father()
    {
        cout << "Father Destructor\n";
    }
};

class Son:public Father
{
public:
    ~Son()
    {
        cout<< "Son Destructor\n";
    }
};

int main()
{
    Father* s = new Son();
    delete s;
}
```



Для иерархии из трёх классов достаточно, чтобы виртуальным был деструктор самого базового класса (в нашем примере – это «Млекопитающие»)

# Коварный виртуальный деструктор

- А такой код будет работать вполне корректно:

```
class Father
{
public:
    ~Father()
    {
        cout << "Father Destructor\n";
    }
};

class Son:public Father
{
public:
    ~Son()
    {
        cout<< "Son Destructor\n";
    }
};

int main()
{
    Son s;
}
```

```
❯ clang++-7 -pthread -std=c++17 -o main main.cpp
❯ ./main
Son Destructor
Father Destructor
❯
```

Видим, что деструкторы были вызваны в правильном порядке, несмотря на то, что деструктор базового класса **Father** не является виртуальным, потому что полиморфного удаления тут нет.

При удалении указателя на сына, который является указателем на сына, поведение также будет корректно.

```
int main()
{
    Son* s = new Son();
    delete s;
}
```