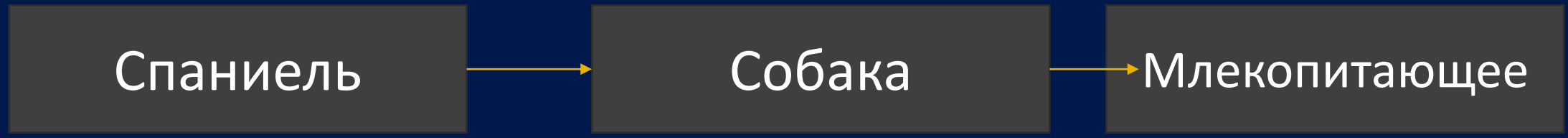
СР 6

Наследование

Наследование в C++

- Отношение наследования – это отношение **IS A** («является», «есть») между классами.
  Пример:
  Всякий спаниель – собака, всякая собака – млекопитающее



Виды наследования C++

Публичное наследование

Публичные (public) и защищённые (protected) члены класса наследуются **без изменения уровня доступа к ним**
(то, что было публичным в классе-родителе, останется публичным в классе наследнике и то, что было защищённым в классе-родителе, останется защищённым в дочернем классе)

- **Частное (private) наследование**

Все публичные и защищённые члены родительского класса становятся приватными в дочернем классе

Виды наследования в C++

- По аналогии: защищённое наследование – члены родительского класса становятся защищёнными в дочернем классе.

  Важно: приватные члены родительского класса унаследовать не получится, поэтому, все члены родительского класса должны быть либо публичными (public) ,но это плохо, т.к. в этом случае кто угодно сможет изменить свойства объекта, либо защищёнными (protected)

Модификаторы доступа членов класса		
Public	**Private**	**Protected**
Можно наследовать	Нельзя наследовать	Можно наследовать
Можно вызвать извне класса пример: Tiger t1(…); tiger.run();	Нельзя вызывать извне класса	Нельзя вызывать извне класса

СР 6

Проектирование родительских классов

FR

- Выстроим иерархию наследования из двух классов для АТД «Тигр» учитывая, что каждый тигр принадлежит семейству кошачьих.
  Выделим основные характеристики представителей семейства кошачьих

```
class Cats {
protected:
    int age; // Age of a cat
    int size; // size of a cat
    bool isHungry; // A cat can be hungry or not
    string colourOfSkin;
    string colourOfEyes;
    string genus; // Genus: Felinae, Pantherinae,Proailurinae
```

Конструктор родительского класса

- Поскольку в регламенте СРП№6 мы сами можем выбрать способ конструирования объекта (инициализация из файла либо «по-старинке», как в СРП№4), реализуем оба варианта. Однако, в качестве примера для лекционного материала я выбрал способ без чтения файла, поскольку он более нагляден.

```
Cats(int age=5, int size=20, bool isHungry=true, string colourOfSkin="red", string
colourOfEyes="white"):
        age(age), size(size), isHungry(isHungry), colourOfSkin(colourOfSkin),
colourOfEyes(colourOfEyes)
{}
```

Здесь происходит инициализация каждого поля родительского класса с помощью списков инициализации (то, что идёт после двоеточия, называется списком инициализации

Конструктор копирования дочернего класса

FR

```
// copying constructor
Tiger(Tiger const& instance):
fangLength(instance.fangLength), speed(instance.speed), breed(instance.breed),
        averageDailyCourse(instance.averageDailyCourse), Cats(instance)
{}
```

Здесь также инициализируем поля класса и вызываем конструктор копирования родительского класса

Структура дочернего класса

```
class Tiger : public Cats
{
    int fangLength; // length of fangs
    int speed; // Speed of a tiger
    string breed; // Breed of a tiger
    int averageDailyCourse; // Average Daily Course
```

В дочерний класс вынесем те характеристики, которые специфичны и значимы именно для тигра
Запись вида `class A : public B` означает,
что класс А является наследником класса В (а класс В – родителем класса А),
public – тип наследования (как уже было оговорено ранее, наследование может быть public, private и protected)

Конструктор дочернего класса

```
Tiger(int age=5, int size=10, bool isHungry=false, string colourOfSkin="orange", string
colourOfEyes="brown",
      int fangLength=35, int speed=700, string breed="Amur", int averageDailyCourse=34) :
        Cats(age, size, isHungry, colourOfSkin, colourOfEyes), speed(speed),
        fangLength(fangLength), breed(breed), averageDailyCourse(averageDailyCourse)
{}
```

Здесь запись вида `Cats(age, size, isHungry, colourOfSkin, colourOfEyes)`

- Вызов конструктора родительского класса, всё остальное – аналогично конструктору родительского класса (списки инициализации). Т.е. после создания объекта родительского класса происходит инициализация **всех собственных** полей дочернего класса

Конструктор дочернего класса

- **Замечание 6.1:** собственными полями класса будем называть такие поля дочернего класса, которых нет ни в одном из родительских классов.

- Аналогично и для конструктора копирования: в конструкторе копирования класса-наследника сначала выполняется вызов конструктора копирования базового (родительского) класса и уже потом – инициализация собственных полей дочернего класса, хотя порядок может быть и другим: сначала инициализируем собственные поля, а потом вызываем конструктор родительского класса

```
// copying constructor
Tiger(Tiger const& instance) :
        Cats(instance), fangLength(instance.fangLength), speed(instance.speed), breed(instance.breed),
        averageDailyCourse(instance.averageDailyCourse)
{}
```

Виртуальные методы класса

FR

- Виртуальные методы класса – это такие методы класса, которые можно переопределить в его наследниках, поэтому методы **родительского** класса (кроме конструктора, потому что конструкторы не наследуются) должны быть виртуальными. Для объявления виртуальной функции используется ключевое слово **virtual**. Таким образом, виртуальные функции – один из способов реализации полиморфизма (а именно, перегрузки методов) в языке C++.

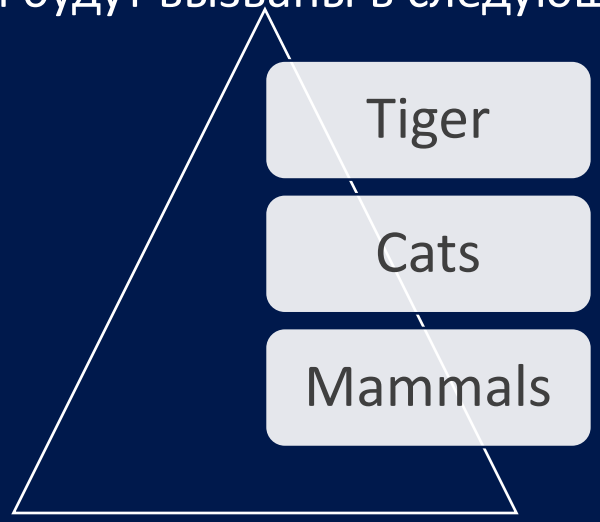
```
virtual void run(int speed = 60) {
    cout << "Running. Should be overriden in childs" << endl;
}
```

```
public:
    virtual ~Cats() {
        cout << "Calling virtual destructor of cats" << endl;
    }

    virtual void hunt() {
        cout << "Hunting. Should be overriden in childs" << endl;
        run(40);
    }
```

Порядок вызова конструкторов при создании класса

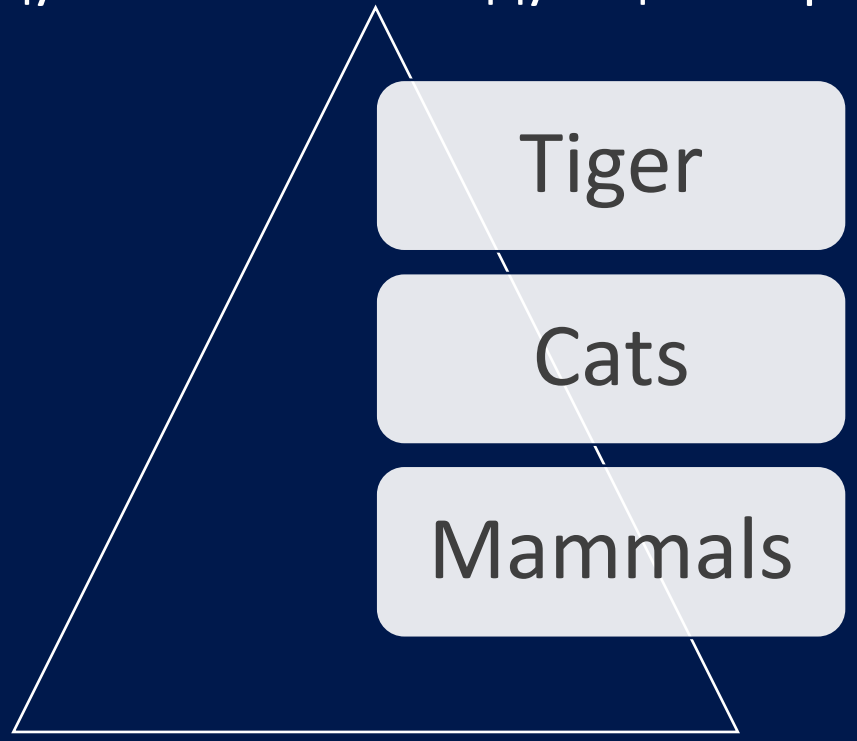
- Конструктор каждого дочернего класса должен вначале вызвать конструктор родительского класса и, после инициализации полей родительского класса, взывать свой конструктор и проинициализировать собственные поля.
- Так, если класс **Tiger** является наследником класса **Cats**, а класс **Cats** является наследником класса **Mammals**, то конструкторы будут вызваны в следующем порядке (**снизу-вверх):**



- Итак, конструкторы вызываются от родителя – к потомку, от общего – к частному.

Порядок вызова деструкторов

- Деструкторы вызываются в обратном порядке – от потомка – к родителю, от частного – к общему. Так, в уже приведённом примере, при уничтожении объекта, деструкторы будут вызваны в следующем порядке **(сверху вниз):**



Зачем делать деструктор базового класса виртуальным?

- В классе Cats мы определили виртуальный деструктор:

```
public:
    virtual ~Cats() {
        cout << "Calling virtual destructor of cats" << endl;
    }
```

Виртуальность деструктора базового класса обеспечивает корректный порядок уничтожения объекта – сначала будет вызван деструктор дочернего класса, а затем – деструктор родителя. Если бы деструктор базового класса не был виртуальным, то мы могли бы смоделировать ситуацию, при которой был бы вызван **только** деструктор базового класса.

- В качестве примера приведу следующий код:

Здесь при удалении указателя на родителя мы не удалим ту часть данных, которая относится только к наследнику (массив), что вызовет утечку памяти.

```cpp
#include <iostream>

class Parent
{
public:
    ~Parent() // Non-virtual destructor
    {
        std::cout << "Calling ~Parent()" << std::endl;
    }
};

class Child: public Parent
{
private:
    int* arr;

public:
    Child()
    {
        arr = new int[10];
    }

    ~Child() //Non-virtual destructor
    {
        std::cout << "Calling ~Child()" << std::endl;
        delete[] arr;
    }
};

int main()
{
    Child *child = new Child();
    Parent *parent = child;
    delete parent; // Memory leak
    return 0;
}
```

Переход от двух классов к трём

FR

- Переход от иерархии из двух классов к иерархии из трёх классов тривиален: создадим класс Mammal, состоящий из конструкторов (по умолчанию и копирования) и виртуального деструктора

```
class Mammal {
protected:
    int age; // Age of a mammal
    int size; // Size of a mammal
    bool isHungry; // A mammal can be hungry or not
    string family; // family: like cats or bears
public:
    // Default constructor
    Mammal(int age=10, int size=20, bool isHungry=false, string family=""):
            age(age), size(size), isHungry(isHungry), family(family) {}
// Copy constructor
    Mammal(const Mammal& m):
            age(m.age), size(m.size), isHungry(m.isHungry), family(m.family) {}
// Virtual destructor
    virtual ~Mammal() {
        cout << "Calling virtual destructor of Mammal" << endl;
    }
};
```

Изменения в дочерних классах

- Теперь часть полей класса Cats перешла в класс Mammals, поэтому их нужно убрать из класса Cats

```
class Cats: public Mammal {
protected:
    string colourOfSkin;
    string colourOfEyes;
    string genus; // Genus: Felinae, Pantherinae, Proailurinae
    virtual void run(int speed = 60) {
        cout << "Running. Should be overriden in childs" << endl;
    }
```

Изменения в дочерних классах

- По аналогии с классом **Tiger**, конструкторы класса **Cats** теперь вызывают конструктор родительского класса (класса **Mammal**), а затем происходит инициализация собственных полей класса **Cats**:

```
// Default constructor
Cats(int age=5, int size=20, bool isHungry=true, string colourOfSkin="red",
string colourOfEyes="white"):
        Mammal(age, size, isHungry, "cats"), colourOfSkin(colourOfSkin),
colourOfEyes(colourOfEyes)
{}

// Copy constructor
Cats(const Cats& c):
Mammal(c), genus(c.genus), colourOfSkin(c.colourOfSkin),
colourOfEyes(c.colourOfEyes)
{}
```