

## Лекция 0 (вводная): Знакомство с ООП на C++.

Вводная лекция строится по типу case study (исследование сразу на примере).

*Кто знаком с синтаксисом C-подобных языков, переходит сразу на следующую страницу.*

Вдаваться в подробности синтаксиса C++ мы пока не будем. Нам достаточно узнать, как написать простейшую программу на C++, как объявить переменную, хранящую целое число (предположим, что других типов данных и не нужно), как объявить функцию на C++ и как вывести строку и число на консоль.

Простейшая программа на C++ (она ничего не делает ☺):

```
int main()
{
    return 0;
}
```

Объявить целочисленную переменную на C++ можно так:

```
int main()
{
    int age = 20;
    return 0;
}
```

Рассмотрим пример простой функции на C++ (суммы 2 чисел) и вызов этой функции:

*// аналог этой записи на паскале - function sum(x,y: integer): integer*

```
int sum(int a, int b)
{
    int result = a + b;
    return result;           // можно сразу – return a + b;
}
```

```
int main()
{
    int x1 = 20;
    int x2 = 35;

    int s = sum(x1, x2);      // вызов функции sum;
```

```
std::cout << "Sum is " << s << std::endl;    // выведет: Sum is 55
```

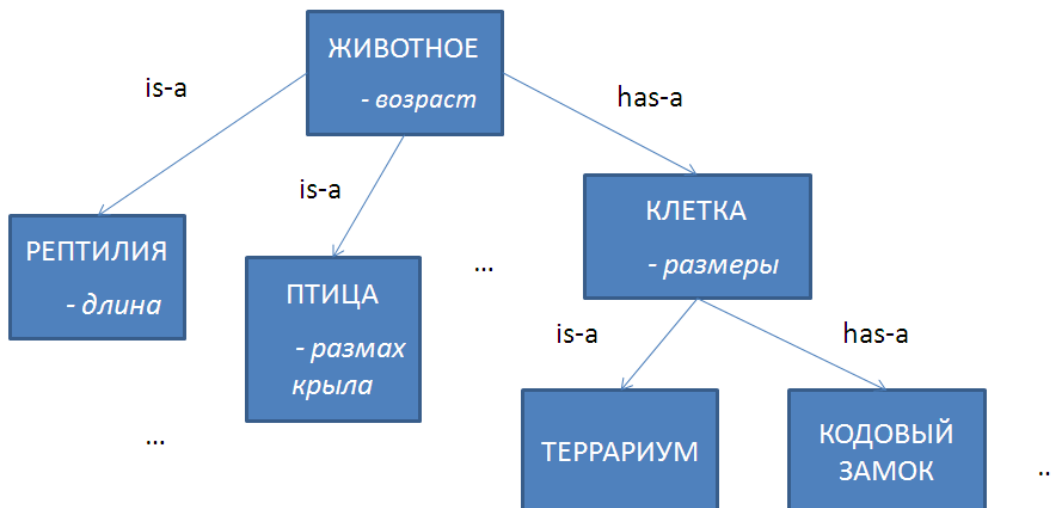
```
    return 0;  
}
```

Последняя строчка кода – это вывод на консоль строки и переменной *s*. Для того чтобы она работала, надо в самом начале текста программы подключить специальную библиотеку:

```
#include <iostream>
```

Вот, собственно, все и готово для того, чтобы заняться кейс стади.

Теперь пару слов об ООП. Данная парадигма начала развиваться 40 лет назад, вместе с фреймовой моделью Марвина Минского представления знаний. Рассмотрим эту модель на примере предметной области зоопарка (zoo domain):



Сразу отметим, что современное ООП – это не просто моделирование таксономии (объектов и связей между ними). В рамках ООП программисты кодируют еще абстракции, интерфейсы, контракты, динамику взаимодействия сущностей и многое другое. Но для понимания самых базовых элементов мы начнем, естественно, с тривиальности.

Фрейм, упрощенно говоря, представляет собой выражение некоторой сущности. Фрейм имеет слоты, в которых хранятся данные фрейма, и определенные функции по работе со своими данными. На рисунке слоты выделены курсивом; таких слотов может быть много. Фрейм *как сущность* (например, сущность «рептилия» как абстракция) в терминологии ООП вылился в понятие КЛАССА. Фрейм с заполненными слотами по конкретному

экземпляру сущности (например, конкретная рептилия – питон Иннокентий из клетки №18) вылился в понятие ООП «ОБЪЕКТ КЛАССА».

Основные связи между фреймами – это «имеет» (has-a) и «является (уточнением)» (is-a). Смысл этих связей из картинки интуитивно понятен: например, у животного ЕСТЬ клетка, а птица – это ЧАСТНЫЙ СЛУЧАЙ животного. Первый тип связи в ООП реализуется в виде **агрегации** или **композиции** объектов, второй тип – в виде **наследования** классов. Это все центральные понятия ООП.

Теперь просто рассмотрим, как можно закодировать на C++ часть приведенной выше модели.

Для этого создадим в MS Visual Studio новый проект типа Win32 Console Application. При создании в окне свойств поставим чекбокс «Empty project» (пустой проект). В окне Solution Explorer («обозреватель решений») увидим три пустые папки – Header Files, Resource Files и Source Files (их смысл рассмотрим на следующих лекциях). Добавим к папке «Source Files» файл main.cpp (создадим его – правой кнопкой Add New... и в категории CPP files введем main.cpp).

И напишем немного кода, помимо функции int main(). Мы добавим класс «Животное» с одним членом (свойством) – *возраст*, потому что это та характеристика, которая присуща любому животному, начиная от амебы и заканчивая соседом, который орет за стенкой в три часа ночи ☺. Также предположим, что животное может повзрослеть на год. Эта функция оформится в виде метода void growOlder() класса:

```
// ===== класс
животное, живое существо
class Animal
{
public:           // публичные (открытые) данные класса:
    int age_;    // у животного есть возраст (целое число)

    void growOlder() // и предположим, что любое животное мы можем заставить
повзрослеть
    {
        age_++;    // это C++-стайл аналог записи age = age + 1
    }
};
//
=====
=====

int main()           // а в главной функции...
{
    Animal a;        // создаем животное “а” (объект класса)
```

```

        a.age_ = 15;                // присвоим ему возраст 15 (можно, т.к. все данные
открыты)
        cout << a.age_ << endl;    // выведет на экран «15»

        a.growOlder();              // пусть повзрослеет
        std::cout << a.age_ << std::endl; // выведет на экран «16»

    return 0;
}

```

## Первый принцип ООП – ИНКАПСУЛЯЦИЯ.

*В широком смысле* означает, что внутреннее содержимое и подробности работы одного модуля не должны быть известны другим модулям, и какие-либо внутренние изменения в этом модуле не должны отражаться на других. *В узком смысле* заключается в том, что данные классов скрываются внутри самих классов, т.е. доступ к ним должен быть возможен только из самих классов (или из наследников). Для указания этого факта применяется **спецификатор доступа** `private` (или `protected`) вместо `public`. Но при этом становится вопрос – как работать с закрытыми данными? Во-первых, для работы с ними обычно создаются так называемые **ГЕТТЕРЫ** (получатели значений свойств) и **СЕТТЕРЫ** (установщики значений свойств). Во-вторых, начальные значения свойств задаются в **КОНСТРУКТОРАХ** классов – это специальные методы класса, которые неявно вызываются при создании класса. Таким образом, код нужно переписать так:

```

class Animal
{
public:
    Animal()                // конструктор; при создании животного просто обнулим
возраст
    {
        age_ = 0;
    }

    void setAge(int age)    // Сеттер: установить значение возраста
    {
        age_ = age;
    }

    int getAge()            // Геттер: узнать возраст
    {
        return age_;
    }

    void growOlder()
    {
        age_++;
    }

private:

```

```

    int age_;           // скрыли от посторонних доступ к возрасту
};

int main()
{
    Animal a;           // здесь неявно вызывается конструктор: age_ = 0
    a.age_ = 15;         // компилятор ругается – возраст то закрыт от посторонних!
    a.setAge(15);        // а вот так можно! Задать возраст через спец.метод - сеттер

    std::cout << a.getAge() << std::endl; // и узнать возраст можно! Через геттер

    return 0;
}

```

Обратите внимание, что теперь нельзя написать `a.age_` – компилятор выдаст ошибку, т.к. данное свойство класса теперь закрыто.

Второй принцип ООП – **НАСЛЕДОВАНИЕ**.

Наследование – это не что иное, как способ закодировать отношение типа is-a.

При наследовании класс-потомок имеет доступ ко всем данным класса-родителя (наследует их), кроме тех, которые объявлены `private`, следовательно, и у рептилии есть метод для взросления на 1 год. У рептилии был бы и возраст, но в прошлом примере мы присвоили ему уровень доступа `private`.

Впрочем, сеттеры и геттеры возраста достались в наследство рептилии.

Можно также открыть доступ к возрасту и для рептилии, при этом оставляя его закрытым для всех остальных – это делается через спецификатор `protected` (об этом также в следующих лекциях). В памяти объект класса-потомка строится «поверх» объекта класса-родителя.

Грамотно отнаследовать Рептилию от Животного можно, например, так:

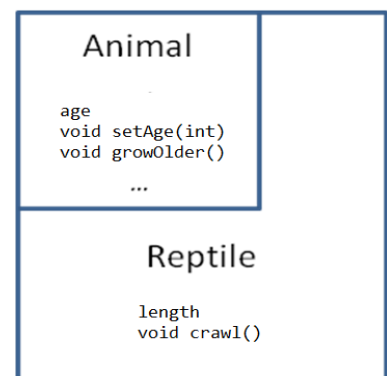
```

class Reptile: public Animal // наследование через :
{
public:
    Reptile()
    {
        length_ = 0;
    }

    void crawl()
    {
        length_++;
    }

    ...
private:
    int length_; // данные конкретно для рептилии
                // + родительские неявно достались
};

```



Для рассмотрения третьего главного принципа ООП – **ПОЛИМОРФИЗМА** – нам пока не хватает знаний, касающихся работы с указателями и динамической памятью в C++, поэтому оставим его на потом. Но суть его можно рассмотреть уже сейчас. Смысл полиморфизма заключается в том, что функционал (вызываемый метод) будет определяться во время выполнения программы, а не на этапе компиляции. Например, предположим, что рептилия имеет свою «версию взросления» и вырастет за раз не на 1 год, а на целых 2. Полиморфизм позволяет динамически определять, метод какого класса нужно будет вызвать в каждом конкретном случае. Если мы создадим целый массив животных, часть из которых будут рептилиями, и пройдемся по этому массиву, вызывая метод `growOlder`, то для рептилий будет вызываться своя версия метода, а для остальных животных – своя, хотя вызывать мы будем одну и ту же функцию в коде.

Наконец, **АГРЕГАЦИЯ/КОМПОЗИЦИЯ** позволяет реализовать связь типа `has-a`. Например, как связать животное и клетку? Ответ довольно прост и интуитивно понятен: включить ссылку (более точно - *указатель*) на класс «Клетка» в класс «Животное» (либо непосредственно сам объект класса «Клетка», а не ссылку; в данном случае это будет называться композиция, в первом случае – агрегация (более общий тип связи, при котором часть и целое могут существовать во времени независимо друг от друга)). Объявим класс «Клетка» и создадим объект Клетки в Животном:

```
class Cage
{
public:
    void setWidth(int width)
    {
        width_ = width;
    }

    int getWidth()
    {
        return width_;
    }
    ...
private:
    int width_;
    int height_;
};
```

```
class Animal
{
public:
    Animal()
    {
        age_ = 0;
    }
};
```

```

    }
    ...
private:
    int age_;
    Cage cage_;           // Композиция: в животном сразу и информация о клетке для
него
};

```

Отметим также, что в этой лекции мы о многом не сказали и кое-что упростили. В частности, объявлять разные классы в одном сpp-файле – дурной тон. По ходу всего курса мы эти пробелы закроем и неточности исправим.