



Отношение между классами:
обобщение,
реализация;
Статические методы и члены класса;
Модификатор `const`;

Отношения между классами

3. Обобщение – Наследование

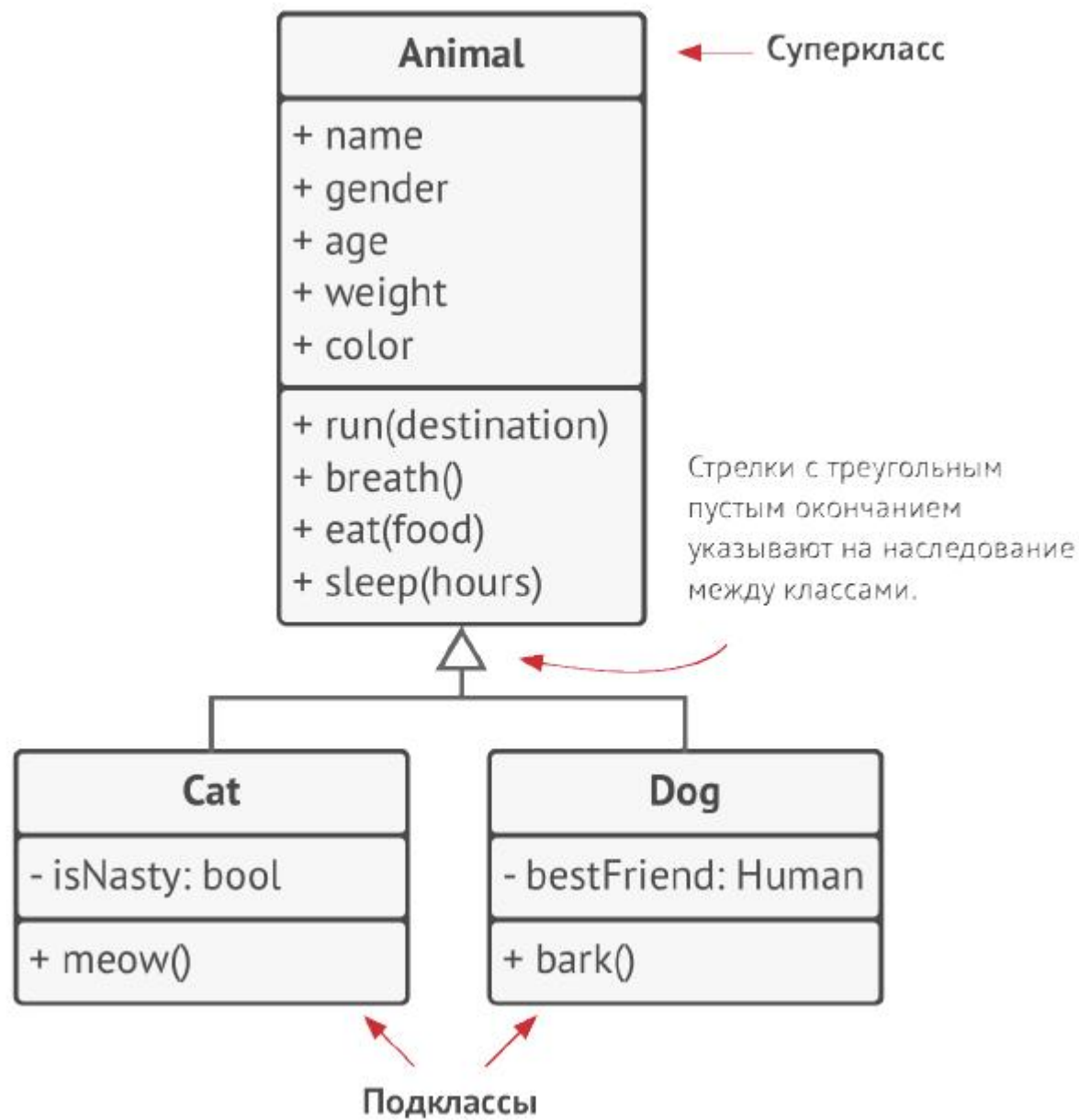
Наследование – это такое отношение между классами, когда один класс повторяет структуру и поведение другого класса (одиночное наследование) или других (множественное наследование) классов.

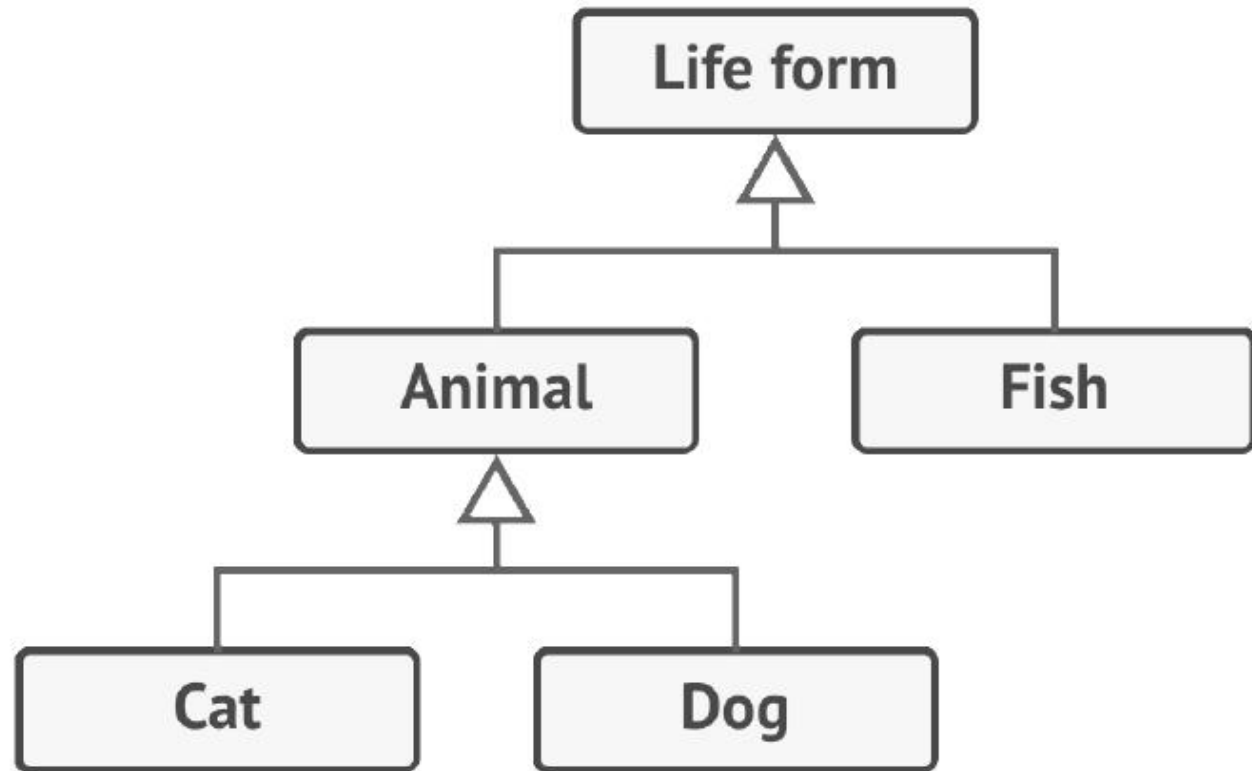
Понятие родительского и дочернего классов

Класс, структура и поведение которого наследуется, называется

суперклассом, надклассом, базовым или родительским классом

Класс, производный от суперкласса, называется *подклассом, производным* или *дочерним* классом





Наследование C++

Наследование – это такое отношение между классами, когда один класс повторяет структуру и поведение другого класса (одиночное наследование) или других (множественное наследование) классов.

Наследование позволяет *структурировать* и *повторно использовать код*, что *позволяет значительно ускорить процесс разработки*.

Но, тем не менее, наследование следует использовать с *осторожностью*, поскольку *большинство изменений в суперклассе затронут все подклассы*, что может привести к *непредвиденным последствиям*.

Наследование C++: пример

```
class Person
{
public:
    Person() { fio = "NoName"; age = 0; } //Конструктор по умолчанию
    Person(const std::string name, int a) //Конструктор с параметрами
    {
        fio = name; age = a;
    }
    void Display()
    {
        std::cout << "Name: " << fio << "\tAge: " << age << std::endl;
    }
protected:
    std::string fio="NoName"; //Инициализация по умолчанию доступна в
    стандарте C++11 и выше
private:
    int age = 0;
};
```

Наследование C++ : Типы наследования

В C++ есть несколько типов наследования:

1. **Публичный (public)** – публичные (public) и защищенные (protected) данные наследуются без изменения уровня доступа к ним;
2. **Защищенный (protected)** – все унаследованные данные становятся защищенными;
3. **Приватный (private)** – все унаследованные данные становятся **приватными**.

```
class Имя_класса : тип_наследования Класс //единочное
наследование
{
    //тело класса
};
```

```
class Имя_класса : тип_наследования Класс_1,
тип_наследования Класс_2 //множественное наследование
{
    //тело класса
};
```

Наследование C++ :пример

Рассмотрим класс работник (class Employee). Этот класс содержит все характеристики класса Персона(Person), к которым нужно добавить информацию о компании, в которой работает сотрудник.

Можно заново описать класс Employee, с включением всех характеристик либо использовать то, что было разработано ранее. Например выполнить наследование, тем самым повторить структуру класса **Person**.

Например.

```
class Employee : public Person
{
    protected:
    std::string company = "NoCompany";
};

void main()
{
    Person p0("Alice Goodwin",8);
    Employee p1;
    p0.Display();
    p1.Display();
}
```


Наследование C++ :Конструкторы и деструкторы

В C ++ конструкторы и деструкторы не наследуются.

Однако они вызываются, когда дочерний класс инициализирует свой объект.

Конструкторы вызываются **один за другим иерархически**, начиная с базового класса и заканчивая последним производным классом.

Деструкторы вызываются в обратном порядке.

Наследование C++ :пример

Добавим конструктор с параметрами.

```
class Employee : public Person
{
public:
    Employee() { company = "NoCompany"; }
    Employee(std::string inf, int val, std::string cmp):Person(inf,val){
        company = cmp;
    }
    void Display() {
        Person::Display();
        std::cout << "Company Name: " << company<< std::endl;
    }
protected:
    std::string company = "NoCompany";
};
```

Наследование C++ :Деструкторы

```
class Person
{
public:
~Person() { cout << "Person destructor called" << endl; }
};

class Employee : public Person
{
public:
~Employee() { cout << "Employee destructor called" << endl; }
};

int main()
{
    Person p0("Alice Goodwin",8);
    Employee p1("Bob Goodwin", 35, "Microsoft");
}
```

```
Employee destructor called
Person destructor called
Person destructor called
```

Наследование C++ : запрет наследования

Иногда наследование от класса может быть нежелательно. И с помощью спецификатора **final** мы можем запретить наследование(C++11,...):

```
class Employee final : public Person
{

};

class X : public Employee //!!!!!!
{

};
```

Наследование C++ :Деструкторы

```
int main()
{
    Person *p0 = new Employee («Alice Goodwin", 8, "Microsoft");
    p0->Display();
    delete p0;
}
```

Microsoft Visual Studio Debug Console

```
Name: Alice Goodwin    Age: 8
Person destructor called
```

Наследование C++ :Деструкторы

```
class Person
{
virtual void Display()
{
    std::cout << "Name: " << fio << "\tAge: " << age << std::endl;
}
};
```

```
int main()
{
    Person *p0 = new Employee («Alice Goodwin", 8, "Microsoft");
    p0->Display();
    delete p0;
}
```

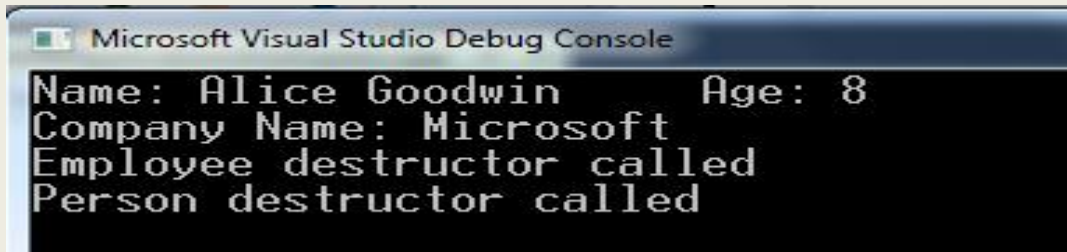
Microsoft Visual Studio Debug Console

```
Name: Alice Goodwin      Age: 8
Company Name: Microsoft
Person destructor called
```

Наследование C++ :Деструкторы

```
class Person
{
    virtual void Display()
    {
        std::cout << "Name: " << fio << "\tAge: " << age << std::endl;
    }
    virtual ~Person() { cout << "Person constructor called" << endl; }
};
```

```
int main()
{
    Person *p0 = new Employee («Alice Goodwin", 8, "Microsoft");
    p0->Display();
    delete p0;
}
```



The screenshot shows the Microsoft Visual Studio Debug Console with the following output:

```
Name: Alice Goodwin      Age: 8
Company Name: Microsoft
Employee destructor called
Person destructor called
```

Наследование C++ :Деструкторы

```
class Person
{
    virtual void Display()
    {
        std::cout << "Name: " << fio << "\tAge: " << age << std::endl;
    }
    virtual ~Person() { cout << "Person constructor called" << endl; }
    virtual void DoSomething() = 0;
};
```

```
int main()
{
    Person *p0 = new Employee («Alice Goodwin", 8, "Microsoft");
    p0->Display();
    delete p0;
}
```

Пример реализации можно посмотреть по ссылке

<https://onlinegdb.com/HFJsk6p6Q>

Наследование C++ :Деструкторы

Когда же следует объявлять деструктор **виртуальным**?

Существует правило - если *базовый класс* предназначен для полиморфного использования, то его деструктор должен объявляться виртуальным.

Для реализации механизма виртуальных функций каждый объект класса хранит указатель на таблицу виртуальных функций **vptr**, что увеличивает его общий размер.

Обычно, при объявлении виртуального деструктора такой класс уже имеет виртуальные функции, и увеличения размера соответствующего объекта не происходит.

Если же базовый класс не предназначен для полиморфного использования (не содержит виртуальных функций), то его деструктор не должен объявляться виртуальным.

Отношения между классами

Реализация – это семантическая связь между классами, когда один из них (**поставщик**) определяет соглашение, которого второй (**клиент**) обязан придерживаться.

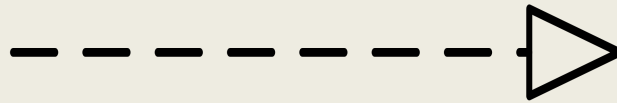
Это связи между *интерфейсами* и *классами*, которые реализуют эти интерфейсы.

Поставщик, как правило, представлен *абстрактным классом*.

В графическом исполнении связь реализации – это гибрид связей *обобщения и зависимости*:

треугольник указывает на *поставщика*, а второй конец пунктирной линии – на клиента.

Реализация – это семантическая связь между классами, когда один из них (поставщик) определяет соглашение, которого второй (клиент) обязан придерживаться.



АБСТРАКТНЫЙ КЛАСС

Абстрактный класс в C++ - это класс, в котором объявлена хотя бы одна *чисто виртуальная функция*.

Абстрактный класс используют, когда необходимо создать семейство классов (*например много разновидностей объекта*), при этом было бы лучше **вынести общую реализацию и поведение** в отдельный класс.

Таким образом *переопределить/дописать* придется только специфичные для каждого класса методы (*у каждого разновидности объекта свое поведение*) и/или расширить функциональность класса.

ИНТЕРФЕЙС

C++ нет понятия интерфейс на уровне языка.

Поведение интерфейса симулируется через поведение абстрактного класса наследуя его.

Интерфейс - абстрактный класс

Отличия между интерфейсом и абстрактным классом

1. Каждый интерфейс является абстрактным классом, не каждый абстрактный класс – интерфейс.
2. Интерфейс содержит только **public** секцию, абстрактный класс не имеет ограничений.
3. Интерфейс содержит только **pure virtual methods**, абстрактный класс может содержать и поля, и не виртуальные методы.

Статические методы и члены класса

Статический метод или член класса означает, что он один и тот же на все экземпляры (объекты) класса.

Обычно статический метод в классе используется, когда необходимо реализовать некоторое действие, которое относится к классу в целом, а не к конкретному объекту класса.

Статический член класса - это свойство, относящееся к классу в целом, а не конкретному объекту.

Особенности

- Статический член класса инициализируется вне класса, для того чтобы была выделенная память компилятором в глобальном участке.
- **static** метод класса не имеет доступа к нестатическим членам класса, так как у него нет указателя **this**. Для организации работы с объектом, можно передать указатель на экземпляр класса.
- Статический член класса /метод класса можно использовать и без создания экземпляра класса

```
class Something
{
public:
    static int some_value;
};
int Something::some_value = 3;

// Something::some_value = 121;
```

```
class SomeObject
{
    public:
        SomeObject(){ count++;}

        ~SomeObject(){ count--;}
        static unsigned getCount();

    private:
        static unsigned count;
};

unsigned SomeObject::count = 0;

unsigned SomeObject::getCount()
{
    return count;
}
```



```
int main()
{
std::cout<<"Count- "<<SomeObject::getCount()<<endl;
SomeObject A, B, C;
std::cout << "Count - " << SomeObject::getCount()<<endl;
if (1)
{
SomeObject A, B, C;
std::cout << "Count - " << SomeObject::getCount() << endl;
}

std::cout << "Count - " << SomeObject::getCount()<<endl;

}
```

МОДИФИКАТОР CONST



МОДИФИКАТОР CONST

```
const int i(1);
```

```
int const j(1);
```

```
int const k=1;
```

```
k = 7; // <-- ошибка на этапе компиляции!
```

МОДИФИКАТОР CONST

При использовании **const** с указателями, действие модификатора распространяется либо на **значение указателя**, либо **на данные** на которые указывает указатель.

ИСПОЛЬЗОВАНИЕ CONST С УКАЗАТЕЛЯМИ

- 1) `int * const p1`
- 2) `int const* p2`
- 3) `const int* p3`
- 4) `const int* const p4`

1) `p1` константа. Тип, на который `p1` указывает, это `int`.

Значит получился константный указатель на `int`. Его можно инициализировать лишь однажды и больше менять нельзя.

```
int q=1;  
int *const p1 = &q; //инициализация в момент  
объявления  
*p1 = 5; //само число можно менять
```

это по разному записанное одно и то же объявление. Указатель на целое, которое нельзя менять.

ИСПОЛЬЗОВАНИЕ CONST С УКАЗАТЕЛЯМИ

- 1) `int *const p1`
- 2) `int const* p2`
- 3) `const int* p3`

2)3) Это по разному записанное одно и то же объявление. Указатель на целое, которое нельзя менять.

```
int q=1;  
const int *p;  
p = &q; //на что указывает p можно менять  
*p = 5; //ошибка, число менять уже нельзя
```

Как правило, используется вариант объявления `const int`, а `int const` используется, чтобы запутать на собеседовании.

ИСПОЛЬЗОВАНИЕ CONST С УКАЗАТЕЛЯМИ

`const` можно использовать со ссылками, чтобы через ссылку нельзя было поменять значение переменной.

```
int p = 4;  
const int& x=p; //нельзя через x поменять значение p  
x=5; //ошибка
```

Константная ссылка - это нонсенс.

Она по определению константная. Компилятор скорее всего выдаст предупреждение, что он проигнорировал `const`.

```
int& const x; //не имеет смысла//ошибка
```

ПЕРЕДАЧА ПАРАМЕТРОВ В ФУНКЦИЮ

`const` удобен, если нужно передать параметры в функцию, но при этом надо обязательно знать, что переданный параметр не будет изменен.

- 1) `void f1(const std::string& s);`
- 2) `void f2(const std::string* sptr);`
- 3) `void f3(std::string s);`

В первой и второй функции попытки изменить строку будут пойманы на этапе компиляции.

В третьем случае в функции будет происходить работа с локальной копией строки, исходная строка не пострадает.

CONST ДАННЫЕ В КЛАССЕ

Значения `const` данных класса задаются один раз и навсегда в конструкторе.

```
class CFoo
{  const int num;
public:
    CFoo(int anum);
};

CFoo::CFoo(int anum):num(anum)
{
    ...
}
```

Значения `static const` данных класса типа (`enum`, `int`, `char`) прямо в объявлении класса.

```
class CFoo
{
public:
    static const int num=50;
};
```

CONST ФУНКЦИИ В КЛАССЕ

Функция класса, объявленная `const`, трактует `this` как указатель на константу.

Т.е `this` в методе класса `X` будет `X*`.

Но если метод класса объявлена как `const`, то тип `this` будет `const X*`.

В таких методах не может быть ничего присвоено переменным класса, которые не объявлены как `static` или как `mutable`

Также `const`-функции не могут возвращать не `const` ссылки и указатели на данные класса и не могут вызывать не `const` функции класса.

`const`-функции иногда называют *инспекторами* (inspector), а остальные *мутаторами* (mutator).

CONST ФУНКЦИИ В КЛАССЕ

```
class CFoo
{
public:
    int inspect() const; // Эта функция обещает не менять
*this
    int mutate(); // Эта функция может менять *this
};
```

В классе могут присутствовать две функции отличающиеся только const.

```
class CFoo
{
    ...
public:

    int func () const;
    int func ();
};
```

```
class A {  
  
    private:  
  
    int x;  
  
    public:  
        void f(int a) const {  
            x = a; // <-- не работает  
        }  
};
```

```
class A {  
    int x;  
public:  
  
    A(int a) {  
        x = a;  
        cout << "A(int) // x=" << x << endl;  
    }  
  
    void f() {  
        cout << "f() // x=" << x << endl;  
    }  
  
    void f() const {  
        cout << "f() const // x=" << x << endl;  
    }  
};
```

```
int main() {  
    A a1(1);  
    a1.f();  
    A const a2(2);  
    a2.f();  
    return 0;}
```

Результат:

A(int) // x=1

f() // x=1

A(int) // x=2

f() const // x=2

То есть для константного объекта (с x=2) был вызван соответствующий метод.

Если планируется использовать **const-объекты**, то необходимо реализовать **const-методы**.

Если вы в этом случае не реализуете **не-const-методы**, то во всех случаях будут по умолчанию использоваться **const-методы**.

CONST ФУНКЦИИ В КЛАССЕ

Не всякая функция может быть объявлена константной.

Конструкторы и **деструкторы** не могут быть объявлены как `const`.

Также не бывает **`static const`** функций.

```
class CFoo
{
    int i;
    public:

    static int func () const;    //ошибка
};
```


Рассмотрим объявление

```
const CFoo p;
```

Экземпляр класса **CFoo**, объявленный таким образом, что обещает сохранить физическое состояние класса, не менять его.

Как следствие, он не может вызвать не **const** функции класса **CFoo**.

Все данные, не объявленные как **const**, начинают трактоваться как **const**.

То есть

```
int становится int const
```

```
int * становится int * const
```

```
const int * становится int const *const
```

и так далее.