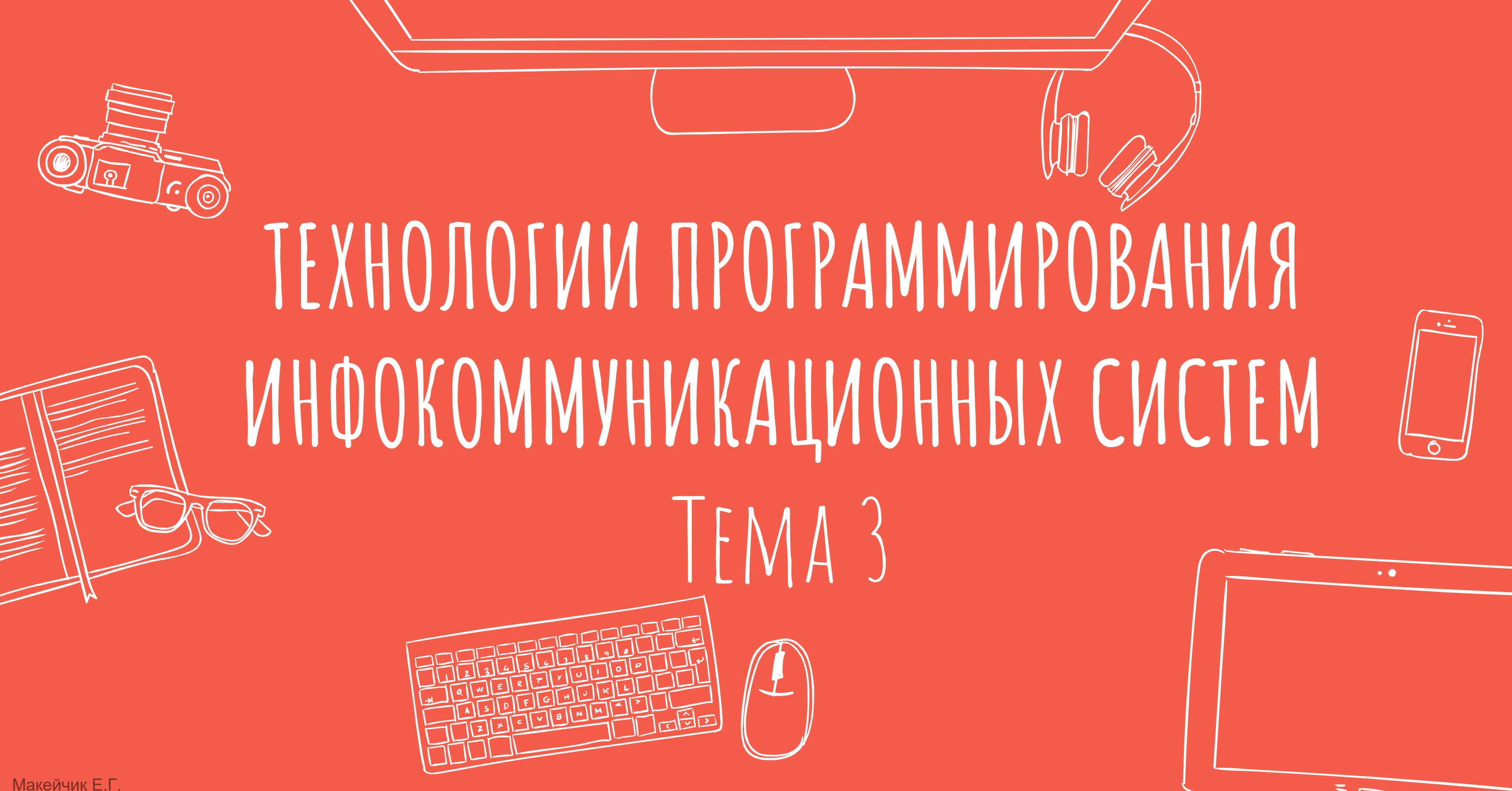
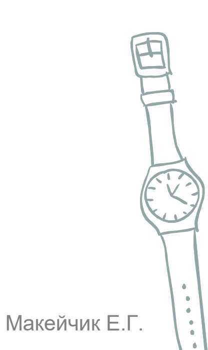


Макейчик Е.Г.

Наследование

Наследование – механизм создания новых производных классов (классов-потомков) из уже имеющихся базовых классов (классов-родителей) посредством добавления собственных полей и методов (свойств).



Наследование

Базовый класс

Свойство А | Свойство Б | Свойство В

Наследование

Свойство Г | Свойство Д

Свойство А | Свойство Б | Свойство В

Производный класс

Animals

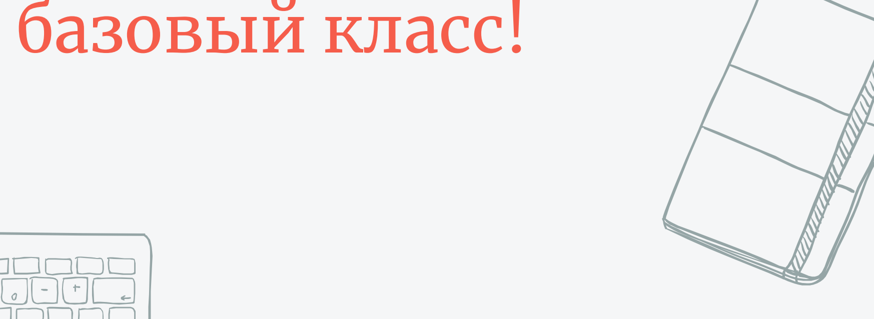
Amphibians | Reptiles | Mammals | Birds

Наследование

Производный класс можно усовершенствовать, добавляя новые:

- поля
- методы
- конструкторы.

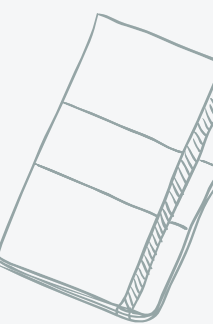
И все это не изменяя базовый класс!



Наследование

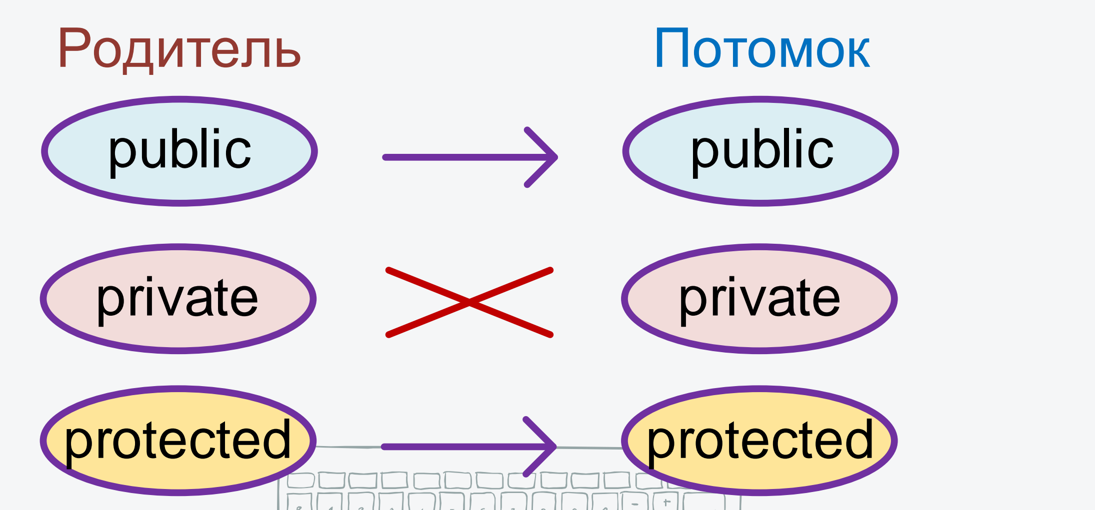
Потомок

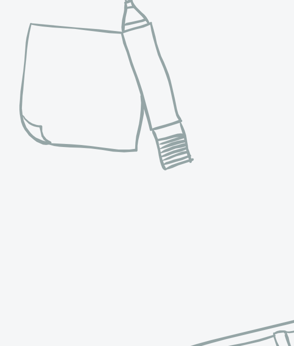
поля, методы, конструкторы	→	поля, методы родителя + свои

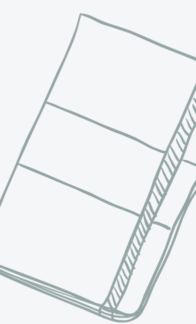


Модификатор доступа protected

protected – это модификатор доступа, который работает как private, но распространяется также на свойства потомка



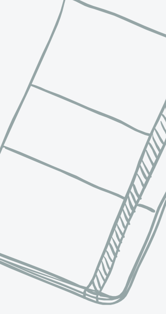


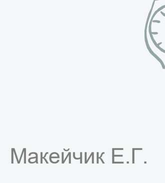


ПРИМЕРЫ

Если бы поле zebras находилась в доступе private, то использование его в методе counter_zebras() привело бы к ошибке.

```
class Animals
{
protected:
    int zebras;
};

class Dog : public Animals
{
    int counter_zebras()
    {
        return zebras;
    }
};
```





Наследование (права доступа)

Чтобы наследовать класс нужно использовать конструкцию

```
class Имя_потомка : модификатор наследования Имя_родительского_класса {/* тело класса */};
```

- Первое на что надо обратить внимание это на двоеточие (:) оно одинарное, а не двойное как у области видимости.
- Второе это <модификатор наследование>. При его оперировании можно задать какими модификаторами доступа родительского класса можно будет пользоваться в дочернем.

Из трех почти всегда используется public

Наследование (права доступа)

public – использовать можно public и protected родительского класса

```
class Animals
{
public:
    int counter; // Общие количество животных
protected:
    int zebras;
    int bears;
    int dogs;
    int count_animals() // Метод, вычисляющий общее количество животных
    {
        counter = dogs + bears + zebras;
    }
    int set_dogs(int counter_of_dogs)
    {
        dogs = counter_of_dogs;
    }
};

class Dog : public Animals
{
public:
    int counter_dogs()
    {
        return dogs; // Использование поля dogs
    }
};
```

Объявили метод public: count_dogs(), который возвращает поле protected: dogs из Animals.

Наследование (права доступа)

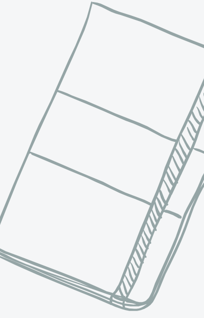
private – пользоваться можно лишь полями (не методами) родителя. Чтобы использовать методы нужно разрешить это напрямую (и без круглых скобок, только имя), а также разрешать нужно в публичном доступе (public)

```
Имя_родительского_класса :: свойства;
```

Наследование (права доступа)

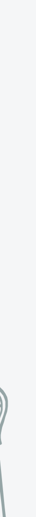
```
class Dog : public Animals
{
public:
    int counter_dogs()
    {
        return dogs; // Использование поля dogs
    }
    Animals::set_dogs;
};

int main()
{
    Dog Pilot;
    int k;
    cout << "Введите количество собак: "; cin >> k;
    Pilot.set_dogs(k);
    cout << "Количество собак ровняется: " << Pilot.counter_dogs();
    return 0;
}
```

Получили доступ к методу set_dogs().



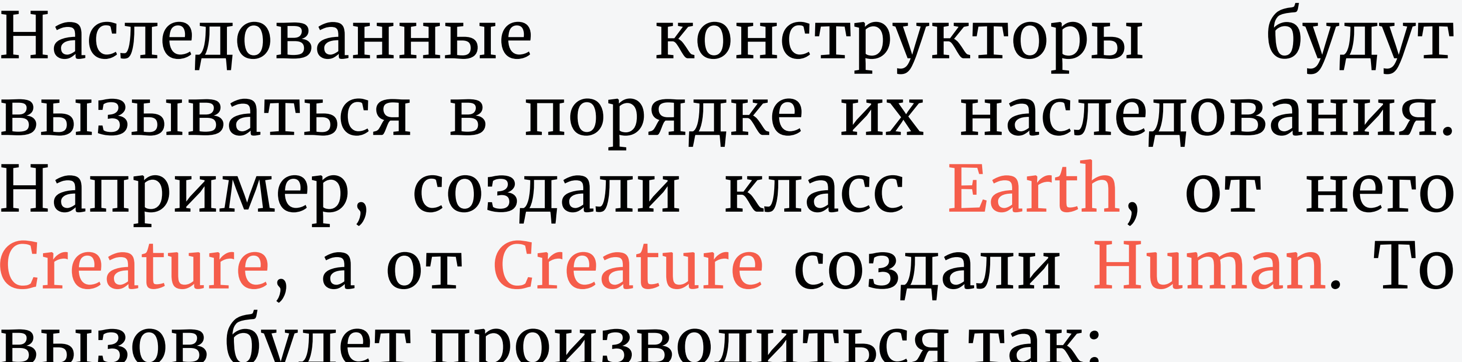
Наследование (права доступа)

protected – идентичен private, но свойства public переходит в доступ protected



Вид наследования	Объявление поля в базовом классе	Видимость в производном классе
private	private protected public	не видимо private private
protected	private protected public	не видимо protected protected
public	private protected public	не видимо protected public

Наследование конструктора



Наследованные конструкторы будут вызываться в порядке их наследования. Например, создали класс Earth, от него Creature, а от Creature создали Human. То вызов будет производиться так:

- Earth()
- Creature()
- Human()

Наследование конструктора

Для наследования конструктора нужно использовать следующую конструкцию:

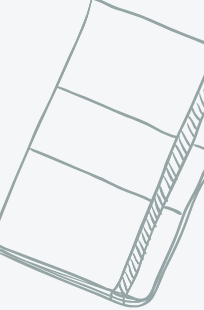
```
Имя_потомка(аргументы конструктора) :
    Имя_родительского_класса(аргументы конструктора) { /* тело */ };
```

Наследование конструктора

В начале указываем имя дочернего класса – Имя_потомка.
Далее аргументы конструктора указываем столько имен переменных сколько требует этого родительский конструктор, дальше передаем базовому конструктору в скобках (аргументы конструктора) объявленные переменные.
тело – это тело конструктора.
Но чтобы все это работало, в конструкторе базового класса к переменным нужно обращаться через this->.

НАСЛЕДОВАНИЕ КОНСТРУКТОРА

```
class Animals
{
public:
    Animals() {
        cout << "Конструктор";
    }
    Animals(int counter) {
        this->counter_of_animals = counter;
    }
protected:
    int counter;
    int counter_of_animals;
};

class Dog : public Animals
{
public:
    Dog() : Animals() {} // Перезгрузка конструкторов
    Dog(int couter) :
        Animals(counter) {}

    int get_count_animals()
    {
        return counter_of_animals;
    }

};

int main()
{
    setlocale(LC_ALL, "Russian");
    Dog Pilot;
    Dog Givik;
}
```

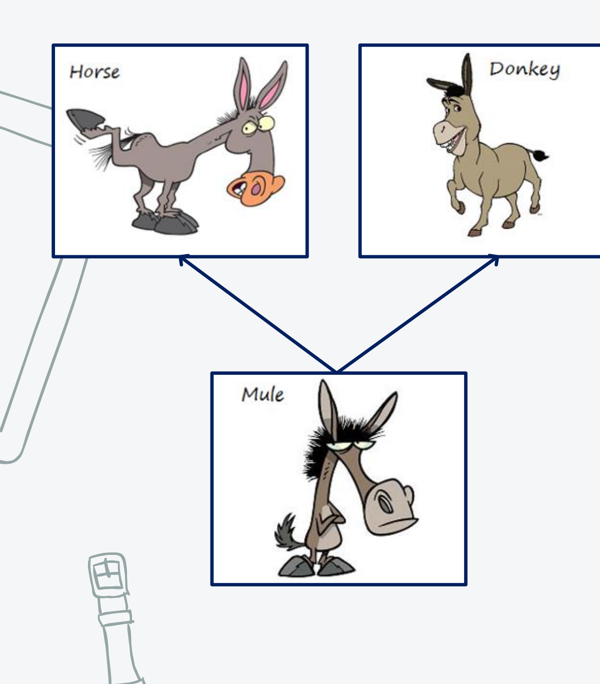


Наследование деструктора

Наследованные деструкторы вызываются наоборот по сравнению с вызыванием конструктора. Если классы созданные так Earth -> Creature -> Human, то цепочка вызовов деструктора имеет такой вид:

- ~Human()
- ~Creature()
- ~Earth()

Простое и множественное наследование



Наследование называется **простым**, если производный класс порождается из одного базового класса и **множественным**, если производный класс обладает свойствами двух и более родительских классов.

Простое и множественное наследование

Простое

```
class A{
    int    m_i;
    float m_f;
};

class B: A{
    double m_d;
};

class C: B{
    long m_l;
};
```

A: int m_i; float m_f;

B: int m_i; float m_f; double m_d;

C: int m_i; float m_f; double m_d; long m_l;

Множественное

```
class A{
    int m_i;
};

class B {
    double m_d;
};

class C: A, B{
    float m_f;
};
```

A: int m_i;

B: double m_d;

C: int m_i; double m_d; float m_f;

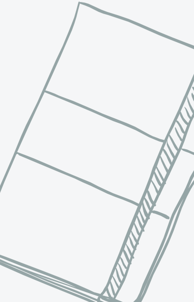
Виртуальные функции

Виртуальная функция – это функция, объявленная с ключевым словом virtual в базовом классе и переопределенная в одном или в нескольких производных классах.

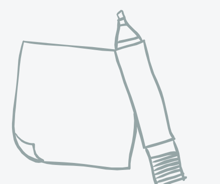
- Виртуальные функции объявляются в производных классах с тем же именем, возвращаемым значением и типом аргументов.
- Механизм виртуальных функций реализует полиморфизм времени выполнения: какой виртуальный метод вызовется будет известно только во время выполнения программы.

Виртуальные функции

- Для каждого класса, содержащего виртуальные методы создается таблица виртуальных функций. Эта таблица предназначена для вызова нужных реализаций виртуальных методов во время исполнения программы.
- При вызове объекта производного класса C++ определяет во время исполнения программы, какую функцию вызвать, основываясь на типе объекта. Для разных объектов вызываются разные версии одной и той же виртуальной функции.

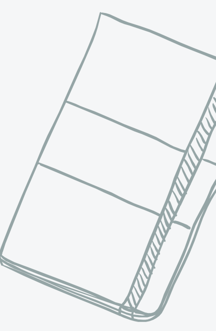


Виртуальные функции



Идите все работать!

```
class Manager {
.
.
.
}
class Developer {
.
.
.
}

class Disigner {
.
.
.
}
```



Виртуальные функции (пример)


```cpp
#include <iostream>
using namespace std;

class Emploee {
public:
    void goToWork() {
        cout << "Работник пошел работать" << endl;
    }
};
class Manager : public Emploee
{
public:
    void goToWork() {
        cout << "Менеджер пошел работать" << endl;
    }
};
class Developer : public Emploee
{
public:
    void goToWork() {
        cout << "Разработчик пошел работать" << endl;
    }
};
class Disigner : public Emploee
{
public:
    void goToWork() {
        cout << "Дизайнер пошел работать" << endl;
    }
};
```

```cpp
int main()
{
    setlocale(LC_ALL, "Russian");
    Emploee* empl;
    Manager* man = new Manager;
    Developer* dev = new Developer;
    Disigner* dis = new Disigner;
    empl = man;
    empl->goToWork();
    empl = dev;
    empl->goToWork();
    empl = dis;
    empl->goToWork();
    return 0;
}
```

```
Работник пошел работать
Работник пошел работать
Работник пошел работать
```

Виртуальные функции (пример)

```cpp
#include <iostream>
using namespace std;

class Emploee {
public:
    virtual void goToWork() {
        cout << "Работник пошел работать" << endl;
    }
};
class Manager : public Emploee
{
public:
    void goToWork() {
        cout << "Менеджер пошел работать" << endl;
    }
};
class Developer : public Emploee
{
public:
    void goToWork() {
        cout << "Разработчик пошел работать" << endl;
    }
};
class Disigner : public Emploee
{
public:
    void goToWork() {
        cout << "Дизайнер пошел работать" << endl;
    }
};
```

```cpp
int main()
{
    setlocale(LC_ALL, "Russian");
    Emploee* empl;
    Manager* man = new Manager;
    Developer* dev = new Developer;
    Disigner* dis = new Disigner;
    empl = man;
    empl->goToWork();
    empl = dev;
    empl->goToWork();
    empl = dis;
    empl->goToWork();
    return 0;
}
```

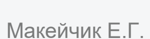
```
Менеджер пошел работать
Разработчик пошел работать
Дизайнер пошел работать
```

Виртуальные функции (пример)

```cpp
#include <iostream>
using namespace std;

class Emploee {
public:
    virtual void goToWork() {
        cout << "Работник пошел работать" << endl;
    }
};
class Manager : public Emploee
{
public:
    void goToWork() {
        cout << "Менеджер пошел работать" << endl;
    }
};
class Developer : public Emploee
{
public:
    void goToWork() {
        cout << "Разработчик пошел работать" << endl;
    }
};
class Disigner : public Emploee
{
public:
    void goToWork() {
        cout << "Дизайнер пошел работать" << endl;
    }
};
```

```cpp
int main()
{
    setlocale(LC_ALL, "Russian");
    Emploee* empl;
    Manager* man = new Manager;
    Developer* dev = new Developer;
    Disigner* dis = new Disigner;
    empl = man;
    //empl->goToWork();
    empl = dev;
    //empl->goToWork();
    empl = dis;
    //empl->goToWork();
    Emploee* mas[3];
    mas[0] = man;
    mas[1] = dev;
    mas[2] = dis;
    for (int i = 0; i < 3; i++)
        mas[i]->goToWork();
    return 0;
}
```

```
Менеджер пошел работать
Разработчик пошел работать
Дизайнер пошел работать
```

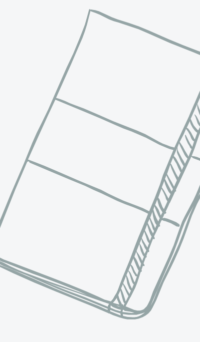


Абстрактные функции и классы

Абстрактной виртуальной называется функция, объявленная в базовом классе как виртуальная, но не содержащая описания выполняемых действий. Еще она называется чисто виртуальной функцией.

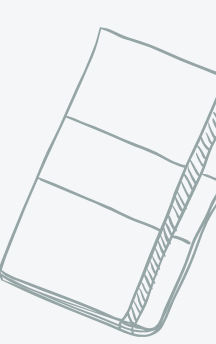
Форма объявления абстрактной виртуальной функции:

```
virtual тип имя_функции(аргументы) = 0;
```



Абстрактные функции и классы

- Класс с одной или большим количеством чисто виртуальных функций называется – абстрактным.
- Создание объектов абстрактного класса запрещено. Они могут использоваться только как базовые для создания других классов.
- Если класс, производный от абстрактного, не содержит переопределения виртуальной функции, то он также является абстрактным.



Абстрактные классы (пример)


```cpp
#include <iostream>
using namespace std;

class Emploee {
public:
    virtual void goToWork() = 0;
};
class Manager : public Emploee
{
public:
    void goToWork() {
        cout << "Менеджер пошел работать" << endl;
    }
};
class Developer : public Emploee
{
public:
    void goToWork() {
        cout << "Разработчик пошел работать" << endl;
    }
};
class Disigner : public Emploee
{
public:
    void goToWork() {
        cout << "Дизайнер пошел работать" << endl;
    }
};
```

```cpp
int main()
{
    setlocale(LC_ALL, "Russian");

    Manager man;
    Developer dev;
    Disigner dis;
    man.goToWork();
    dev.goToWork();
    dis.goToWork();
    return 0;
}
```

```
Менеджер пошел работать
Разработчик пошел работать
Дизайнер пошел работать
```

Типы связей между объектами

Наша жизнь полна повторяющихся шаблонов, отношений и иерархий между объектами. Изучая их, мы получаем более глубокое представление о том, как они работают и взаимодействуют между собой в реальной жизни.

Типы связей между объектами

Когда речь заходит об объектах в программировании, то те же шаблоны, которыми мы руководствуемся по отношению к объектам в реальной жизни, применимы и к объектам в программировании, которые мы создаем сами. Изучая эти отношения, повторяющиеся шаблоны и иерархии подробнее, мы можем понять, как улучшить код для его повторного использования в других программах и как писать классы, функционал которых можно будет легко расширить.

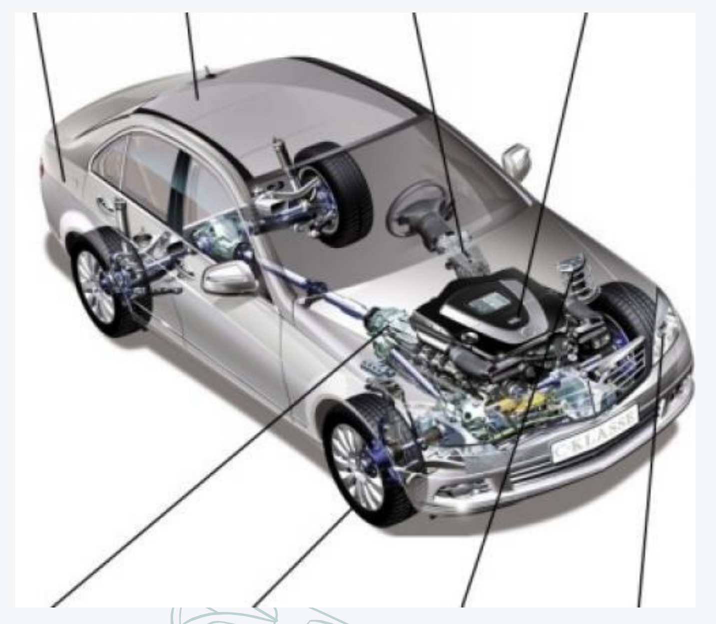
Связи между объектами в программировании

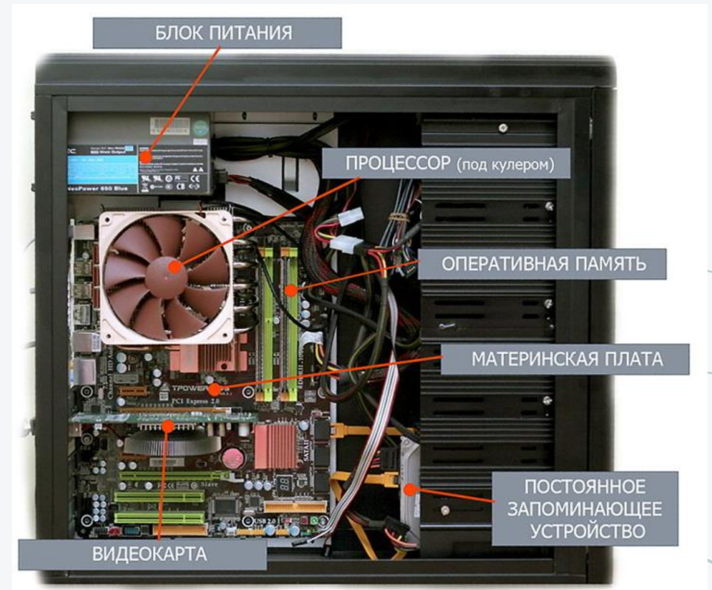
Существует много разных типов отношений, которые два объекта могут иметь в реальной жизни, и нужно использовать определенные слова (типы отношений) для их описания, например:

- Квадрат «является» геометрической фигурой.
- Автомобиль «имеет» руль.
- Программист «использует» клавиатуру.
- Цветок «зависит от» растения.
- Ученик является «членом» класса.
- Наш мозг существует как «часть» нас самих.

Композиция объектов

Процесс построения сложных объектов из более простых называется композицией объекта.

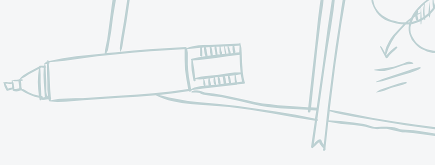




Композиция объектов

В композиции между двумя объектами представлен тип отношения «имеет». Автомобиль «имеет» коробку передач. Компьютер «имеет» центральный процессор. Вы «имеете» сердце. Сложный объект иногда называют целым (или «родителем»). Более простой объект часто называют частью (или «дочерним элементом», «компонентом»).

Композиция объектов полезна в контексте языка C++, поскольку позволяет создавать сложные классы, объединяя более простые и легко управляемые части. Это уменьшает сложность и позволяет писать код быстрее и с меньшим количеством ошибок, так как можно повторно использовать код, который уже был написан, протестирован, и является рабочим. Существует два основных подтипа композиции объекта: композиция и агрегация.



Композиция

Для реализации композиции объект и часть должны иметь следующие отношения:

- Часть (член) является частью объекта (класса).
- Часть (член) может принадлежать только одному объекту (классу) в моменте.
- Часть (член) существует, управляемая объектом (классом).
- Часть (член) не знает о существовании объекта (класса).

Композиция

Сердце является частью тела человека. Часть в композиции может быть частью только одного объекта в моменте. Сердце, которое является частью тела одного человека, не может быть одновременно частью тела еще одного человека.

В отношениях внутри композиции объект несет ответственность за существование частей. Чаще всего это означает, что часть создается при создании объекта и уничтожается при его уничтожении.

Объект управляет временем жизни части таким образом, что пользователь, который использует объект, не должен участвовать в этом. Например, при создании тела создается и сердце. Когда тело человека уничтожается, то и его сердце уничтожается тоже. Часть не знает о существовании целого. Сердце работает круглосуточно, не зная, что оно является частью более крупной организации. Это называется однонаправленным отношением, поскольку тело знает о сердце, а сердце о теле – нет.

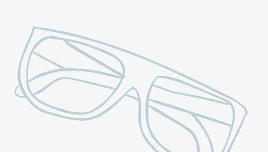
Для реализации агрегации целое и его части должны соответствовать следующим отношениям:

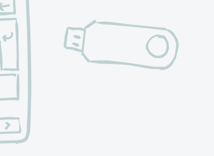
- Часть (член) является частью целого (класса).
- Часть (член) может принадлежать более чем одному целому (классу) в моменте.
- Часть (член) существует, не управляемая целым (классом).
- Часть (член) не знает о существовании целого (класса).

В отличие от композиции, части могут принадлежать более чем одному целому в моменте, и целое не управляет существованием и продолжительностью жизни частей. При создании/уничтожении агрегации, целое не несет ответственности за создание/уничтожение своих частей.

Агрегация

У каждого человека есть свой адрес. Однако этот адрес может принадлежать более чем одному человеку в моменте, например, вам и вашему соседу по комнате или родственникам, которые живут вместе с вами. К тому же этот адрес не управляется человеком – адрес существовал до того, как человек заселился и будет существовать после того, как человек выселится. Кроме того, человек знает, по какому адресу он живет, но адрес, в свою очередь, не знает, что это за человек и вообще, сколько их там находится. Такие отношения и являются агрегацией.





Агрегация

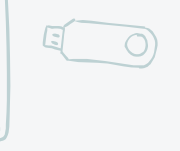
Двигатель является частью автомобиля. И хотя двигатель принадлежит автомобилю, он может принадлежать и другим объектам, например, человеку, которому принадлежит автомобиль. Автомобиль не несет ответственности за создание или уничтожение двигателя. И в то же время автомобиль знает, что у него есть двигатель (ведь благодаря ему он двигается), но сам двигатель не знает, что он является частью автомобиля.

В композиции:

- ✗ Используются обычные переменные-члены.
- ✗ Используются указатели, если класс реализовывает собственное управление памятью (происходит динамическое выделение/освобождение памяти).
- ✗ Класс ответственный за создание/уничтожение своих частей.

В агрегации:

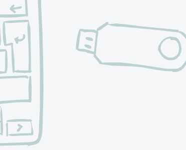
- ✗ Используются указатели/ссылки, которые указывают/ссылаются на части вне класса.
- ✗ Класс не несет ответственности за создание/уничтожение своих частей.



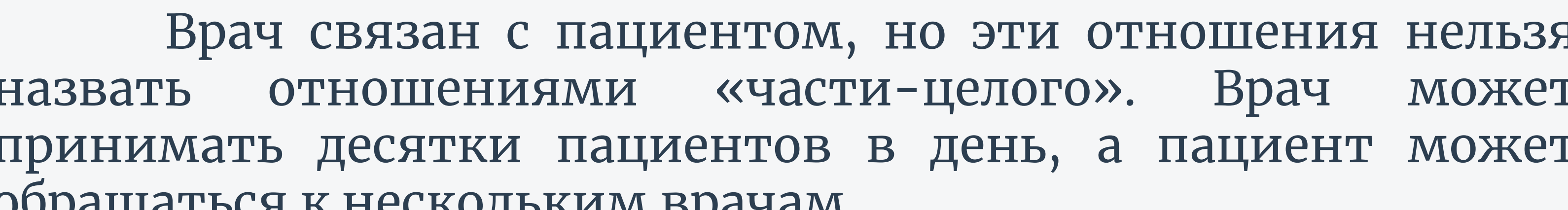
Ассоциация

В ассоциации два несвязанных объекта должны соответствовать следующим отношениям:

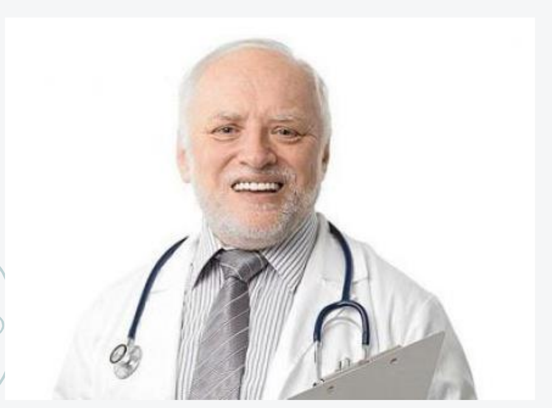
- Первый объект (член) не связан со вторым объектом (классом).
- Первый объект (член) может принадлежать одновременно сразу нескольким объектам (классам).
- Первый объект (член) существует, не управляемый вторым объектом (классом).
- Первый объект (член) может знать или не знать о существовании второго объекта (класса).



Ассоциация



Врач связан с пациентом, но эти отношения нельзя назвать отношениями «части-целого». Врач может принимать десятки пациентов в день, а пациент может обращаться к нескольким врачам.



Типом отношений в ассоциации является «использует».

Врач «использует» пациента для получения дохода. Пациент «использует» врача, чтобы вылечить болезнь или улучшить свое самочувствие.

Композиция. Агрегация. Ассоциация

Свойства	Композиция	Агрегация	Ассоциация
Отношения	Части-целое	Части-целое	Объекты не связаны между собой
Члены могут принадлежать одновременно сразу нескольким классам	Нет	Да	Да
Существование членов управляется классами	Да	Нет	Нет
Вид отношений	Однонаправленные	Однонаправленные	Однонаправленные или Двунаправленные
Тип отношений	«Часть чего-то»	«Имеет»	«Использует»

Зависимость

В повседневной жизни мы используем термин «зависимость», чтобы указать, что один объект зависит от второго объекта для выполнения определенного задания. Например, если кто-то сломает ногу, то будет зависеть от костылей, чтобы иметь возможность передвигаться (но не наоборот). Цветковые растения зависят от пчёл, которые опыляют их, чтобы те имели возможность размножаться (но не наоборот).

Зависимость возникает, когда один объект обращается к функционалу другого объекта для выполнения определенного задания. Эти отношения слабее отношений в ассоциации, любое изменение объекта, который предоставляет свой функционал зависимому объекту, может стать причиной сбоя работы зависящего объекта. Зависимость всегда является однонаправленной.

Зависимость. Ассоциация

А чем зависимость отличается от ассоциации?

В языке С++ ассоциация – это отношения между двумя классами на уровне классов. То есть, первый класс сохраняет прямую или косвенную связь со вторым классом через переменную-член. Например, в классе Врач есть массив указателей на объекты класса Пациент в виде переменной-члена. Всегда можно спросить у Врача, кто его Пациенты. Класс Водитель содержит Идентификатор Автомобиля в виде целочисленной переменной-члена. Водитель всегда знает к чему привязан Автомобиль, и как получить к нему доступ.

Зависимости обычно не представлены на уровне классов, то есть зависимый объект не связан со вторым объектом через переменную-член. Зависимый объект обычно создается при необходимости.