ООП

ООП в C++.

Черновик

Кафедра ИВТ и ПМ

2019

План

Outline

Прошлые темы

- Опишите парадигму ООП
- Чем она отличается от парадигмы процедурного и модульного программирования?
- Из каких элементов строится программа написанная согласно парадигме объектно-ориентированного программирования ?
- Что такое класс?
- Что такое объект?
- Чем отличается класс от объекта?
- Что такое поле класса?
- Что такое метод класса?

Прошлые темы

- Для чего нужен this?
- Какие модификаторы доступа могут применяться к атрибутам класса?

Outline

Описание класса в C++

```
class ClassName {
private:
  // закрытые члены класса
  // рекомендуется для описания полей
public:
 // открытые (доступные извне) члены класса
 // рекомендуется для описания интерфейса
protected:
  // защищенные члены класса
  // доступны только наследникам

// дружественные функции и классы
// модификатор доступа не важен
friend заголовок-функции;
friend имя_класса;
};
```

Объекты и обращение к методам

Пример

```
class MyClass {
        float _x;
    public:
        int n;
        void foo() const { cout << "foo" << endl;}
        float bar() const {return 42;}                    };

int main(){
    MyClass c; // статическое создание объекта

    // обращение к полям
    c.n = 42;
    // c._x не доступно

    // вызов метода
    c.foo();

    // вызов метода и запись возвращаемого значения в переменную
    float a = c.x();
        }
```

Outline

Объекты

Пример. Динамическое создание объекта

```
int main(){
    MyClass *c1 = new MyClass(); // динамическое создание объекта
    // c1 - указатель на объект

    // обращение к полям
    c1->n = 43;

    c1->foo();

    float b = c1->x();

    // после динамического создания объектов нужно
    // освободить занимаемую ими память
    delete c1;
}
```

Объекты

Пример. Указатели и ссылки

```
int main(){
    MyClass c3;

    // Объявление ссылки на объект
    // ссылка обязательно инициализируется
    MyClass &c4 = c3;
    // c4 и c3 идентичны

    // В остальном работа с ссылками на объект
    // не отличается от работы с самим объектом
    c4.n = 43;

    k = c3.n; // k = 43

    c4.bar();
    }
```

Outline

Объекты и обращение к методам

Пример. Массивы объектов

```
unsigned n = 20;
MyClass cc[n]; // статически созданный массив из объектов

for (unsigned i = 0; i<n; i++){
    cc[i].n = rand();
    cc[i].foo();
    }

// Запись отдельного объекта из массива в отдельную переменную
MyClass mc = cc[2];
mc.foo();

// динамический массив из объектов с заранее заданным количеством n
vector<MyClass> v(n);
for (unsigned i =0; i<v.size(); i++) {
    v[i].n = rand();
    v[i].foo();
}
```

Объекты и обращение к методам

Пример. Массивы объектов

```
// класс vector - динамический массив из объектов
// он удобнее классических динамических массивов
vector<MyClass> v;
unsigned n = 10;

// добавление объектов в динамический массив
for (unsigned i =0; i<n; i++) {
    MyClass mc;
    mc.n = rand();
    v.push_back(mc);
}

for (unsigned i =0; i<v.size(); i++) {
    v[i].foo();
}
```

Объекты и обращение к методам

Пример. Массивы из указателей

```
// динамический массив из указателей на MyClass
vector<MyClass*> v;
unsigned n = 10;

// добавление объектов в динамический массив
for (unsigned i = 0; i<n; i++) {
    MyClass *mc = new MyClass();
    mc->n = rand();
    v.push_back(mc);
}

for (unsigned i=0; i<v.size(); i++) {
    v[i]->foo();
}

// освобождение памяти, занимаемой объектами
for (unsigned i=0; i<v.size(); i++) {
    delete v[i];
}
```

Outline

Контроль постоянства

http://alenacpp.blogspot.com/2005/10/mutable-constcast.html

Outline

Статические члены классов

Outline

Конструктор

Конструктор — это особый метод, инициализирующий экземпляр своего класса.

```
class MyClass{
    float x, y;
    public:

        // Это конструктор
        MyClass(){
            x = 0;
            y = 42;
            cout << "new object";}
};

...

MyClass o1;                    // new object
MyClass *o2 = new MyClass(); // new object
```

Конструктор

- Имя конструктора совпадает с именем класса[1].
- Тип возвращаемого значения не указывается.
- У конструктора может быть любое число параметров.
- У класса может быть любое число конструкторов.
- Конструкторы могут быть доступными (public), защищенными (protected) или закрытыми (private).
- Если не определено ни одного конструктора, компилятор создаст конструктор по умолчанию, не имеющий параметров (а также некоторые другие к. и оператор присваивания)

[1] конструктор в python называется `__init__`

Деструктор

Деструктор — специальный метод класса, служащий для деинициализации объекта (например освобождения памяти).

```
class MyClass{
    float x, y;
    public:

        // Конструктор
        MyClass();

        // Деструктор
        ~MyClass();
};
```

Деструктор

- Деструктор - метод класса
- Объявление деструктора начинается с символа ~
- У деструкторов нет параметров и возвращаемого значения.
- В отличие от конструкторов деструктор в классе может быть только один.
- Деструктор вызывается *автоматически* при удалении объекта
- Если деструктор не определён, то он будет создан компилятором
- Такой деструктор не будет выполнять никакой работы
- Деструкторы как правило нужны если объекту необходимо освободить ресурсы, например закрыть файл; освободить память, выделенную вручную и т.п.

Деструктор. Пример

```
class MyClass{
    float x, y;
    public:
        // Конструктор
        MyClass();
        // Деструктор
        ~MyClass() {cout << "I'm Finished"; }
};

int main(){
MyClass c1, c2;

if ( 1 ){
    MyClass c3;}
// вызов деструктора c3;

cout << "End.";
// вызов деструторов c1 и c2
}
```

Outline

Конструкторы

- конструктор умолчания (default constructor)
- Конструктор с параметрами:
  - конструктор преобразования (conversion constructor)
  - конструктор с двумя и более параметрами (parameterized constructors
- конструктор копирования (copy constructor)
- конструктор перемещения (move constructor)

Конструкторы

Компилятор выбирает тот конструктор, который удовлетворяет ситуации по количеству и типам параметров.

В классе не может быть двух конструкторов с одинаковым набором параметров.

Конструктор по умолчанию (Default constructor)

`MyClass()`

- Не имеет параметров.
- Может быть только один.
- Может отсутствовать.
- Создаётся компиляторам, если отсутствует

Когда вызывается

```
class MyClass {...};
...

MyClass c0 = MyClass();
MyClass c1;
MyClass cv[16];       // к. будет вызван 16 раз
list<MyClass> cl(10) // к. будет вызван 10 раз
```

Конструктор с параметрами (Parametrized constructor)

- Принимает несколько параметров

Общий вид:

```
MyClass(T1 t1, T2 t2, T3 t3, .... )
```

T1, T2, T3, ... - некоторые типы

Конструктор преобразования (Conversion constructor)

Общий вид:

`MyClass(T t)`

T - некоторый тип

- Принимает один параметр
- Тип параметра должен отличатся от самого класса
- Такой конструктор как бы преобразует один тип данных в экземпляр данного класса
- Может вызываться при инициализации объекта значением принимаемого типа

  `MyClass c = t`

Конструктор с параметрами (parametrized constructor)

Пример

```
class Point{
    float _x, _y;
    public:
        Point() { _x = 0; _y = 0; }

        Point(float x) { _x = x; }

        Point(float x, float y){
                _x = x;
                _y = y;}

        Point(const vector<float> &v){// v - вектор из двух значений
            if (v.size() == 2){_x = v[0]; _y = v[1];}
            else throw "Vector Size Error";}
        // ...
};
```

Outline

Пример

Какие конструкторы будут вызваны для объектов типа Point?

```
int main(){
    Point p1;
    Point p11 = Point();
    Point *pp1 = new Point();

    Point p2(2);
    Point *pp2 = new Point(42);

    Point p3(1.5, -1);
    Point *pp3 = new Point(-10.7, 127.2);

    vector<float> v = {1,2};
    Point p4(v);
    Point *pp4 = new Point(v);


    Point pa[3] = {Point(), Point(), Point(2.3)};
}
```

Пример

```
int main(){
    Point p1;                       // к. по умолчанию
    Point p11 = Point();            // к. по умолчанию (явный вызов)
    Point *pp1 = new Point();       // к. по умолчанию

    Point p2(2);                    // к. преобразования
    Point *pp2 = new Point(42); // к. преобразования

    Point p3(1.5, -1);              // к. с параметрами
    Point *pp3 = new Point(-10.7, 127.2); // к. с параметрами

    vector<float> v = {1,2};
    Point p4(v);                    // к. преобразования
    Point *pp4 = new Point(v);  // к. преобразования

    // Явный вызов конструкторов
    Point pa[3] = {Point(), Point(), Point(2.3)};
}
```

Пример

```
int main(){
    Point *pp5;           // к. не вызывается
    Point *pp6 = &p1;     // к. не вызывается
    Point *pp7 = &p3;     // к. не вызывается
    vector<Point* > vp0;// к. не вызывается

    Point p5 = 5;         // к. преобразования
    Point p6 = v;         // к. преобразования

    Point pp[3] = {0.2, 3, -4.2}; // к. преобразования
    vector<Point> vp3 = {0.2, 3, -4.2};// к. преобразования

    vector<Point> vp;             // к. не вызывается
    vector<Point> vp2(10);        // к. по умолчанию
}
```

Конструктор копирования (copy constructor)

- Один параметр: ссылка на экземпляр данного класса
- Необходим, если простого (поверхностного) копирования всех полей класса недостаточно

```
MyClass(MyClass &c)
```

Конструктор перемещений (move constructor)

```
MyClass(MyClass &&c)
```

Outline

Оператор присваивания копированием (assignment operator)

```
MyClass& operator=(MyClass& data)
```

- используется для присваивания одного объекта текущему (существующему)
- генерируется автоматически компилятором если не объявлен
- сгенерированный компилятором, выполняет побитовое копирование
- должен очищать поля цели присваивания (и правильно обрабатывать самоприсваивание)

Оператор присваивания перемещением (move assignment operator)

```
MyClass& operator = (const MyClass &c)
```

- используется для присваивания *временного* объекта существующему
- "забирает"временный объект "в себя"; временный объект перестаёт существовать
- генерируется автоматически компилятором если не объявлен
- сгенерированный компилятором, выполняет побитовое копирование
- должен очищать поля цели присваивания (и правильно обрабатывать самоприсваивание)

Когда вызывается?
Когда существующему объекты присваиваю значение временного объекта.

Правило пяти

Если класс или структура определяет один из следующих методов, то нужно явным образом определить все методы:

- Конструктор копирования
- Конструктор перемещения
- Оператор присваивания копированием
- Оператор присваивания перемещением
- Деструктор

Спецификаторы default и delete

Спецификаторы **default** и **delete** заменяют тело метода.

Спецификатор **default** означает реализацию по умолчанию (компилятором). Может быть применён только к конструкторам, деструктору и операторам присваивания.

Спецификатором **delete** помечают те методы, работать с которыми нельзя.

Спецификаторы default и delete

```
class Foo{
public:
    Foo() = default;
    Foo(const Foo&) = delete;
    Foo operator = (const Foo& f) = delete;
};

...

Foo o1, o2; // вызов констр. созданного компилятором
o1 = o2; // Ошибка компиляции! Оп-р присваивания запрещён.
Foo o3(01); // Ошибка компиляции! Констр. копирования запре
```

Защита от копирования

Чтобы запретить копирование объекта достаточно определить в закрытой части класса (private) конструктор копирования и оператор присваивания. Тела этих методов можно оставить пустыми.

Вопросы

- Зачем нужны конструкторы?
- Как запретить создание объекта на основе уже существующего?
- Как запретить любой другой способ создания объекта?
- Зачем нужны конструкторы перемещения? В чём их отличие от к. копирования?
- Когда вызывается конструктор, а когда оператор присваивания?
- Что если не описать ни одного конструктора?
- Что если не описать ни одного оператора присваивания?

Outline

Перегрузка операторов (operator overloading)

- Что такое перегрузка функций?
- Как должны отличатся перегруженные функции?

Перегрузка операторов (operator overloading)

- Что такое перегрузка функций?
- Как должны отличатся перегруженные функции?
- Что такое перегрузка методов?
- Что такое оператор?
- Что такое операнд?
- Что такое арность оператора?

Перегрузка операторов (operator overloading)

Код может выглядеть логичнее и читаться лучше если использовать операторы.

Без использования операторов:

```
// комплексное число
class Complex {
    float im, re;  //мнимая и действительная часть

public:
// ...
// Метод не изменяет текущий объект, а создаёт новый
// поля суммируются и записываются в новый объект
Complex plus (const Complex& и){
    Complex result;
    result.im = this->im + b.im;
    result.re = this->re + b.re;
    return result; }
};

Complex a, b;
Complex c = a.plus(b);
```

Перегрузка операторов (operator overloading)

```
Complex a, b;
Complex c = a.plus(b);

// если бы оператор сложения (+) был перегружен для класса Compl
// то сумма выглядела бы лаконичнее:

Complex a, b;
Complex c = a + b;
```

Перегрузка операторов (operator overloading)

Общий вид перегружаемого оператора:

```
ReturnType operator opr ( parameters );
```

`type` - тип возвращаемого значения

`opr` - обозначение оператора (например +, *, = и др.)

`parameters` - параметры, описываются также как и в функции

Такое объявление похоже на объявление функции за исключением того, что используется ключевое слово `operator` и вместо имени функции указывается обозначение оператора.

Список доступных операторов: Операторы в C и C++

Перегрузка операторов (operator overloading)

Оператор может быть перегружен как отдельная функция и как метод класса.

Если оператор определяется как отдельная функция (часто такие операторы определяются дружественными функциями Число параметров функции должно соответствовать арности оператора. Например для бинарных операторов (`+`, `-`, `*` и др) параметра два.

Когда перегруженный оператор является методом класса, тип первого операнда должен быть указателем на данный класс (всегда this), а второй должен быть объявлен в списке параметров.

Перегрузка операторов

оператор как функция

- Перегрузим оператор сложения как функцию для класса `Complex`.
- Это бинарный оператор, поэтому у оператора будет два аргумента - комплексные числа.
- После сложения двух чисел должно получится тоже комплексное число, поэтому и возвращаемый тип данных тоже `Complex`.

Общий вид оператора:

```
Complex operator + (const Complex &a, const Complex &b);
```

Перегрузка операторов

оператор как функция

```
class Complex {
    public:    float im, re;
    // ...

friend Complex operator + (const Complex &a, const Complex &b);
};

Complex operator + (const Complex &a, const Complex &b){
    Complex result;
    // этот оператор дружественный для класса Complex
    // поэтому имеет доступ к его закрытым членам
    result.im = a.im + b.im;
    result.re = a.re + b.re;
    return result;          }

Complex a, b;
Complex c = a + b;
// аналогично вызов оператора можно записать:
Complex d = operator+(a, b);
```

Перегрузка операторов

оператор как метод

- Каждому методу любого класса неявно передаётся параметр - this
- Поэтому первым параметром оператора определяемого внутри класса *всегда* будет объект данного класса
- тело оператора будет таким же как и в методе plus (на предыдущих слайдах)

```
class Complex {
    // ...
    Complex operator + (const Complex &b);
    // ...
};
```

Перегрузка операторов

оператор как метод

```
class Complex {
    Complex operator + (const Complex &b){
        Complex result;
        // этот оператор дружественный для класса Complex
        // поэтому имеет доступ к его закрытым членам
        result.im = a.im + b.im;
        result.re = a.re + b.re;
        return result;          }
};

Complex a, b;
Complex c = a + b;
// аналогично вызов оператора можно записать:
Complex d = a.operator+(b);
```

Перегрузка операторов (overloading)

Когда оператор делать методом, а когда дружественной функцией?

Унарные операторы и бинарные операторы типа “X=” рекомендуется реализовывать в виде методов класса, а прочие бинарные операторы — в виде дружественных функций.
Так стоит делать потому, что оператор-метод всегда вызывается для левого операнда.

Перегрузка операторов (overloading)

Вопросы

- Будет ли компилироваться следующий код? Почему?

```
Complex a, b;
Complex c = a + 42;
```

Перегрузка операторов (overloading)

Вопросы

- Будет ли компилироваться следующий код? Почему?

```
Complex a, b;
Complex c = a + 42;
```

- Будет ли компилироваться следующий код? Почему?

```
class Complex {
// ...
Complex operator + (double b);
// ...
                                        };
Complex a;
Complex c = a + 42;
Complex e = 42 + a;
```

Перегрузка операторов (overloading)

Вопросы

- Будет ли компилироваться следующий код? Почему?

```
Complex a, b;
Complex c = a + 42;
```

- Будет ли компилироваться следующий код? Почему?

```
class Complex {
// ...
Complex operator + (double b);
// ...
                                        };
Complex a;
Complex c = a + 42;
Complex e = 42 + a;
```

- В чём разница между вызовами оператора в двух последних строчках?

Перегрузка операторов (overloading)

Вопросы

- Будет ли компилироваться следующий код? Почему?

```
Complex a, b;
Complex c = a + 42;
```

- Будет ли компилироваться следующий код? Почему?

```
class Complex {
// ...
Complex operator + (double b);
// ...
                                        };
Complex a;
Complex c = a + 42;
Complex e = 42 + a;
```

- В чём разница между вызовами оператора в двух последних строчках?
- Как определить оператор чтобы последний вариант вызова оператора работал?

Outline

Пример

- Определение класса.
- Создание экземпляров класса (объектов), вызов методов.
- Указатели. Динамическое создание экземпляров класса, вызов методов.
- Создание динамического массива (vector) из объектов.
- Создание динамического массива (vector) из указателей на объекты.

github.com/VetrovSV/OOP/tree/master/examples/simple_class

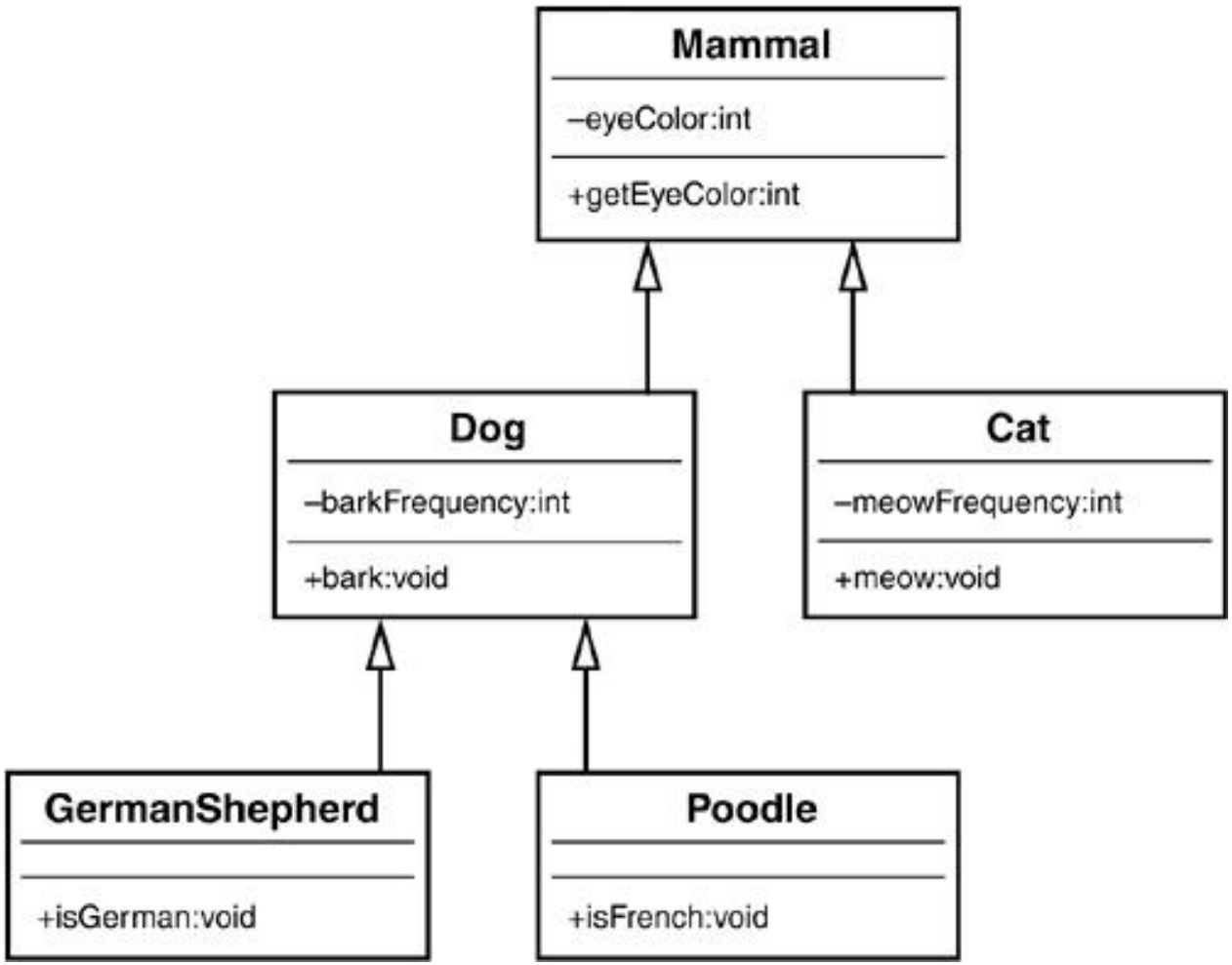
Outline

Отношения между классами

- обобщение/специализация (generalization/specialization)
  кошки – это животные
- целое/часть (whole/part)
  двигатель – часть автомобиля
- ассоциация (семантическая зависимость)
  художник – кисть

Обобщение\специализация

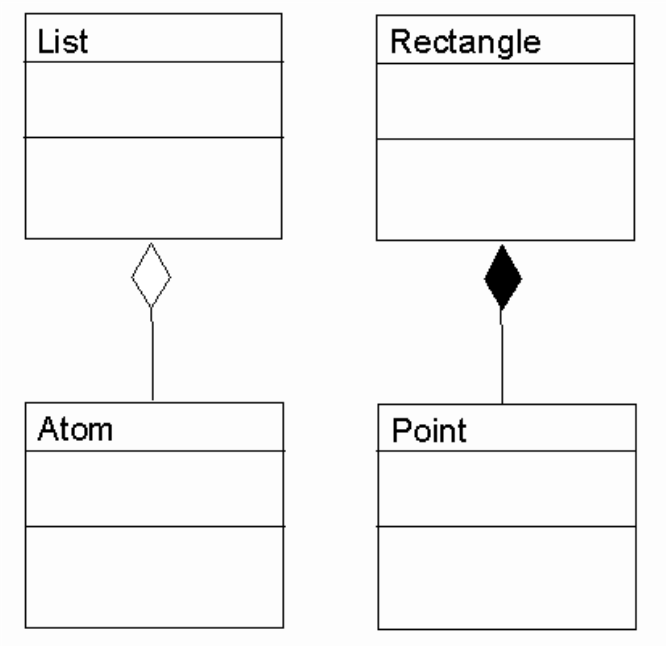
Наследование (inheritance)



Стрелка всегда указывает на базовый (родительский) класс.

Часть\целое

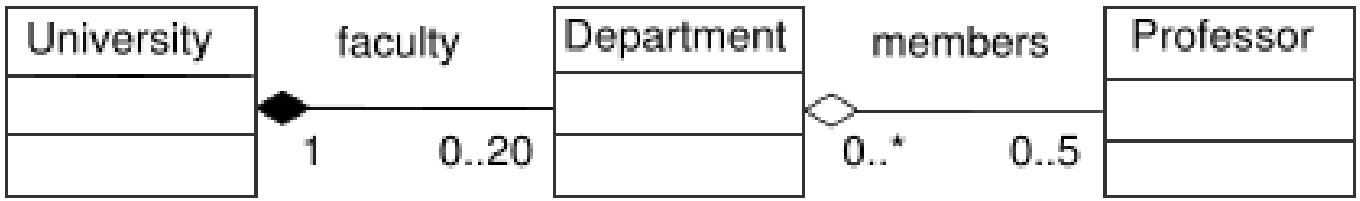
Агрегация и композиция



Агрегация (aggregation):
агрегирующий объект (list на рис. слева) может существовать и не включая в себя экземпляры агрегируемого (в примере atom)

Композиция (composition): один объект (Rectangle на втором рис. слева) не может существовать без своих составляющих (в примере Point)
Композиция – более строгий вариант агрегации.

Композиция связывает время жизни объектов: если контейнер будет уничтожен, то всё его содержимое будет также уничтожено.



Агрегация: профессора – факультеты, профессора остаются жить после разрушения факультета

Композиция: университет – факультеты, факультеты без университета погибают.

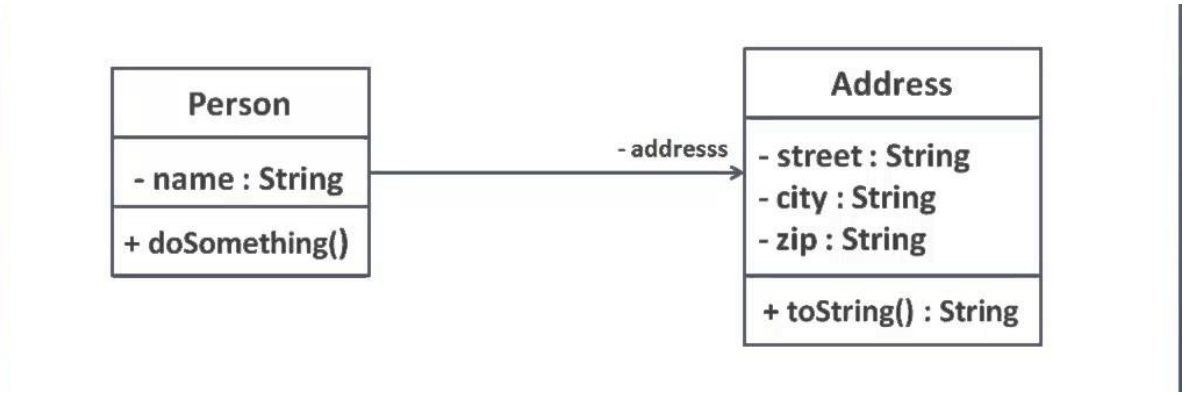
Наследование и агрегация

Часто при построении иерархии классов приходится выбирать между наследованием и агрегацией (композицией)

Наследование – сильный вид отношения, поэтому к нему следует прибегать с осторожностью рассматривая в качестве альтернативы агрегацию.

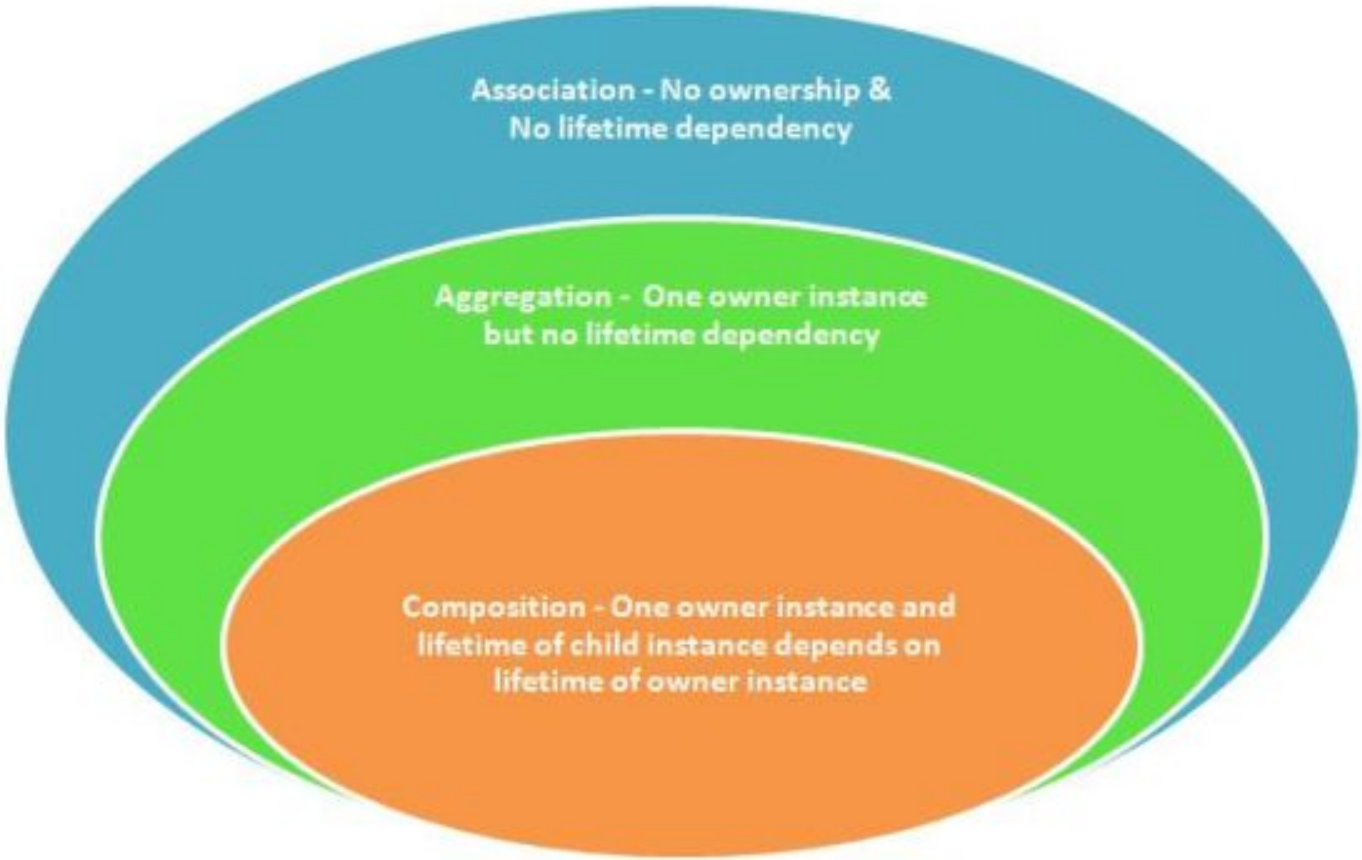
Ассоциация

Ассоциация - самый слабый вид отношения из перечислены.



- Как правило при ассоциации один класс так или иначе использует другой, но не владеет им.
- Например один класс может хранить ссылку на другой, однако при удалении объекта первого класса, второй продолжит существовать.
- Ассоциация может быть направленной (изображается стрелкой), а может и не иметь направления (без стрелки)

Ассоциация - Агрегация - Композиция



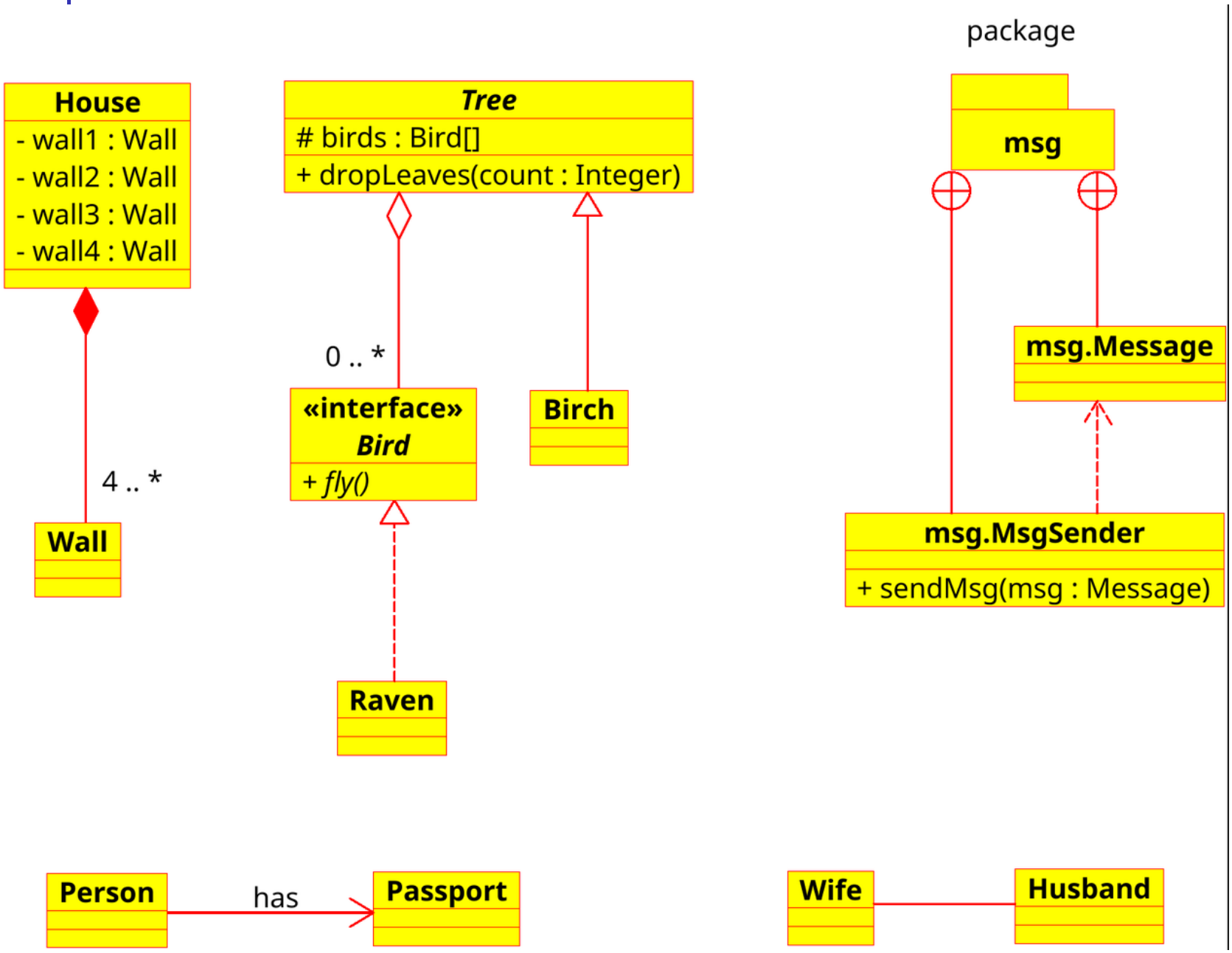
Мощность отношений (Кратность)

Мощность отношения (мультипликатор) означает число связей между каждым экземпляром класса (объектом) в начале линии с экземпляром класса в её конце.

Мощность:

- **0..1** Ноль или один экземпляр
  кошка имеет или не имеет хозяина
- **1** Обязательно один экземпляр
  у кошки одна мать
- **0..* или *** Ноль или более экземпляров
  у кошки могут быть, а может и не быть котят
- **1..*** Один или более экземпляров
  у кошки есть хотя бы одно место, где она спит

Пример


House
- wall1 : Wall
- wall2 : Wall
- wall3 : Wall
- wall4 : Wall
4 .. *
Wall
Tree
birds : Bird[]
+ dropLeaves(count : Integer)
0 .. *
«interface»
Bird
+ fly()
Birch
Raven
package
msg
msg.Message
msg.MsgSender
+ sendMsg(msg : Message)
Person
has
Passport
Wife
Husband

Outline

Реализация отношений между классами в C++

- Зависимость (dependency)
  Один класс использует другой класс, например принимает в параметре.
- Ассоциация
  Поле одного класса может быть указателем на другой класс. Два класса могут иметь указатели на друг друга. Нельзя сказать, что один объект есть часть другого (по смыслу).
- Агрегация
  Поле одного класса (контейнера) может быть указателем на другой класс, при этом первый класс может управлять временем жизни агрегируемого объекта.
- Композиция
  Объект-контейнер может содержать в себе другие объекты или указатели на них. При уничтожении контейнера уничтожатся и его содержимое.

Наследование рассматривается в отдельной лекции.

Композиция

```
class Part{
    // ...
};

class Big{
    Part filed1;
}
```

Big включает в себя Part. Причем при создании экземпляра класса Big будет обязательно создан экземпляр класса Part;

В Big можно включить несколько экземпляров класса Part, например использовав массив.

Big может хранить указатель на Part, создавать экземпляр Part в своём конструкторе и уничтожать в деструкторе.

Агрегация

```
class Part{
    // ...
};

class Big{
    Part *filed1;
}
```

Класс Big может хранить в себе указатель на Part, но экземпляр класса Big можно создать и отдельно;

Если того требует мощность отношения в Big можно включить несколько экземпляров класса Part, например использовав массив:

```
vector<Part*> filed1;
```

Агрегация

```
class Doctor{
    // ...
};

class Patient{
    Doctor *filed1;
}
```

Класс Patient знает про ассоциированный с ним объект типа Doctor. Может менять ссылку на Doctor, но не может удалять экземпляр класса Doctor. Ассоциация не потеряет своего смысла, если в классе Doctor добавить указатель (или массив указателей) на Patient.

Синтаксически, эта форма ассоциации, может быть похожа на агрегацию. Однако объекты здесь более независимы.

В зависимости от своего характера ассоциация, может быть представлена и иначе.

Реализация отношений между классами в C++

- What is the difference between dependency and association?

Outline

struct vs class

разница между struct и class

- В struct модификаторы доступа по умолчанию public, в class private
- Наследование по умолчанию у struct — public, у class — private

Память занимаемая объектом

Память занимаемая объектом складывается из памяти занимаемой

- полями класса + остаток для выравнивания (по умолчанию выравнивание 4 байта)
- указателем на vtable (если есть виртуальные функции)
- указателями на классы предков, от которых было сделано виртуальное наследование (размер указателя * количество классов)

Не освещенные темы

- Структуры vs объединения vs классы
- inline методы
- RAII
- Почему в обработчике исключений C++ нет раздела finalize?
- Статические члены класса
- Ссылки на методы класса
- ...

Ссылки и литература

1. stepik.org/course/7 - Программирование на языке C++
   курс по C++ и ООП от Computer Science Center (CS центр)
2. Г. Буч. Объектно-ориентированный анализ и проектирование с примерами приложений. 720 с. 2010 г. 700 страниц. Теория. Примеры на C++. Картинки! Вторая половина книги - примеры OOA и OOD с UML диаграммами.
3. MSDN - Microsoft Developer Network
4. Qt 5.X. Профессиональное программирование на C++. Макс Шлее. 2015 и более поздние издания г. 928 с. Книга периодически обновляется с выходом новых версий фреймворка Qt.
5. www.stackowerflow.com - система вопросов и ответов
6. draw.io — создание диаграмм.

Материалы курса

Слайды, вопросы к экзамену, задания, примеры

github.com/VetrovSV/OOP