Лекция 2

Методы класса

Методы

- Функции, определенные внутри структуры
- Отличие заключается в прямом доступе к полям структуры
- Обращение к методу аналогично обращению к полям

```
struct Segment{
    Point2D start;
    Point2D end;
    double length(){
        double dx = start.x - end.x;
        double dy = start.y - end.y;
        return sqrt(dx * dx + dy * dy);}
};
int main(){
    Segment segment = {{0.4, 1.4}, {1.2, 6.3}};
    std::cout << segment.length() << '\n';}
```

Реализация метода вне класса

- В классе только сигнатура функции
- Имя функции надо указывать с названием класса через оператор ::

```
struct Segment{
    Point2D start;
    Point2D end;
    double length();
};

double Segment::length(){
    double dx = start.x - end.x;
    double dy = start.y - end.y;
    return sqrt(dx * dx + dy * dy);}

int main(){
    Segment segment = {{0.4, 1.4}, {1.2, 6.3}};
    std::cout << segment.length() << '\n';}
```

Объявление и определение методов

- Как и для обычных функций, можно разделять объявление и определение

```
//point.h
struct Point2D {
    double x;
    double y;
    void shift(double x, double y);};
```

```
#include "point.h"
void Point2D::shift(double x, double y){
    this->x += x;
    this->y += y;
}
```

Неявный указатель this

- Методы реализованы как обычные функции, имеющие дополнительный параметр
- Неявный параметр является указателем типа класса и имеет имя this
- Можно считать, что настоящая сигнатура методов следующая:

```
struct Point2D {
    double x;
    double y;
    void shift(/* Point2D *this,*/ double x, double y){
        this->x += x;
        this->y += y; }
};
```

- Позволяет обратиться к полям объекта при перекрытии имен

Перегрузка функций

- Определение нескольких функций с одинаковым именем, но отличающимся списком параметров (типами и/или количеством)

```
class Point2D {
    int x;
    int y;
public:
    void move(int dx, int dy);
    void move(Point2D vector);
} zero;
int main() {
    Point2D point;
    point.move(10, 20);
    zero.move(point);
}
```

Инвариант класса

- Публичный интерфейс – список методов, доступный внешним пользователям класса

- Инвариант класса – набор утверждений, которые должны быть истинны применительно к любому объекту данного класса в любой момент времени, за исключением переходных процессов в методах объекта

- Для сохранения инварианта класса:
  - Все поля должны быть закрытыми
  - Публичные методы должны сохранять инвариант

Определение констант

- Ключевое слово const позволяет определять типизированные константы
- Попытка изменений таких значений пресекается компилятором
- Попытка изменить константные данные приводит к неопределённому поведению

Константные методы

- Методы классов и структур могут быть помечены модификатором const
- Такие методы не могут изменять поля объекта
- Указатель this является Type const * this
- У константных объектов можно вызывать только константные методы
- Является частью сигнатуры метода

Константные методы

- Перегрузка через const позволяет делать версии для константных и не константных объектов

```
class IntArray {
    int size;
    int *data;
public:
    int get(int index) const {
        return data[index];
    }
    int &get(int index) {
        return data[index];
    }
};
```

Синтаксическая и логическая константность

- **Синтаксическая** – константные методы не могут модифицировать поля (обеспечивается компилятором)
- **Логическая** – запрещено изменение данных, определяющих состояние объекта в константных методах (обеспечивается разработчиком)

```
class IntArray {
    mutable int size; //Нарушение логической константности
    int *data;
public:
    void method() const {
        data[10] = 1; //Нарушение логической константности
    }
};
```

Ключевое слово mutable

- Позволяет определять поля, доступные для изменения внутри константных методов
- Можно использовать только с полями, не являющимися частью состояния объекта

```
class IntArray {
    int size;
    int *data;
    mutable int counter;
public:
    int size() const {
        ++counter;
        return size;
    }
};
```

```
struct Example {
    int n1;
    mutable int n2;
};
int main(){
    const Example a;
    a.n1 = 2; //ошибка
    a.n2 = 2;
}
```

Ключевое слово static (поля класса)

- Модификатор static создает поле класса, разделяемое между всеми объектами класса
- static поле инициализируется во время запуска программы
- Изменение static поля в одном объекте класса видно во всех остальных объектах класса
- static поля противоречат концепции инкапсуляции

Ключевое слово static (методы)

- static метод класса «не привязан» к объектам класса
- static метод класса может обращаться только к static полям класса
- static метод не имеет доступа к идентификатору this
- static методы нарушают принципы ООП

Создание объектов

Конструкторы

- Специальная функция объявляемая в классе
- Имя функции совпадает с именем класса
- Не имеет возвращаемого значения
- Предназначены для инициализации объектов

```
class Date {
    int year;
    int month;
    int day;
public:
    void init(int day, int month, int year);
    void setYear(int year);
    void setMonth(int month);
    void setDay(int day);
};
```

```
class Date {
    int year;
    int month;
    int day;
public:
    Date(int day, int month, int year);
    void setYear(int year);
    void setMonth(int month);
    void setDay(int day);
};
```

Перегрузка конструкторов

- Может быть объявлено несколько конструкторов
- Должны иметь разное количество или тип параметров

```
class Date {
    int year;
    int month;
    int day;
public:
    Date(int day, int month, int year);
    Date(int day, int month);
    Date(int day);
    Date();
    Date(char const *date);
};
```

Списки инициализации

- Предназначены для инициализации полей
- Инициализация происходит в порядке объявления полей

```
class Date {
    int year;
    int month;
    int day;
public:
    Date(int day, int month, int year):year(year),day(day),month(month){}
};
```

Значения по умолчанию

- Конструкторы, как и другие функции, могут иметь значения по умолчанию

- Значения параметров по умолчанию необходимо указывать при объявлении

Значения по умолчанию

```
class Date {
    int year;
    int month;
    int day;
public:
    Date(int day = 1, int month = 1, int year = 1970)
        :year(year), month(month), day(day) {}
    Date() {} // ОШИБКА
};

int main() {
    Date zero;
    Date days (10);
    Date daysAndMonths (10, 2);
}
```

Конструктор от одного аргумента

- Задает неявное преобразование от значения аргумента к значению класса/структуры

```
class Segment {
    Point first;
    Point second;
public:
    Segment() : first(0, 0), second(0, 0) {}
    Segment(int length) : first(0, 0), second(length, 0) {}
};

int main() {
    Segment first;
    Segment second(10);
    Segment third = 20;
}
```

Ключевое слово explicit

- Запрещает неявное преобразование

```
class Segment {
    Point first;
    Point second;
public:
    Segment() : first(0, 0), second(0, 0) {}
    explicit Segment(int length) : first(0, 0), second(length, 0) {}
};

int main() {
    Segment first;
    Segment second(10);
    Segment third = 20; // Compile error
}
```

Конструктор по умолчанию

- Создается компилятором
- Только если в классе отсутствует конструктор
- Не имеет параметров

```
class Segment {
    Point first;
    Point second;
public:
    Segment(Point first, Point second) : first(first), second(second) {}
};

int main() {
    Segment first; //Compile Error
    Segment second(Point(), Point(1, 2));
}
```

Ключевое слово default

- Позволяет явно задать конструктор по умолчанию

```
class Segment {
    Point first;
    Point second;
public:
    Segment() = default;
    Segment(Point first, Point second) : first(first), second(second) {}
};

int main() {
    Segment first;
    Segment second(Point(), Point(1, 2));
}
```

Делегирующий конструктор

- Позволяет вызывать конструктор из конструктора того же класса
- Сокращает дублирование кода

```
class Point {
    int x;
    int y;
public:
    explicit Point(int x = 0, int y = 0)
        :x(x), y(y){
        cout << x << " " << y << endl;
    }
    explicit Point(double y)
        :x(0), y(int(y)){
        cout << x << " " << y << endl;
    }
};
```

```
class Point {
    int x;
    int y;
public:
    explicit Point(int x = 0, int y = 0)
        :x(x), y(y){
        cout << x << " " << y << endl;
    }
    explicit Point(double y)
        :Point(0, int(y)) {}
};
```

Деструктор

- Специальная функция, объявляемая в классе
- Имя функции совпадает с именем класса, плюс знак ~ в начале
- Не имеет возвращаемого значения и аргументов
- Предназначены для освобождения используемых ресурсов
- Вызывается автоматически при удалении экземпляра класса / структуры

```
class IntArray {
    int _size;
    int *data;
public:
    explicit IntArray(int size)
        :_size(size), data(new int[_size]){}
    ~IntArray(){
        delete []data;
    }
};
```

Методы, генерируемые компилятором

- Конструктор по умолчанию
- Конструктор копирования
- Конструктор перемещения
- Операторы присваивания
- Деструктор

- Запретить создание можно спецификатором delete

```
class SomeClass{
public:
    SomeClass() = delete;
};
```

Копирование и перемещение объектов

lvalue и rvalue

- lvalue (locator value) представляет собой объект, который занимает идентифицируемое место в памяти (имеет имя и адрес)
- rvalue – всё, что не является lvalue

```
int main(){
    int a = 10; int b = 20;                          //1
    int arr[10] = {1, 2, 3, 4, 5, 5, 4, 3, 2, 1};    //2
    int &link1 = a;                                  //3
    int &link2 = a + b;                              //4
    int &link3 = *(arr + a / 2);                     //5
    int &link4 = *(arr + a / 2) + 1;                 //6
    int &link5 = (a + b > 10) ? a : b;               //7
}
```

Примеры

- Выражения в C++ делятся на:
  1. lvalue - выражения, значения которых являются ссылкой
  2. rvalue - временные значения, не соответствующие какой-то переменной / элементу массива
- Указатели и ссылки могут использоваться только с lvalue

Корректно

```
int var;
var = 4;
```

Не корректно

```
4 = var;          // Ошибка
(var + 1) = 4;    // Ошибка
```

Примеры

```
int foo() {
    return 2;
}

int main() {
    foo() = 2;
    return 0;
}
```

```
int globalvar = 20;

int& foo() {
    return globalvar;
}

int main() {
    foo() = 10;
    return 0;
}
```

```
int& foo() {
    return 2;
}
```

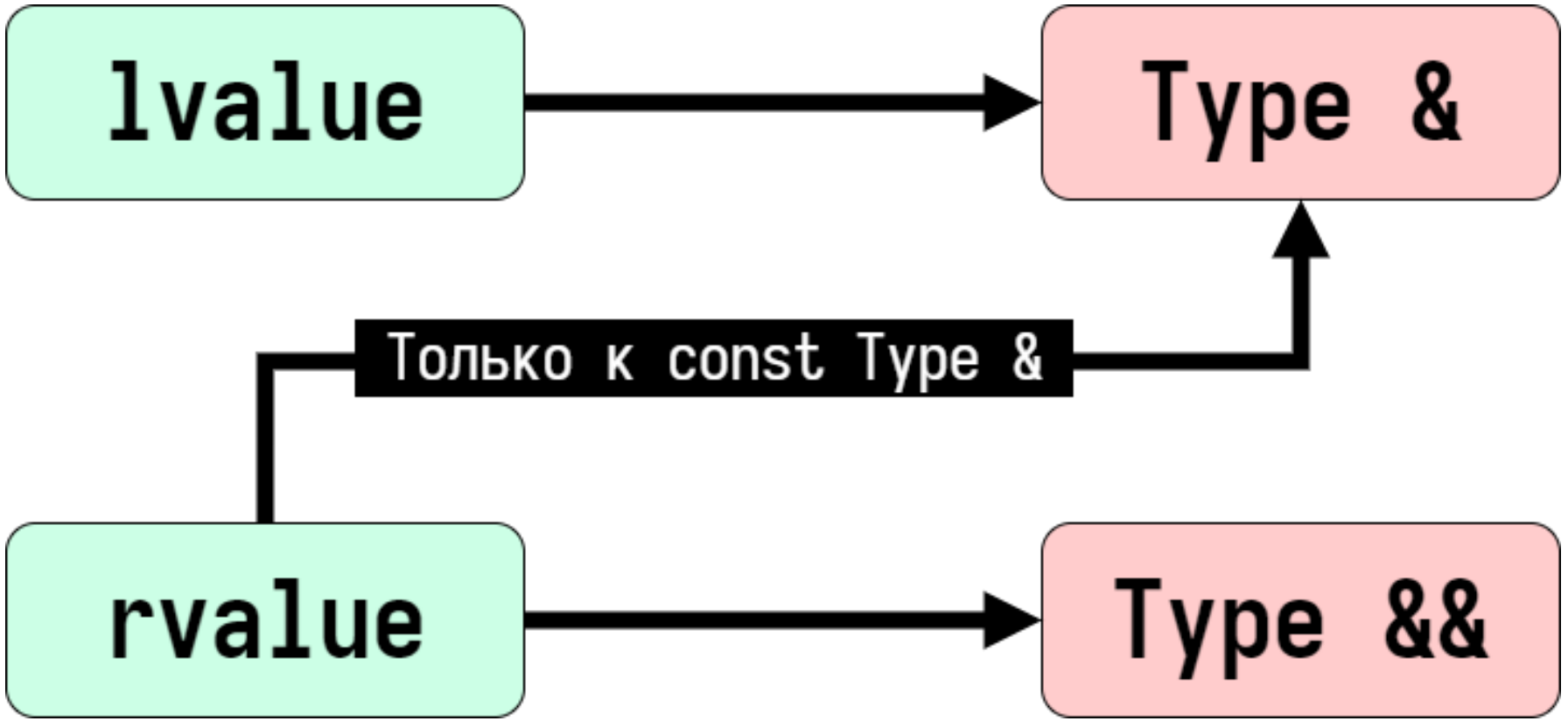
Преобразования между lvalue и rvalue

- Все операции со значениями требуют rvalue в качестве аргументов

```
int a = 1;  // a - lvalue int
b = 2;      // b - lvalue int
c = a + b;  // '+' требует rvalue
```

- В rvalue могут быть преобразованы все lvalue, которые не являются массивом, функцией и не имеют неполный тип
- Оператор * принимает rvalue и возвращает lvalue
- Оператор & принимает lvalue и возвращает rvalue
- Const ссылка может ссылаться на rvalue. Время жизни rvalue расширяется

lvalue и rvalue ссылки



Конструктор копирования

- Позволяет определить, каким образом будет происходить копирование объекта класса
- Правило, если реализован конструктор копирования, то необходимо реализовать оператор присваивания с копированием

```
class SomeClass{
public:
    SomeClass(const SomeClass& obj){
        //...
    }
};
```

Конструктор перемещения

- Позволяет избегать излишнего копирования
- Основывается на move-семантике (std::move и std::swap)

```
class SomeClass{
public:
    SomeClass(SomeClass&& obj){
        //...
    }
};
```

RAII

Идиома программирования

- Устойчивый способ выражения некоторой составной конструкции в языках программирования
- Шаблон решения задачи, алгоритма или структуры данных путем комбинирования встроенных элементов языка
- Может выглядеть по-разному в разных языках, либо в ней может не быть надобности в некоторых из языков

Идиома RAII

- Resource Acquisition Is Initialization - получение ресурса есть инициализация
- Идиома объектно-ориентированного программирования
- Основная идея - с помощью механизмов языка получение ресурса неразрывно совмещается с инициализацией , а освобождение – с уничтожением объекта
- Типичный способ реализации - получение доступа в конструкторе, а освобождение в деструкторе

- Применяется для:
  - Выделения памяти
  - Открытия файлов / устройств / каналов
  - Мьютексов / критических секций / других механизмов блокировки

Пример реализации RAII на C++

```
class File {
    std::FILE *file;
public:
    //Захват ресурса
    File(const char *filename) : file(std::fopen(filename, "w+")) {
        if (!file)
            throw std::runtime_error("file open failure");
    }
    //Освобождение ресурса
    ~File() {
        std::fclose(file);
    }
    //Взаимодействие с ресурсом
    void write(const char *data) {
        if (std::fputs(data, file) == EOF)
            throw std::runtime_error("file write failure");
    }
};
```

Пример реализации RAII на C++

```
int f(){
    try{
        File file("log.txt");
        file.write("information");
    }
    catch(std::runtime_error &e){
        e.what();
    }
}
```

Отношения классов

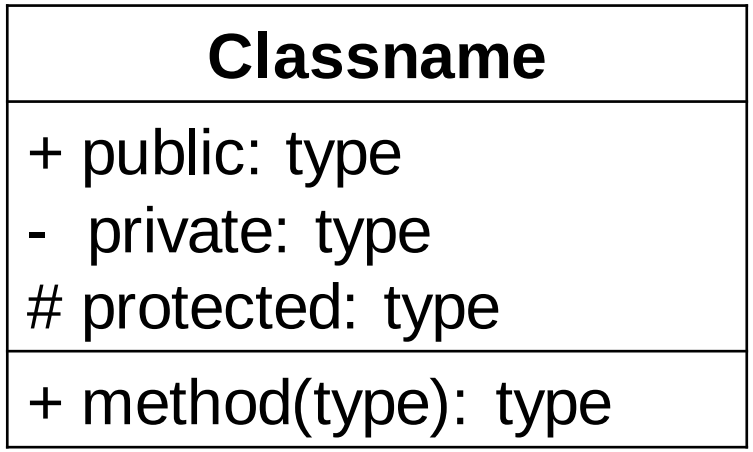
UML

https://www.uml-diagrams.org/

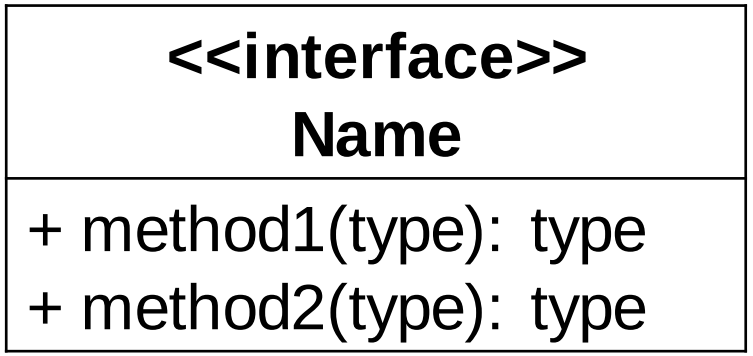
- UML – унифицированный язык моделирования (Unified Modeling Language)
- Включает в себя такие диаграммы как:
  - Диаграмма классов
  - Диаграмма пакетов
  - Диаграмма объектов
  - Диаграмма развертывания
  - Use-case диаграмма
  - Диаграмма состояний
  - Диаграмма последовательностей

Структурные сущности

Класс

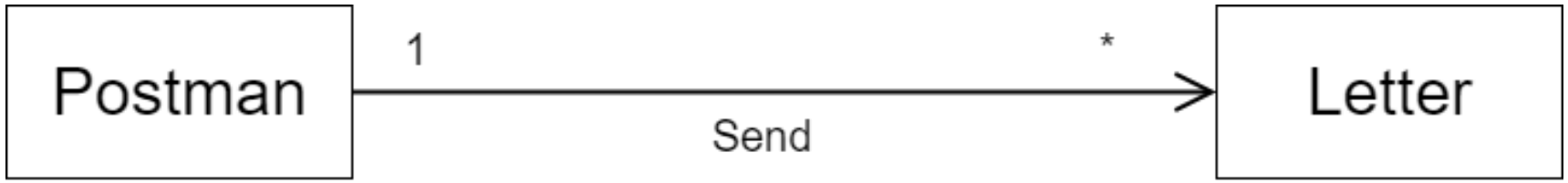


Интерфейс



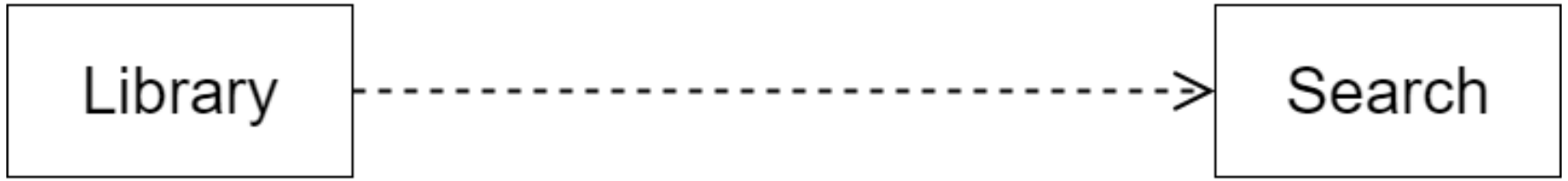
Ассоциация

- Означает, что классы связаны физически или логически
- Самая слабая связь



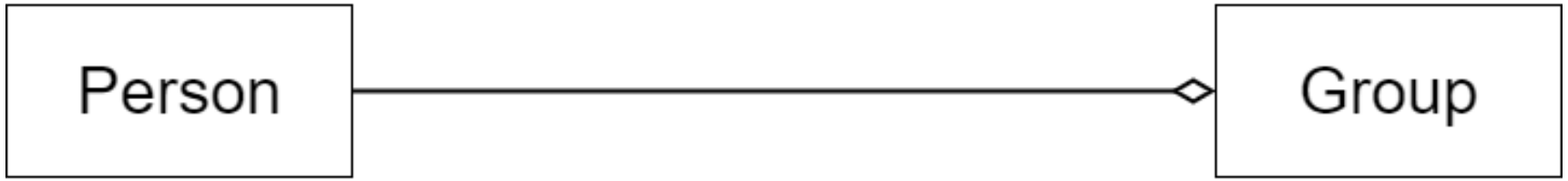
Зависимость

- Означает, что классы связаны напрямую физически
- Зависимость отображает то, что класс хранит другой класс, либо принимает его в качестве аргументов методов



Агрегация

- Определяет отношение HAS A – то есть отношение владения
- Описывает целое и составные части, которые в него в ходят
- Целое НЕ является владельцем части и НЕ управляет временем её жизни



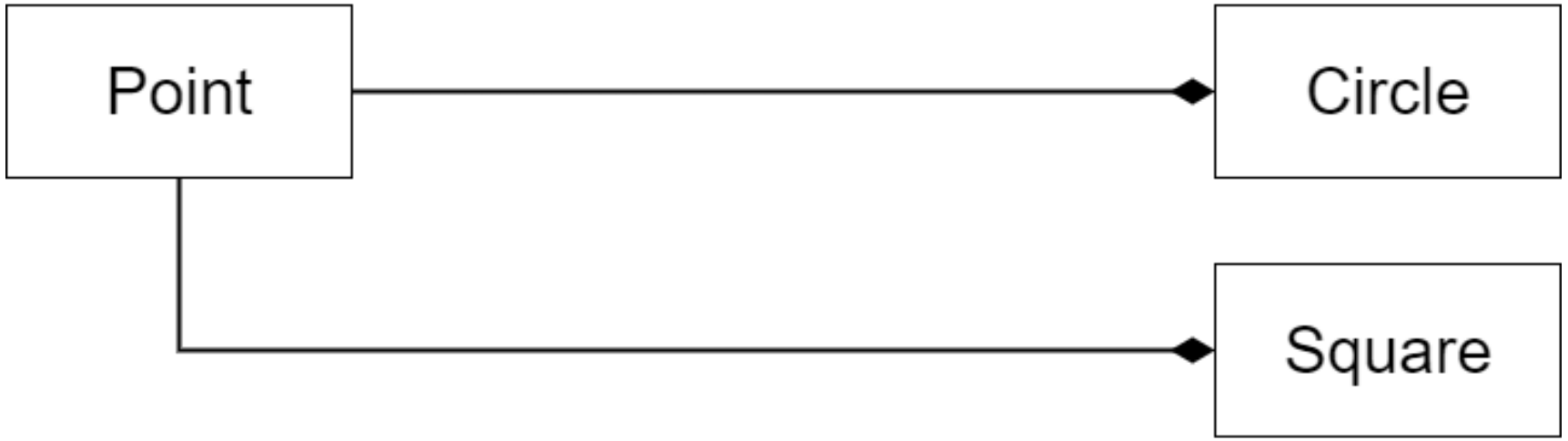
Пример агрегации

```
class Person {
    //...
};

class Group {
    Person *members;
public:
    void addMember(Person *member) {
        // Добавление участника
    }
    void removeMember(Person *member) {
        // Исключение участника
    }
};
```

Композиция

- Определяет отношение HAS A – то есть ношение владения
- Описывает целое и составные части, которые в него входят
- Конкретный экземпляр части может принадлежать только одному владельцу
- Целое управляет временем жизни входящих в него частей



Пример композиции

```
class Point {
    int x;
    int y;
public:
    Point(int x, int y): x(x), y(y) {}
};

class Circle {
    Point *center;
    int radius;
public:
    Circle(int x, int y, int radius)
        :center(new Point(x, y)), radius(radius) {};
    ~Circle() { delete center; }
};
```

Обобщение (наследование)

- Базовый принцип ООП
- Определяет отношение IS A - то есть "является"
- Позволяет дочернему получить функционал родительского

Реализация

- Наследование от интерфейса
- Интерфейс – класс, содержащий только чистые виртуальные методы и не обладающий состоянием