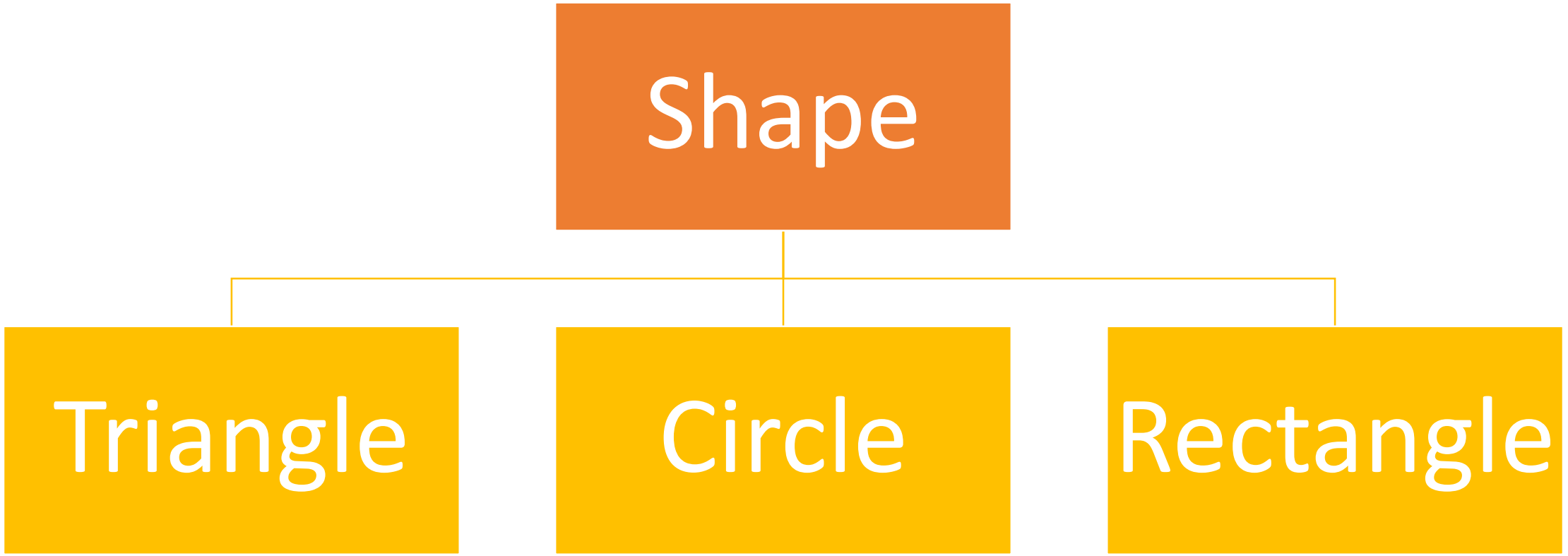
Основы Программирования

Лекция #6

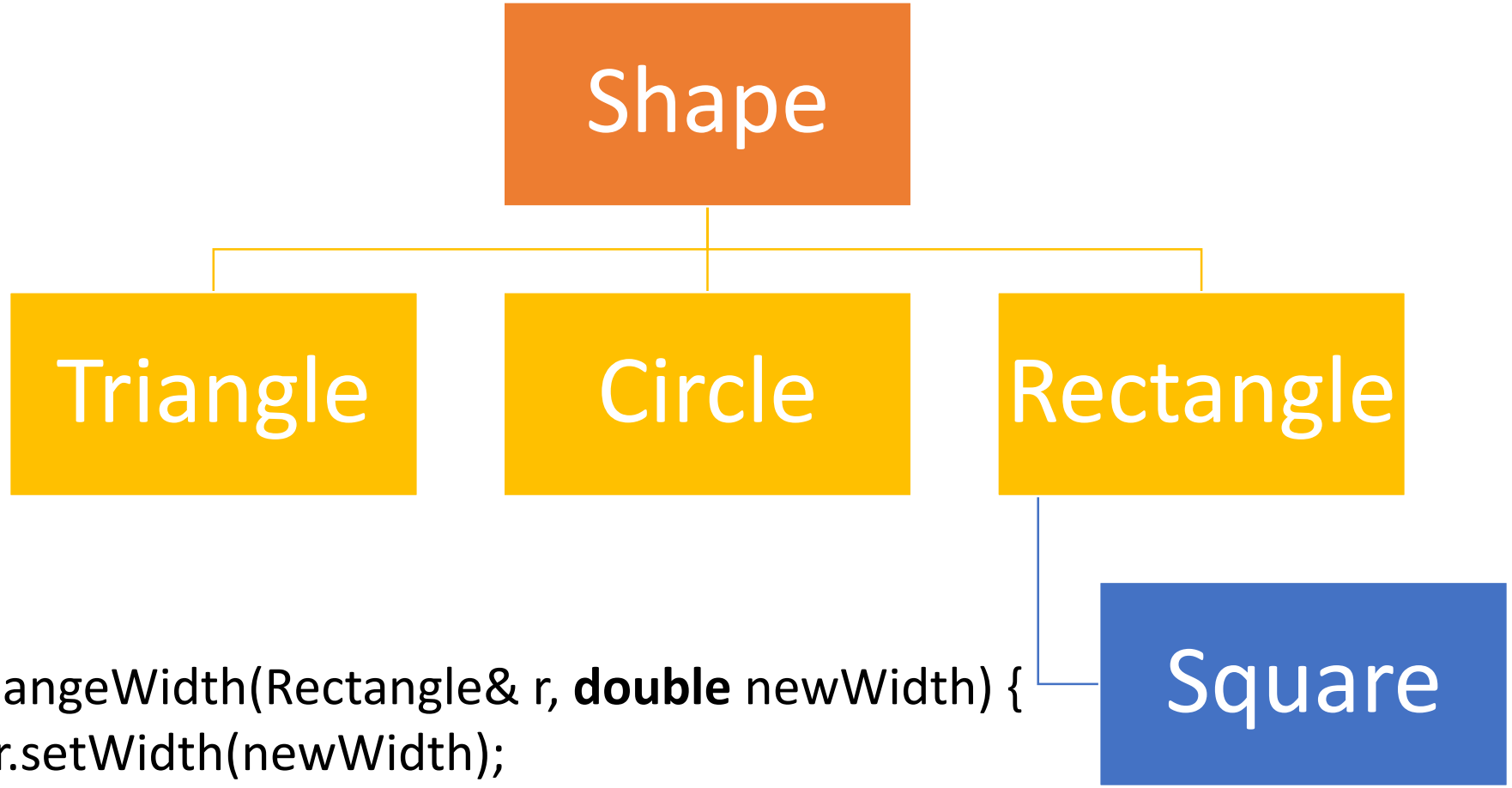
Темы

- Композиция/агрегация
- Операторы
- inline
- static
- friend

Построение иерархии (куда добавить Square?)

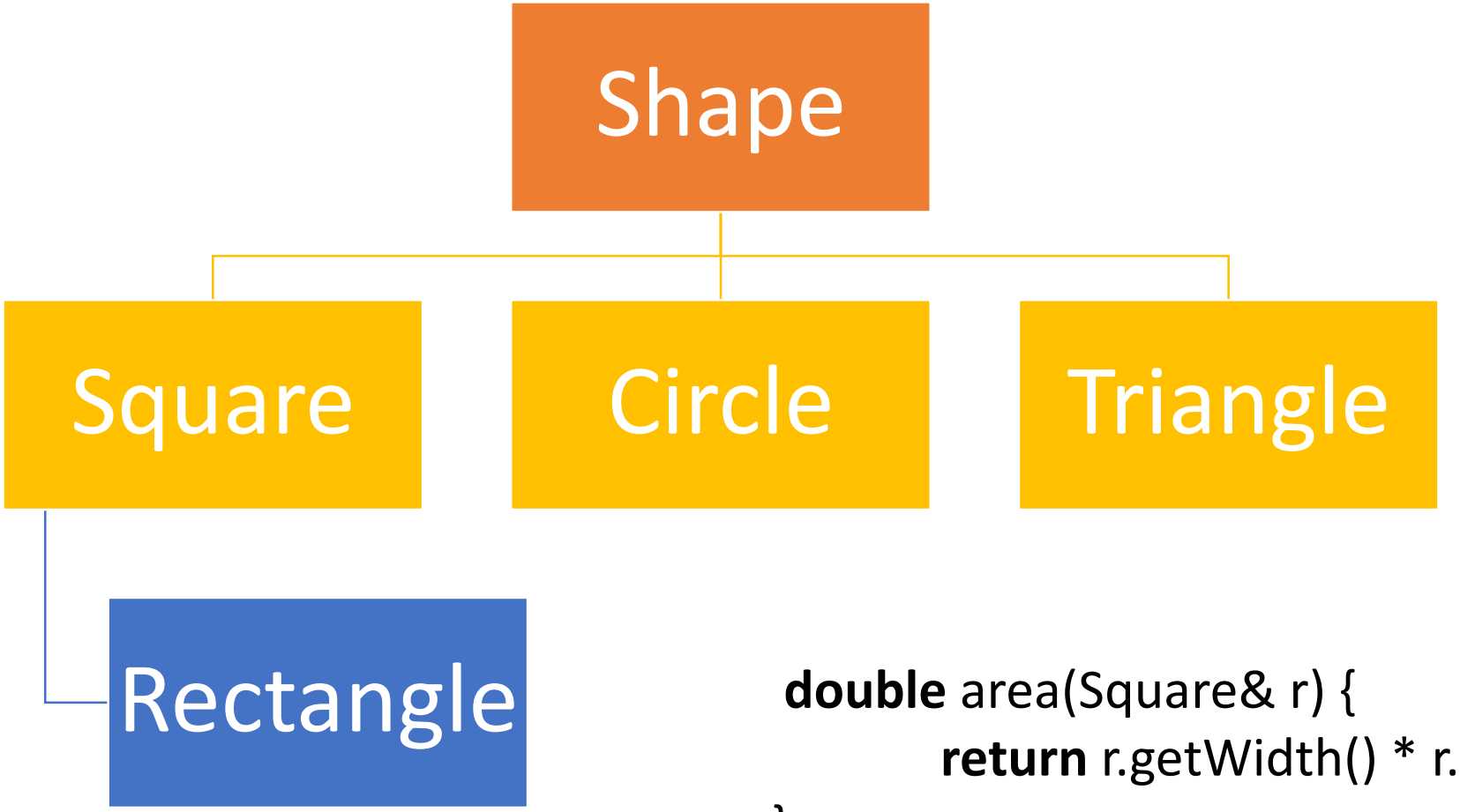


Построение иерархии (куда добавить Square?)



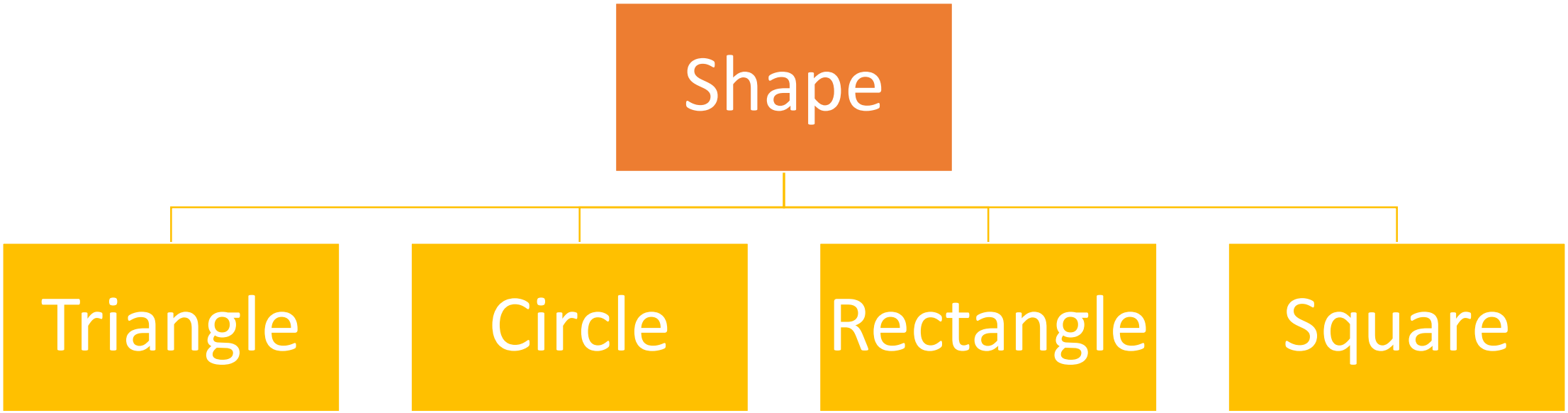

```
void changeWidth(Rectangle& r, double newWidth) {
        r.setWidth(newWidth);
}
```

Построение иерархии (куда добавить Square?)

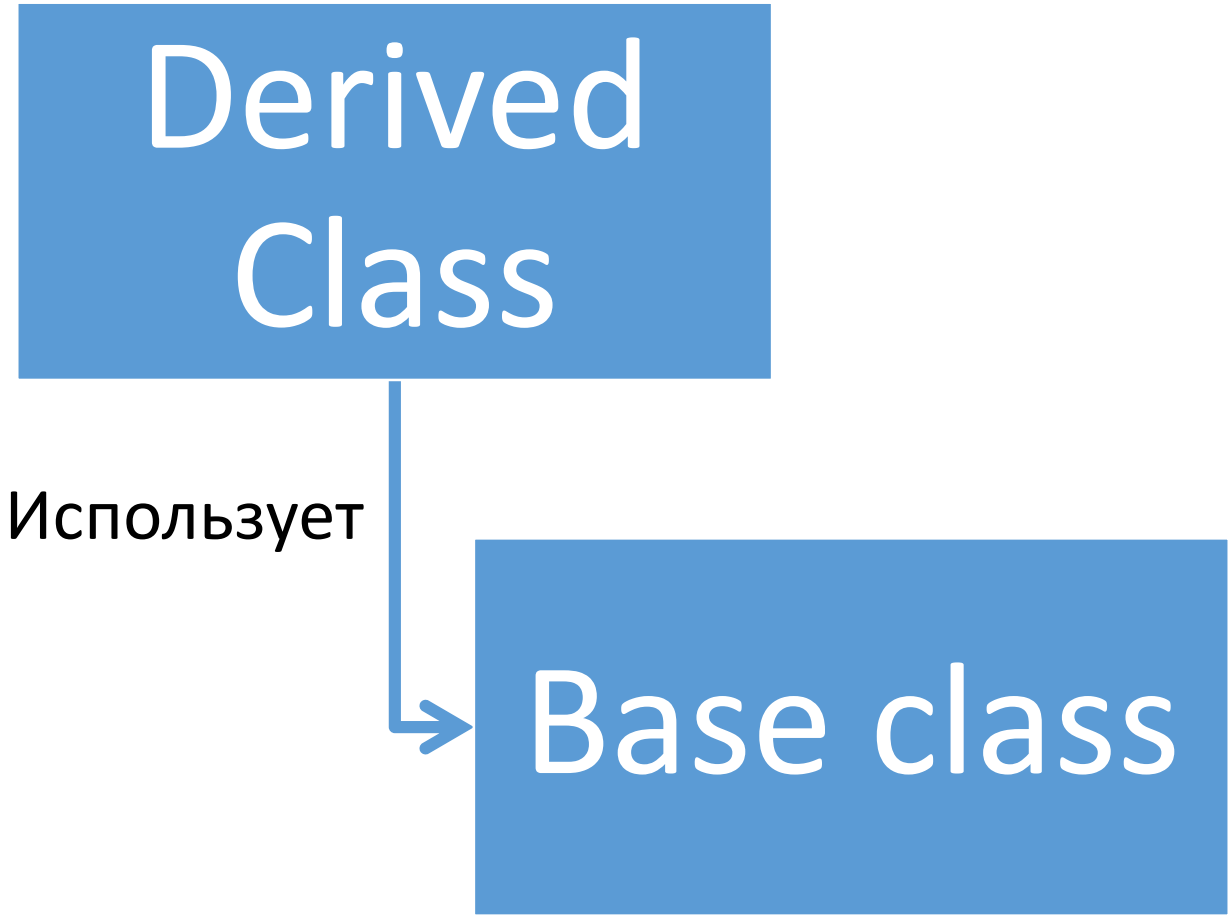



```
double area(Square& r) {
        return r.getWidth() * r.getWidth()
}
```

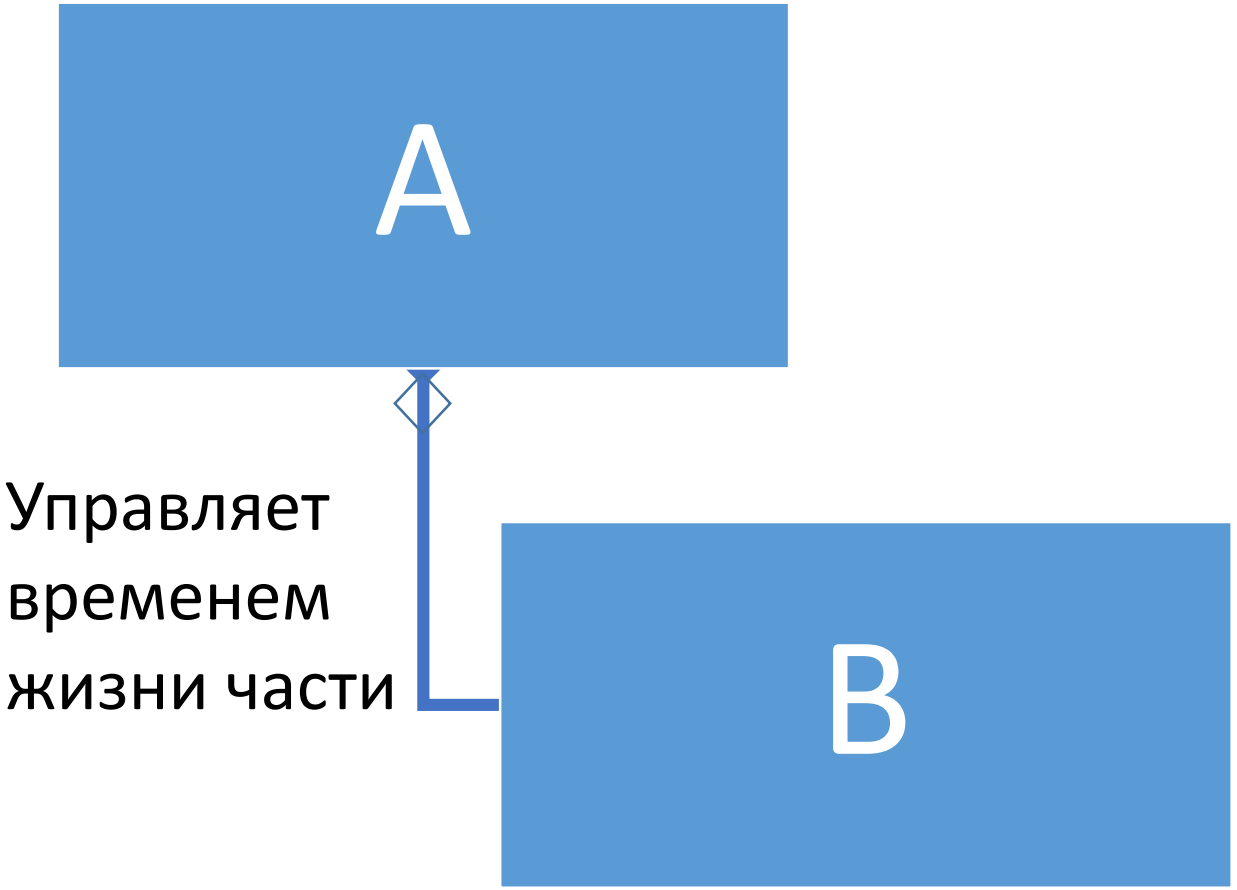
Построение иерархии



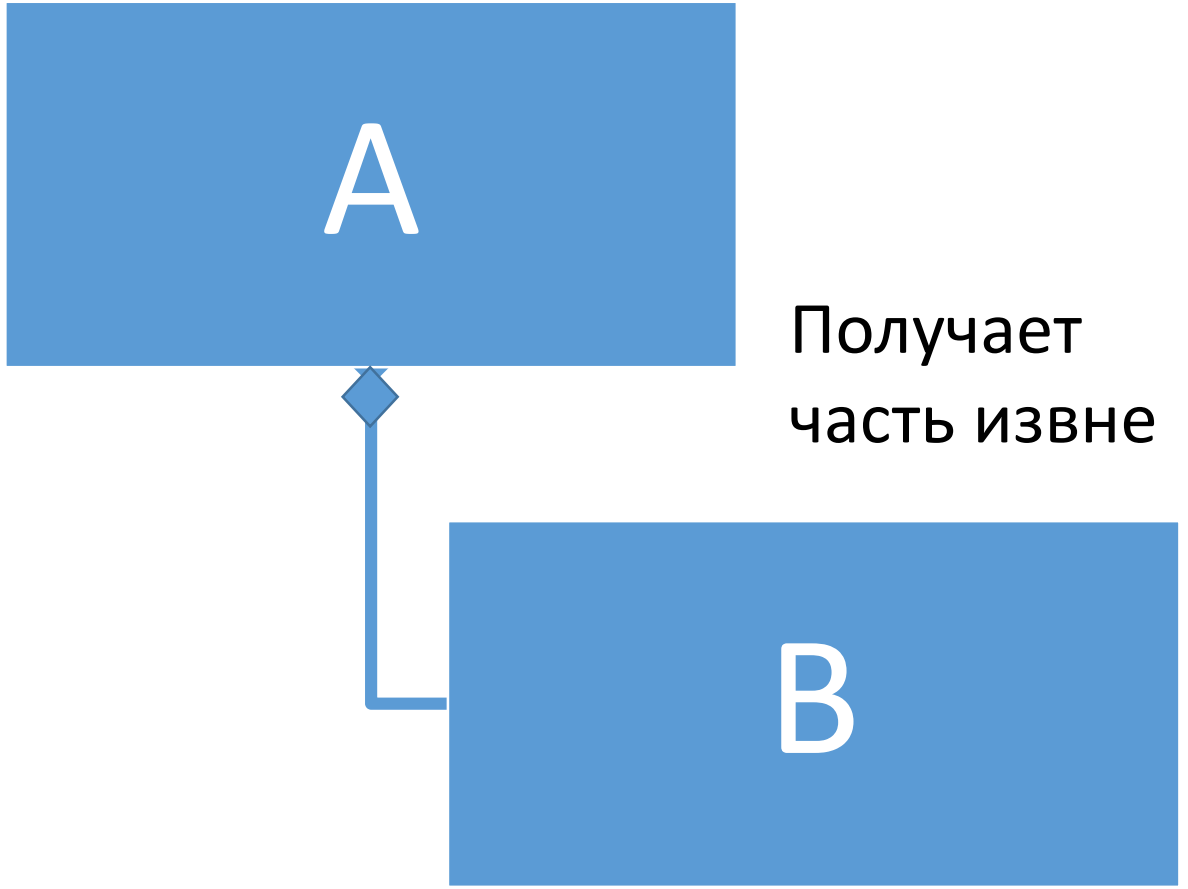
Наследование



Композиция



Агрегация



Композиция

```
class Composite {
        Dependent _dep;
public:


        void doSmth() {
                _dep.foo();
        }
};
```

Агрегация

```
class Aggregate {
        Dependent* _dep;
public:
        Aggregate(Dependent*
dep) : _dep(dep) { }
        void doSmth() {
                _dep.foo();
        }
};
```

Агрегирование и наследование

- **Агрегирование** – это включение объекта одного класса в качестве поля в другой
- Наследование устанавливает более сильные связи между классами, нежели агрегирование:
  - Приведение между объектами
  - Доступ к **protected** членам
- Если наследование можно заменить легко на агрегирование, то это нужно сделать

Операторы

- Арифметические:
  - Унарные: (префиксные/постфиксные) + - ++ --
  - Бинарные: + - * / % += -= /= %=
- Битовые:
  - Унарные: ~
  - Бинарные: & | ^ &= |= ^= >> <<
- Логические:
  - Унарные: !
  - Бинарные: & | && ||
  - Сравнения: == != > < >= <=

Операторы

- Присваивания: =
- Тернарный: ?:
- Специальные: , . ::
- Скобки: [] ()
- Оператор приведения: (type)
- Доступа: ->

- Операторы ?: . :: перегружать нельзя

Перегрузка операторов

```
Vector operator-(Vector const& v) {
        return Vector(-v.x, -v.y);
}
Vector operator+(Vector const& v, Vector const& w) {
        return Vector(v.x + w.x, v.y + w.y);
}
Vector operator*(Vector const& v, double d) {
        return Vector(v.x * d, v.y * d);
}
Vector operator*(double d, Vector const& v) {
        return v * d;
}
```

Перегрузка операторов внутри классов

```
struct Vector {
        Vector operator-() const { return Vector (-x, -y); }
        Vector operator-(Vector const& v) const {
                return Vector(x – v.x, y – v.y);
        }
        Vector& operator*=(double d) {
                x *= d;
                y *=d;
                return *this;
        }
```

Перегрузка операторов внутри классов

```
        double operator[] (int i) const {
                return i == 0 ? x : y;
        }
        bool operator()(double d) const    { /*… */ }
        void operator()(double a, double b)        { /*… */ }
private:
        double x, y;
};
```

Обязательна для (type) [] () = ->

Перегрузка инкремента/декремента

```
struct Counter {
        Counter& operator++() { // prefix
                //…
                return *this;
        }
        Counter operator++(int) { //postfix
                Counter temp(*this);
                ++(*this);
                return temp;
        }
};
```

Переопределение операторов ввода-вывода

```
#include <iostream>
 struct Vector { … };
 istream& operator>>(istream& is, Vector &p) {
        is >> p.x >> p.y;
        return is;
}
 ostream& operator<<(ostream& os, Vector const& p) {
        os << p.x << “ “ << p.y;
        return os;
}
```

Оператор приведения

```
struct Vector {
        operator double() const {
                return sqrt(x * x + y * y);
        }
        //…
};
```

Операторы с особым порядком вычисления

```
int main() {
        int a = 0;
        int b = 5;
        bool b1 = (a != 0) && (b = b / a);
        bool b2 = (a == 0) || (b = b / a);
        bool b3 = (a != 0), (b = b / ++a);
}
Sample operator&&(Sample const& a, Sample const& b) { … }
```

Операторы сравнения

```
bool operator==(Vector const& x, Vector const& b) { return … }
bool operator!=(Vector const& x, Vector const& b) { return ! (a == b); }
bool operator<(Vector const& x, Vector const& b) { return … }
bool operator>(Vector const& x, Vector const& b) { return b < a; }
bool operator<=(Vector const& x, Vector const& b) { return !(b < a); }
bool operator>=(Vector const& x, Vector const& b) { return !(a < b); }
```

Статические переменные

- Различают локальные и глобальные переменные
- Статическая локальная переменная – это глобальная переменная, доступная только в пределах функции.
- Время жизни – от первого вызова функции до конца программы

```
int next(int start = 0) {
        static int k = start;
        return k++;
}
```

Статические поля класса

- Статические поля класса – это глобальные переменные, определенные внутри класса. Для доступа не нужен объект

```
struct Sample {
        static int cout() { return _instances; }
private:
        static int _instances;
};
//….
cout << Sample::cout();
```

inline

- **Совет** компилятору встроить реализацию функции в местах вызова данной функции

```
inline double square(double x) { return x * x; }
```

- Все методы, определенные внутри класса, являются inline

Дружественные классы

```
struct A {
        friend struct B;
private:
        int _x;
        int _y;
};
struct B {
        void increase(A const& a, int d) { a._x += d; a._y += d; }
};
```

Дружественные функции

- Можно определять внутри описания класса (становятся **inline**).

```
struct A {
        friend ostream& operator<<(ostream& os, A const& a) {
                return os << a._x << “ “ << a._y;
        }
private:
        int _x, _y;
};
```

- Стоит избегать friend, так как данный подход нарушает инкапсуляцию