

# Множественное наследование

```
class A { ... };
```

```
class B { ... };
```

```
class C : public A, protected B { ... };
```

!!! Спецификатор доступа распространяется только на один базовый класс; для других базовых классов начинает действовать принцип умолчания.

!!! Класс не может появляться как непосредственно базовый дважды:

```
class C : public A, public A { ... }; - Er.!
```

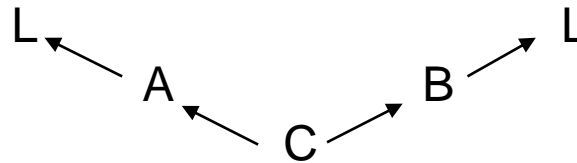
но может быть более одного раза непрямым базовым классом:

```
class L { public: int n; ... };
```

```
class A : public L { ... };
```

```
class B : public L { ... };
```

```
class C : public A, public B { ... void f (); ... };
```



A::L
Собственно А
B::L
Собственно В
Собственно С

Здесь **решетка смежности** такая:

При этом может возникнуть неоднозначность из-за «многократного» базового класса.

## О доступе к членам производного класса

```
void C::f () { ... n = 5; ...} // Er.! – неясно, чье n, но
```

```
void C::f () { ...A::n = 5; ...} // O.K.! , либо B::n = 5;
```

Имя класса в операции разрешения видимости (А или В) – это указание, в каком классе в решетке смежности искать заданное имя.

## О преобразовании указателей

Указатель на объект производного класса может быть неявно преобразован к указателю на объект базового класса, только если этот базовый класс является **однозначным** и **доступным** !!!

Продолжение предыдущего примера:

```
void g () {  
    C* pc = new C;  
    L* pl = pc;      // Er.! – L не является однозначным,  
    pl = (L*) pc;   // Er.! – явное преобразование не помогает,  
                    // но возможно:  
    pl = (L*) (A*) pc; // либо pl = (L*) (B*) pc; O.K.!
```

Базовый класс считается **доступным** в некоторой области видимости, если доступны его public-члены.

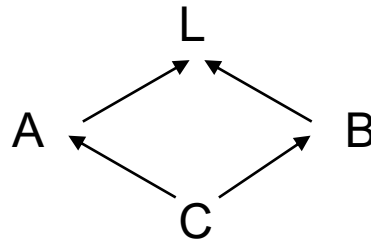
```
class B { public: int a; ... };  
class D : private B { ... };
```

```
void g () {  
    D* pd = new D;  
    B* pb = pd; // Er.! – в g() public-члены B, унаследованные  
                // D, недоступны, такое преобразование  
                // может осуществлять только  
                // функция-член D, либо друзья D.  
}
```

# Виртуальные базовые классы.

```
class L { public: int n ; ... };  
class A : virtual public L { ... };  
class B : virtual public L { ... };  
class C : public A, public B { ... void f (); ... };
```

Теперь решетка смежности будет такой:



и теперь допустимо:

```
void C :: f () { ... n = 5; ...} // О.К.! – n в одном экземпляре
```

```
void g () {  
    C* pc = new C;  
    L* pl = pc;      // О.К.! – появилась однозначность.  
}
```

# Правила выбора имен в производном классе.

- 1 шаг:** контроль **однозначности** (т.е. проверяется, определено ли анализируемое имя в одном базовом классе или в нескольких); при этом контекст не привлекается, совместное использование (в одном из базовых классов) допускается.
- 2 шаг:** если однозначно определенное имя есть имя перегруженной функции, то пытаются **разрешить** анализируемый вызов (т.е. найти best-matching).
- 3 шаг:** если предыдущие шаги завершились успешно, то проводится контроль **доступа**.

## Неоднозначность из-за совпадающих имен в различных базовых классах.

```
class A {  
    public:  
        int a;  
        void (*b) ();  
        void f ();  
        void g (); ...  
};
```

```
class B {  
        int a;  
        void b ();  
        void h (char);  
public:  
        void f ();  
        int g;  
        void h ();  
        void h (int); ...  
};
```

```
class C : public A, public B { ... };
```

## Пример.

```
void gg (C* pc) {  
    pc -> a = 1;    // Er.! – A::a или B::a  
    pc -> b();     // Er.! – нет однозначности  
    pc -> f ();    // Er.! – нет однозначности  
    pc -> g ();    // Er.! – нет однозначности,  
                  // контекст не привлекается!  
    pc -> g = 1;   // Er.! – нет однозначности,  
                  // контекст не привлекается!  
    pc -> h ();    // O.K.!  
    pc -> h (1);   // O.K.!  
    pc -> h ('a'); // Er.! – доступ в последнюю очередь  
    pc -> A::a = 1; // O.K.! – т.е. снимаем неоднозначность  
                  // с помощью операции «::»  
}
```

# Статические члены класса.

- Статические члены-данные и члены-функции описываются в классе с квалификатором **static**.
- Статические члены-данные существуют в одном экземпляре и доступны для всех объектов данного класса.
- Статические члены класса существуют **независимо от конкретных экземпляров класса**, поэтому обращаться к ним можно еще до размещения в памяти первого объекта этого класса, а также изменять, используя, например, имя константного объекта класса.
- **Необходимо** предусмотреть выделение памяти под каждый статический член-данные класса (т.е. описать его вне класса с возможной инициализацией), т.к. при описании самого класса или его экземпляров память под статические члены-данные не выделяется.
- Доступ к статическим членам класса (наряду с обычным способом) можно осуществлять через имя класса (без указания имени соответствующего экземпляра) и оператор разрешения области видимости «**::**».



# Пример.

```
class A {  
public:  
    static int x;  
    static void f (char c);  
};
```

**int** A::x; // !!! – размещение статического объекта в памяти

```
void g() {  
    ...  
    A::x = 10;  
    ...  
    A::f ('a');  
    ...  
}
```

# Особенности использования статических методов класса

- Статические методы класса используются, в основном, для работы с глобальными объектами или статическими полями данных соответствующего класса.
- Статические методы класса не могут пользоваться нестатическими членами класса.
- Статические методы класса не могут пользоваться указателем `this`, т.е. использовать объект, от имени которого происходит обращение к функции.
- Статические методы класса не могут быть виртуальными и константными (`inline` - могут).