Единичное наследование

Конструкторы, деструкторы и operator= не наследуются!!!

(***class*** | ***struct***) < имя ***derived-cl*** > ***:*** { ***public*** | ***protected*** | ***private*** } < имя ***base-cl*** > {...};

Для **классов** наследование по умолчанию — **private**

Для **структур** наследование по умолчанию — **public**

При создании объекта из ***derived-cl*** **сначала** вызывается **конструктор** ***base-cl*** , затем собственный конструктор.

При уничтожении объекта из ***derived-cl*** **сначала** вызывается **собственный деструктор**, затем деструктор ***base-cl*** .

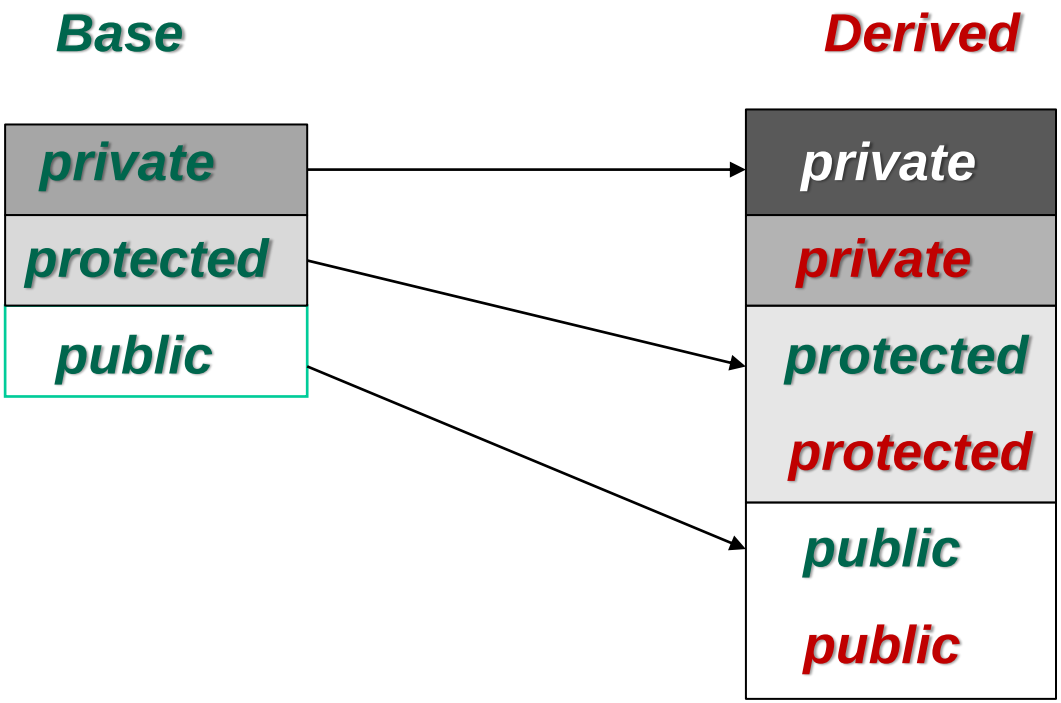
Использование **списка инициализации** – единственный способ вызвать конструктор ***base-cl*** **с параметрами** при создании объекта из ***derived-cl***.

Пример:

```
struct A  {   int  x;
                  A (int y) { x = y;  }  };
struct B : A  {  int z;
                     B (int n = 0) : A(n) { z = n; }  };
```

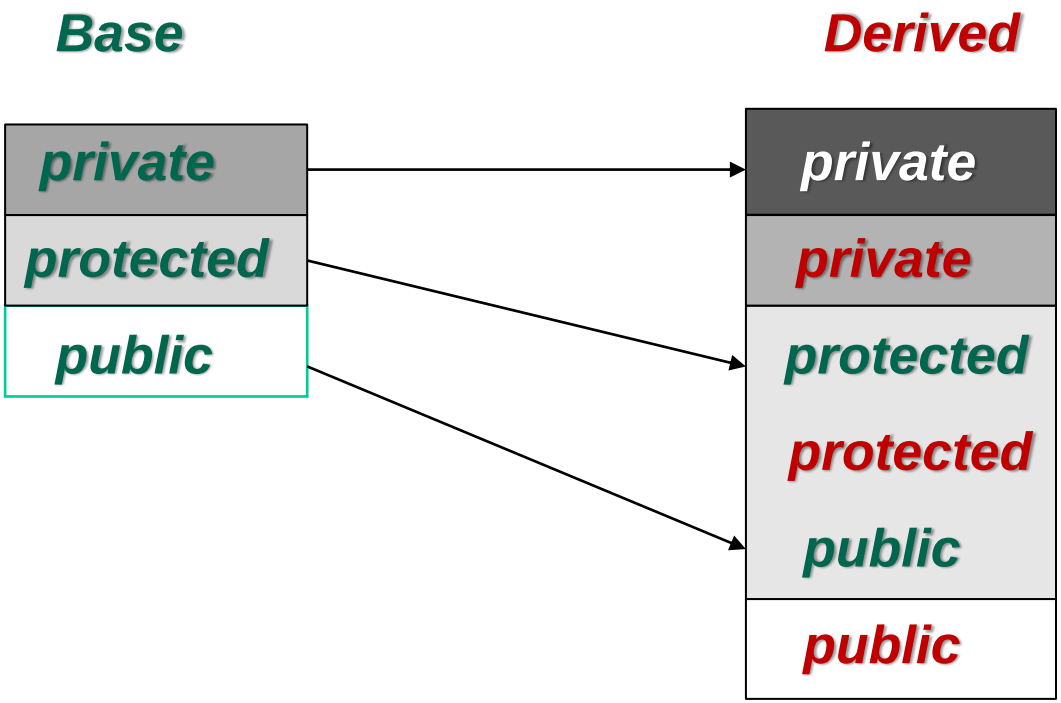
Единичное наследование. Public-наследование

class < имя ***derived***-cl > : **public** < имя ***base***-cl > {...};



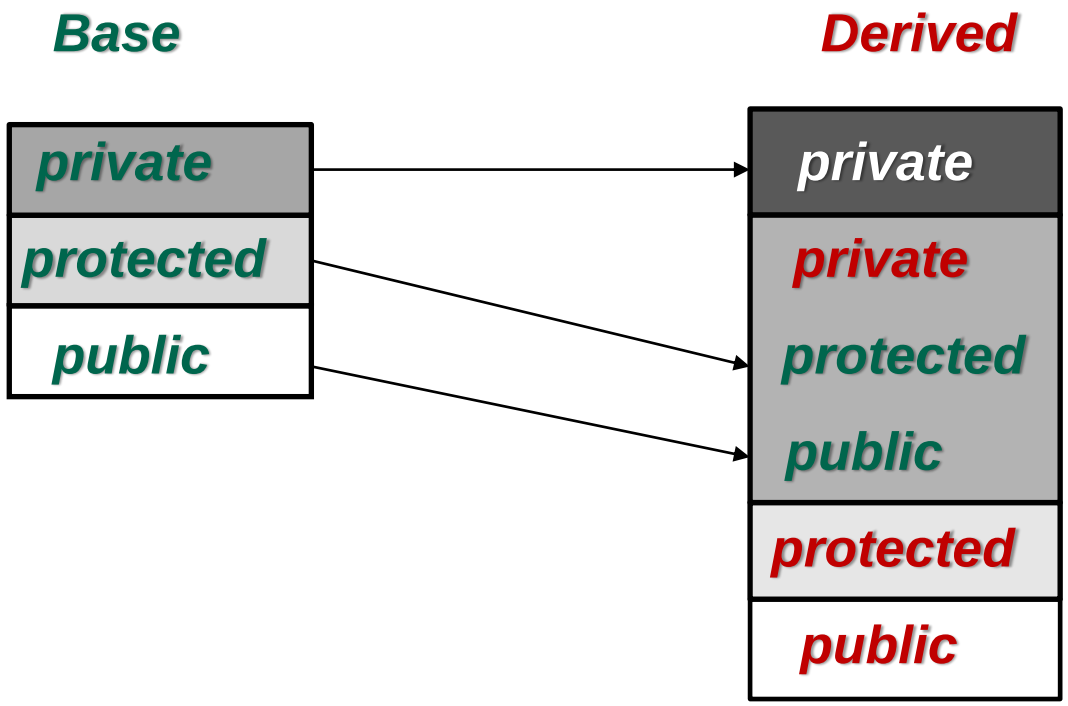
Единичное наследование. Protected-наследование

class < имя ***derived***-cl > : **protected** < имя ***base***-cl > {...};



Единичное наследование. Private-наследование

class < имя ***derived***-cl > : **private** < имя ***base***-cl > {...};



Единичное наследование. Пример.

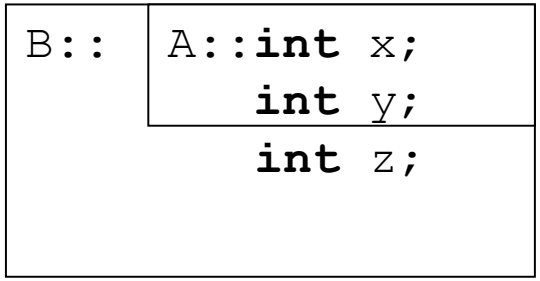
```
struct A  {  int  x;   int  y; };
struct B : A  { int z; };
class C : protected A { int z; };
```

.....

```
A a;
B b;
b.x = 1;
b.y = 2;
b.z = 3;
a = b;
b = a;
```

```
A * pa;
C   c,  * pc = &c;
pc -> z;   // ошибка: доступ к закрытому полю
pc -> x;   // ошибка: доступ к защищённому полю
pa = ( A * ) pc;
pa -> x;   // правильно: поле A::x – открытое
```

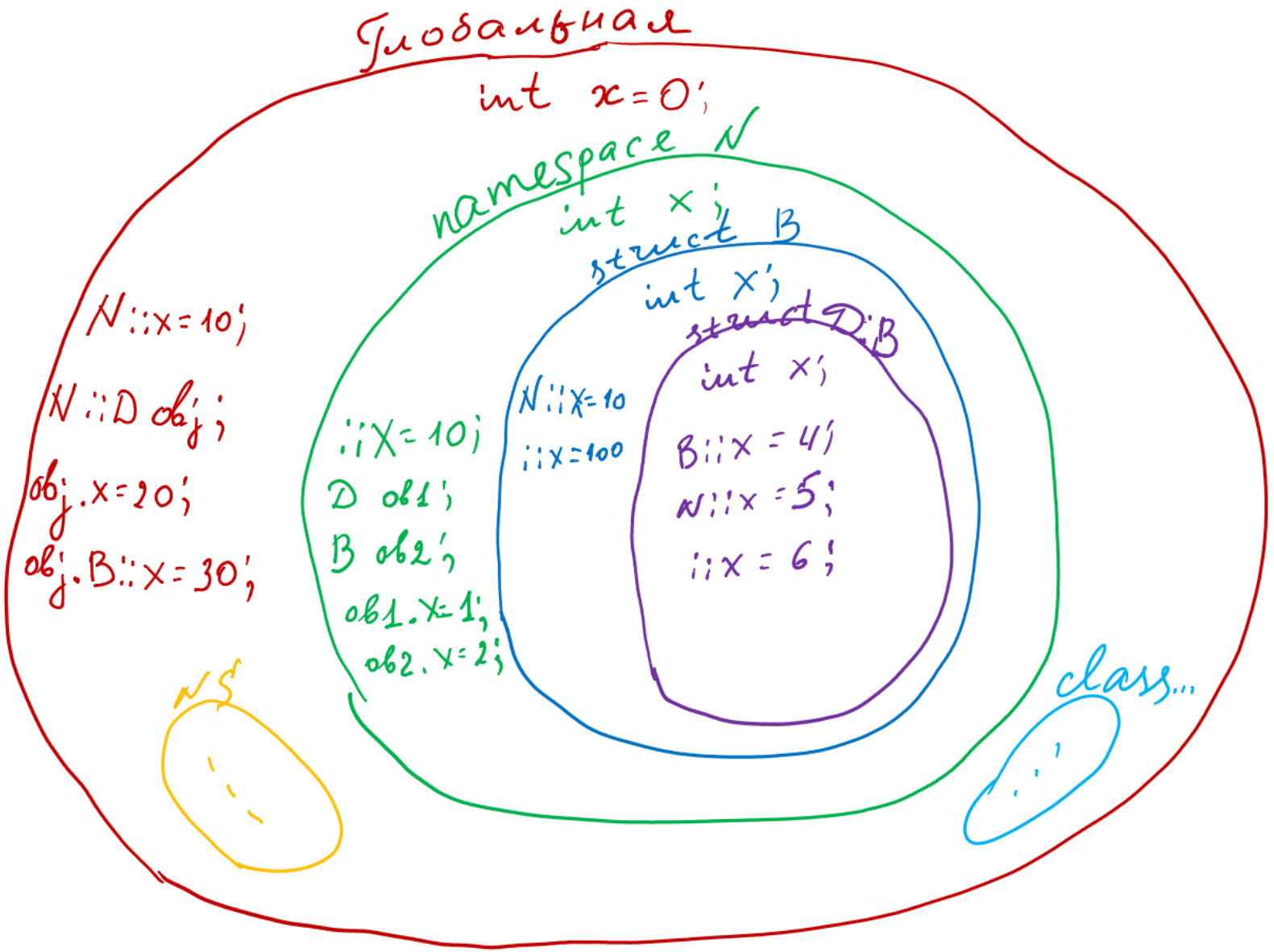
```
A a, *pa;
B b, *pb;
pb = &b;
pa = pb;
pb = pa;
pb = ( B* ) pa;
```



Сокрытие имён

```
struct A {
        int f ( int x , int y);
        int g ();
        int h;
};
struct B : public A {
        int x;
        void f ( int x );
        void h ( int x );
};
.......
A  a,  *pa;
B  b,  *pb;
pb = &b;
pb -> f (1);          // вызывается B::f(1)
pb -> g ();           // вызывается A::g()
pb -> h = 1;          // Err.! функция h(int) – не L-value выражение
pa = pb;
pa -> f (1);          // Err.! функция A::f(1) имеет 2 параметра
```

Области видимости



Области видимости. Пример.

```
int x = 0;
namespace N {
    int x = 1;
    struct B {
        int x = 2;
        void b();
    };
    struct D : B {
        int x = 3;
        void d(){
            cout << "d::x = " << x << endl;
            cout << "d: ::x = " << ::x << endl;
            cout << "d: B::x = " << B::x << endl;
            cout << "d: N::x = " << N::x << endl;
        }
    };
}
void N::B::b() {  D d1;   cout << d1.x << endl; }

int main() {        int x = 5;
                    N::D od;
                    cout << "main:   x = " << x << endl;
                    cout << "main:   ::x = " << ::x << endl;
                    cout << "main:   B::x = " << od.B::x << endl;
                    cout << "main:   D::x = " << od.x << endl;
                    cout << "main:   N::x = " << N::x << endl;
                    return 0; }
```

Пространства имен с совпадающими именами и объявление using. Пример.

```
void f() { cout << "::f()\n"; }
int x = 2, y = 22, z = 222;
namespace NS1 { int x = 1, y = 11, z = 111;
                void f(int i){ cout << "NS1::f(int)\n";} }
namespace NS2 { int x = 3, y = 33, z = 333;
                void f(double i){ cout << "NS2::f(double)\n";} }
using namespace NS1;
using namespace NS2;
int main() {
   using ::x; // без этого объявления при использовании x – Err!
   using NS1::y; // без этого объявления при использовании y – Err!
   using NS2::z; // без этого объявления при использовании z – Err!
   f(3);        //  NS1::f(int)
   cout << x << " " << y << "  " << z << endl; //   2 11 333
}
```

Видимость и доступность имен. Пример.

```
int x;
void f (int a) { cout << " :: f " << a << endl; }

class A {
    int x;
public:
    void f (int a) { cout << " A:: f " << a << endl; }
};
class B : public A {
public:
    void f (int a) { cout << " B:: f " << a << endl; }
    void g ();
};

void B::g() {
    f(1);           // вызов  B::f(1)
    A::f(1);
    ::f(1);         // вызов глобальной void f(int)
    //x = 2;        // ошибка!!! – осущ. доступ к закрытому члену класса A
}
```

Вызов конструкторов базового и производного классов. Пример.

```
class A { ... };

class B : public A {
public:
    B ();
    B (const B &); // есть явно описанный конструктор копирования
    ...
};
class C : public A {
public:
    // нет явно описанного конструктора копирования
    ...
};
int   main ( ) {
    B b1;            //  A (),   B ()
    B b2 = b1;       //  A (),   B (const B &)
    C c1;            //  A (),   C ()
    C c2 = c1;       //  A (const A &),   C (const C &)
    ...
}
```

Классы student и student5

```
...
class student {
     char * name;
     int year;
     double est;
public:
     student ( char* n, int y, double e);
     void print () const;
      ~student ();
};
```

```
class student5: public student {
     char * diplom;
     char * tutor;
public:
     student5 ( char* n, double e, char* d, char* t);
     void print () const;
    // эта print  скрывает print из базового класса
       ~student5 ();
};
```

```
student5:: student5 ( char* n, double e, char* d, char* t) : student (n, 5, e) {
     diplom = new char [strlen (d) + 1];
     strcpy (diplom, d);
     tutor = new char [strlen (t) + 1];
     strcpy (tutor, t);
}
```

```
student5 :: ~student5 () {

     delete [ ] diplom;
     delete [ ] tutor;
}
```

```
void  student5 :: print () const {
        student :: print ();    // name, year, est
        cout << diplom << endl;
        cout << tutor << endl:
}
```

Использование классов student и student5

```
void f ( ) {
    student  s ("Kate", 2, 4.18),    * ps = & s;
    student5  gs ("Moris", 3.96, "DIP", "Nick"), * pgs = & gs;

    ps -> print();              // student :: print ();
    pgs -> print();             // student5 :: print ();

    ps = pgs;      // base = derived – допустимо с преобразованием по
                   // умолчанию.
    ps -> print(); // student :: print () – функция выбирается статически
                   // по типу указателя.
 }
```

Виртуальные методы

Метод называется **виртуальным**, если при его объявлении в классе используется квалификатор **virtual**.

Класс называется **полиморфным**, если содержит хотя бы один виртуальный метод.

Объект полиморфного класса называют **полиморфным** объектом.

Чтобы динамически выбирать функцию print () по типу объекта (**реализовать динамический полиморфизм**), на который ссылается указатель, переделаем наши классы т.о.:

```
class student {… public:
                                ...
                               virtual void print ( ) const ;
 };
class student5 : public student {…  public:
                                                    ...
                                                  [virtual] void print ( ) const ;
};
```

Тогда: ps = pgs;

ps -> print(); // student5 :: print () – ф-я выбирается динамически по типу

// объекта, **чей адрес в данный момент хранится в указателе**

Виртуальные деструкторы

Совет: для полиморфных классов делайте деструкторы виртуальными!!!

```
void f () {
    student * ps = new  student5 ("Moris", 3.96, "DIP", "Nick");
            …
    delete ps; // вызовется ~student, и не вся память зачистится
}
```

Но если:

```
    virtual ~student ();    и
    [virtual] ~student5 ();
```

то вызовется ~student5(), т.к. сработает динамический полиморфизм.

Механизм виртуальных функций (механизм динамического полиморфизма)

1. !Виртуальность функции, описанной с использованием служебного слова **virtual** не работает сама по себе, она начинает работать, когда появляется класс, производный от данного, с функцией с **таким же прототипом**.
2. Виртуальные функции выбираются по типу объекта, на который ссылается указатель (или ссылка) на базовый класс.
3. У виртуальных функций должны быть одинаковые прототипы. Исключение составляют функции с одинаковым именем и списком формальных параметров, у которых тип результата есть указатель или ссылка на себя (т.е. соответственно на базовый и производный класс).
4. Если виртуальные функции отличаются только типом результата (кроме случая выше), генерируется ошибка.
5. Для виртуальных функций, описанных с использованием служебного слова **virtual,** с разными прототипами работает механизм сокрытия имен.

Абстрактные классы

Абстрактный класс содержит хотя бы одну **чистую виртуальную** функцию.

Чистая виртуальная функция: **virtual** тип_рез имя (сп_фп) **= 0**;

Пример:

```
class shape {
public:
    virtual double area () = 0;
};
class rectangle: public shape {
    double height, width;
public:
    double area () {
        return   height * width;
    }
};
class circle: public shape {
    double radius;
public:
    double area () {
        return  3.14 * radius * radius;
    }
};
```

```
#define N 100
....
shape* p [ N ];
double total_area = 0;
 ....
for (int i =0; i < N; i++)
total_area += p[i] -> area();
        ....
```

Интерфейсы

Интерфейсами называют абстрактные классы, которые:

- не содержат нестатических полей-данных, и
- все их методы являются открытыми чистыми виртуальными функциями.

Реализация виртуальных функций

- Для **каждого полиморфного класса** создается таблица указателей на его виртуальные методы — **vtbl** — массив, индекс элемента которого соответствует прототипу виртуальной функции класса, а содержимое — ее адресу.

- В каждый **объект** полиморфного класса в public-область неявно помещается указатель на его **vtbl — vtbl* pvtbl;**

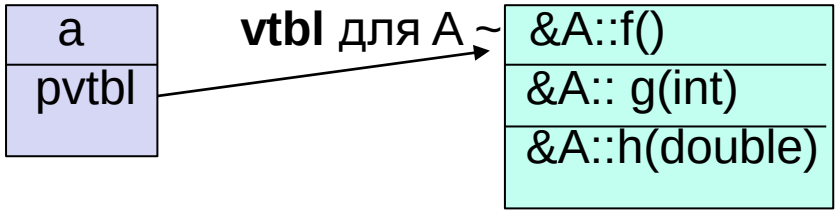
- При вызове виртуального метода используется механизм косвенных вызовов: **this** — адрес объекта, по значению поля **pvtbl** по этому адресу находится таблица **vtbl**, по прототипу выбранной функции определяется индекс элемента таблицы, а затем по адресу, указанному в таблице вызывается функция.

- Для **производного** полиморфного класса **vtbl** строится по таблице базового класса, в которой производится **замещение** адресов тех виртуальных функций, прототипы которых совпали с прототипами виртуальных функций базового класса. Адреса новых виртуальных функций производного класса добавляются в конец его **vtbl.**

Реализация виртуальных функций. Пример.

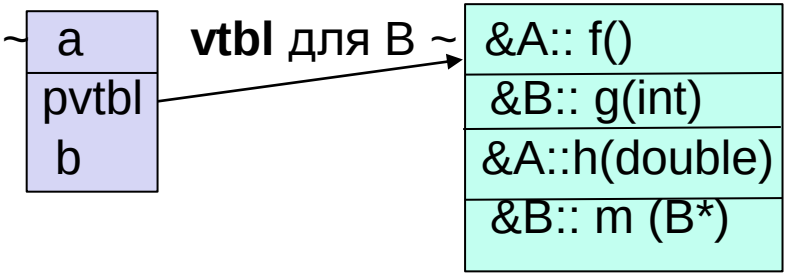
```
class A {   int a;
public:
     virtual void f ();
     virtual void g (int);
     virtual void h (double);
};
```

A a; ~



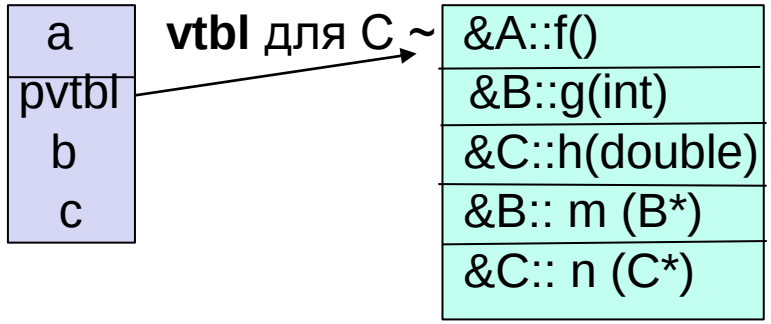

```
class B : public A {
public:
     int b;
     void g (int);
     virtual void m (B*);
};
```

B b; ~




```
class C : public B {
public:
        int c;
        void h (double);
        virtual void n (C*);
};
```

C c; ~




```
C c;
A *p = &c;
p -> g (2);   ~   (* ( p -> pvtbl [1]) ) (p, 2); // p = this
```

Виртуальные функции. Пример 1.

```
class X {
public:
    void g ( ) {
        cout << "X::g\n";
        h ( );
    }
    virtual void f() {
        g ( );
        h ( );
    }
    virtual void h ( ) {
        cout << "X::h\n";
    }
};
```

```
class Y : public X {
public:
    void g ( ) {
        cout << "Y::g\n";
        h ( );
    }
     virtual void f ( ) {
        g ( );
        h ( );
    }
    virtual void h ( ) {
         cout << "Y::h\n";
    }
};
```

```
int main () {
        Y b;
        X  *px = &b;
        px -> f();          //  Y::g    Y::h    Y::h
        px -> g();          //  X::g    Y::h
        return 0;
}
```

Виртуальные функции. Пример 2.

```
struct A {
    virtual int f (int x, int y) {
        cout << "A : :f (int, int) \n";
        return x + y;
    }
    virtual void f ( int x ) {
        cout << "A :: f( ) \n";
    }
};

struct B :  A {
    void f ( int x ) {
        cout << "B :: f( ) \n";
    }
};

struct C :  B {
        virtual int f (int x, int y) {
        cout << "C :: f (int, int) \n";
        return x + y;
    }
};
```

```
int main () {
        B b,  *pb = &b;
        C c;
        A  * pa = &b;
        pa -> f (1);      // B::f(int);
        pa -> f (1, 2);  // A::f(int, int)
    //pb -> f (1, 2);   // Err.! Эта f не видна

        A & ra = c;
        ra.f(1,1);        // C::f(int, int)

        B & rb = c;
    //rb.f(0,0);   // Err.! Эта f не видна
         return 0;
}
```

Виртуальные функции. Пример 3.

```
struct B {
    virtual B& f () { cout << " f ( ) from B\n"; return *this;}
    virtual void g (int x, int y = 7) { cout << "B::g\n"; }
};

struct D : B {
    virtual D& f ( ) { cout << " f ( ) from D\n"; return *this; }
    virtual void g (int x, int y) { cout << "D::g y = " << y << endl; }
};
    int main () {
        D  d;
        B  b1,  *pb = &d;
        pb -> f();                  // f ( ) from D
        pb -> g(1);                 // D::g  y = 7
        pb -> g(1,2);               // D::g  y = 2
        return 0;
    }
```

Если значение параметра y по умолчанию будет в функции g класса D, а в B не будет, компилятор на вызов pb -> g(1) выдаст ошибку.