

# Перегрузка функций

О перегрузке можно говорить только для функций из одной области описания

Перегрузка:

```
struct A {  
    int f (int x);    // две функции с одинаковым именем в одной области  
    int f (char x); // описания;    обращение f(1) вызовет f(int)  
                    //                обращение f('a') вызовет f(char)  
}
```

Перекрытие:

```
namespace N {  
    int f (int x) {...};  
    namespace M {  
        int f (char x) {...};  
        // обращение f ('a') вызовет f(char); f (int) – не видна
```

## Алгоритм поиска оптимально отождествляемой функции

1. Выбираются только те перегруженные (одноименные) функции, для которых фактические параметры соответствуют формальным по количеству и типу (приводятся с помощью каких-либо преобразований).
2. Для каждого параметра функции (отдельно и по очереди) строится множество функций, оптимально отождествляемых по этому параметру (best matching).
3. Находится пересечение этих множеств:
  - если в нем содержится ровно одна функция – она и является искомой,
  - если множество пусто или содержит более одной функции, генерируется сообщение об ошибке.

## Пример 1.

```
class X { public: X(int);...};
```

```
class Y {<нет конструктора с параметром типа int>...};
```

```
void f (X, int);           // 1 пар. - 'да'           2 пар. - 'да'
```

```
void f (X, double);       // 1 пар. - 'да'           2 пар. - 'нет'
```

```
void f (Y, double);       //отбрасывается на 1-м шаге
```

```
void g () {... f (1,1); ...}
```

Т.к. в пересечении множеств, построенных для каждого параметра, одна функция  $f(X, int)$  – вызов разрешим.

## Пример 2.

```
struct X { X (int);...};
```

```
void f (X, int);    // 1 пар. - 'нет'  2 пар. - 'да'
```

```
void f (int, X);    // 1 пар. - 'да'  2 пар. - 'нет'
```

```
void g () {... f (1,1); ...}
```

Т.к. пересечение множеств, построенных для каждого параметра, пусто – вызов неразрешим.

### Пример 3.

```
void f (char);
```

```
void f (double);
```

```
void g () {... f (1); ...} // ?
```

Не всегда просто выполнить шаг 2 алгоритма, поэтому стандартом языка C++ закреплены правила сопоставления формальных и фактических параметров для выбора одной из перегруженных функций.

## **Правила для шага 2 алгоритма выбора перегруженной функции**

- а) Точное отождествление
- б) Отождествление при помощи расширений
- в) Отождествление с помощью стандартных преобразований
- г) Отождествление с помощью преобразований, определенных пользователем
- д) Отождествление по ... (по многоточию)

## **Правила для шага 2 алгоритма выбора перегруженной функции**

Шаги (а), (б), (в), (г), (д) выполняются по очереди. Очередной шаг выполняется только если на предыдущих шагах не была найдена наиболее подходящая функция. Полная картина возможных комбинаций шагов будет дана далее.

## а) Точное отождествление

- точное совпадение,
- совпадение с точностью до typedef,
- тривиальные преобразования:

$T[] \leftrightarrow T^*$ ,  
 $T \leftrightarrow T\&$ ,  
 $T \rightarrow \text{const } T$ , // в одну сторону!  
 $T(\dots) \leftrightarrow (T^*)(\dots)$ .

**Пример** (точное совпадение):

<code>void f (float);</code>	<code>void g () {...</code>	<code>f (1.0);</code>	<code>// f (double)</code>	
<code>void f (double);</code>		<code>f (1.0F);</code>	<code>// f (float)</code>	
<code>void f (int);</code>		<code>f (1);</code>	<code>// f (int);</code>	<code>...</code>
	<code>}</code>			

## б) Отождествление при помощи расширений

- Целочисленные расширения:

char, short (signed и unsigned), enum, bool --> int ( unsigned int, если не все значения могут быть представлены типом int – тип unsigned short не всегда помещается в int);

- Вещественное расширение: float --> double

### Пример:

```
void f (int);  
  
void f (double);  
  
void g () {  
    short aa = 1;  
    float ff = 1.0;  
    f (ff);           // f (double)  
    f (aa);          // f (int)  
}
```

Неоднозначности нет, хотя short приводится к int и к double, float приводится к int и к double.

## в) Отождествление с помощью стандартных преобразований

- Все оставшиеся стандартные целочисленные и вещественные преобразования, которые могут выполняться неявно.
- Преобразования указателей:
  - 0 --> любой указатель,
  - любой указатель --> void\*,
  - derived\* --> base\* - для однозначно доступного базового класса;

### Пример:

```
void f (char);  
void f (double);
```

```
void g () { ... f (0); // неоднозначность, т.к.  
                // преобр. int --> char и  
                // int --> double равноправны  
}
```

## г) Отождествление с помощью пользовательских преобразований

- С помощью конструкторов преобразования

- С помощью функций преобразования

### Пример:

```
struct S {  
    S (long);           // long --> S  
    operator int ();   // S --> int    ...  
};
```

```
void f (long);           void g (S);           void h (const S&);  
void f (char*);         void g (char*);        void h (char*);
```

```
void ex (S &a) {  
    f (a); // O.K. f ( (long) ( a.operator int()) ); т.е. f (long) - на шаге г).  
    g (1); // O.K. g ( S ( (long) 1 ) ); т.е. g (S) - на шаге г).  
    g (0); // O.K. g ( (char*) 0); т.е. g (char*) - на шаге в)!!!  
    h (1); // O.K. h ( S ( (long) 1 ) ); т.е. h (const S&) - на шаге г).  
}
```

## Замечание 1

Пользовательские преобразования применяются **неявно** только в том случае, если они однозначны

**Пример:**

```
class Boolean {
    int b;
public:
    Boolean operator+ (Boolean);
    Boolean (int i) { b = i != 0;}
    operator int () { return b; }
...
};
void g () {
    Boolean b (1), c (0); // O.K.
    int k;
    c = b + 1; // ошибка!    т.к. может интерпретироваться двояко:
                // b.operator int () +1 – целочисленный '+' или
                // b.operator+ (Boolean (1)) – Boolean '+'
    k = b + 1; // ошибка!    -- “ --
}
```

## Замечание 2

Допускается не более **одного пользовательского** преобразования для обработки одного вызова для одного параметра

### Пример:

```
class X { ... public: operator int (); ... };  
class Y { ... public: operator X (); ... };
```

```
void f () {  
    Y a;  
    int b;  
    b = a; // ошибка! , т.к. требуется a.operator X ().operator int ()  
    ...  
}
```

Но! **явно** можно делать любые преобразования, явное преобразование сильнее неявного.

## д) Отождествление по ... .

### Пример1:

```
class Real {
    public:
        Real (double);
        ...
};

void f (int, Real);
void f (int, ...);           // можно и без ','

void g () {
    f (1,1);                 // O.K. f (int, Real);
    f (1, "Anna");          // O.K. f (int, ...);
}
```

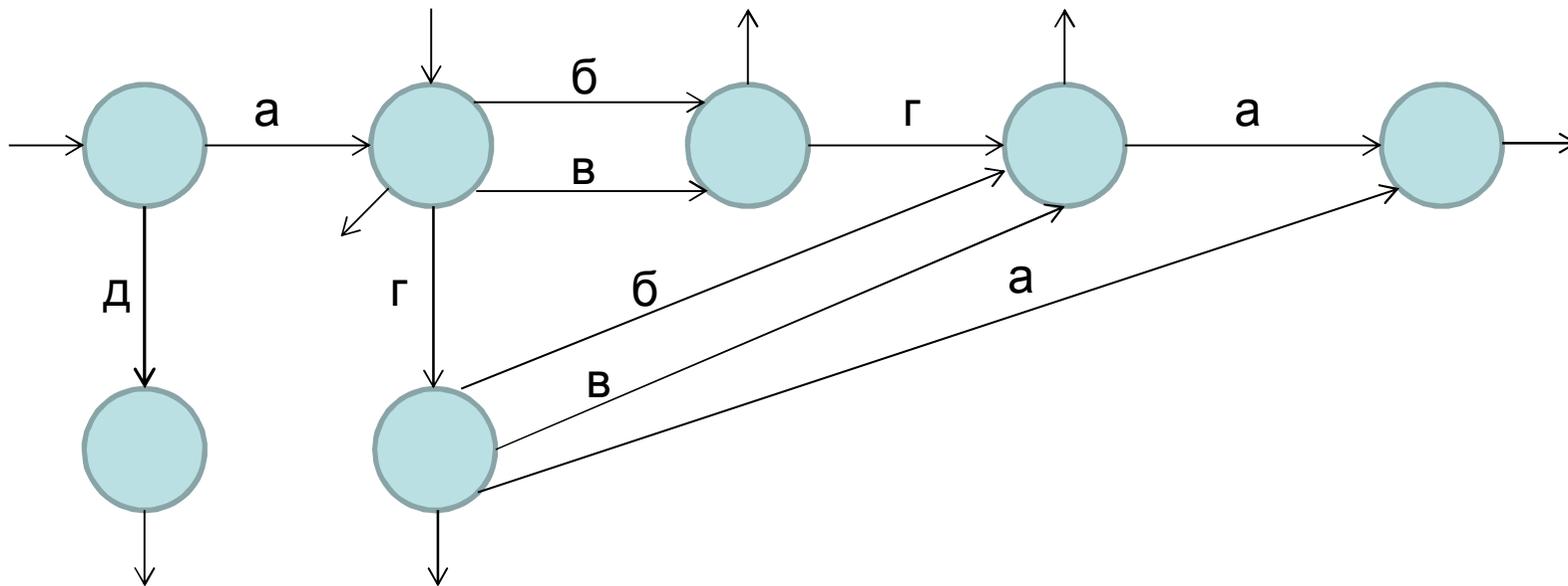
## Пример2:

Многоточие может приводить к неоднозначности:

```
void f (int);  
void f (int ...);
```

```
void g () {...  
    f (1); // ошибка! т.к. отождествление по  
           // первому параметру дает  
           // обе функции.  
}
```

# Допустимые цепочки преобразований



```
class C {  
    public: C (double); ...  
};  
int i=1; int & ri=i;  
f (C &) {...};  
f(ri); // а → в → г → а
```

# Наследование

Синтаксис описания производного класса:

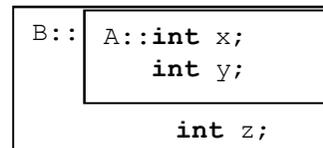
```
class < имя производного класса > :  
    < способ наследования > < имя базового класса > {...};
```

Пример:

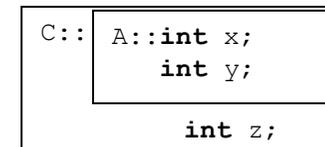
```
struct A {  
    int x;  
    int y;  
};
```

.....

```
struct B: A {  
    int z;  
};
```



```
class C: protected A {  
    int z;  
};
```



**Конструкторы, деструкторы и operator= не наследуются**

# Наследование

Синтаксис описания производного класса:

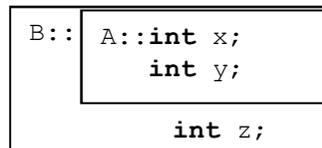
```
class < имя производного класса > :  
    < способ наследования > < имя базового класса > {...};
```

Пример:

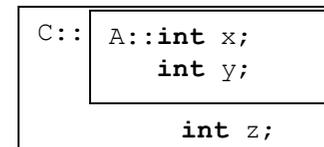
```
struct A {  
    int x;  
    int y;  
};
```

.....

```
struct B: A {  
    int z;  
};
```



```
class C: protected A {  
    int z;  
};
```



Способы наследования : **private** , **protected**, **public**

## Доступ в классе-наследнике в зависимости от способа наследования и доступа в базовом классе

Способ наследов. Доступ в базовом классе	private	protected	public
private	не доступен	не доступен	не доступен
protected	private	protected	protected
public	private	protected	public

# Наследование

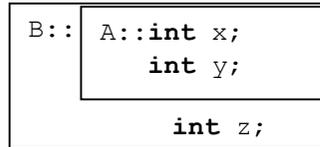
Пример:

```
struct A {  
    int x;  
    int y;  
};
```

.....

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

```
struct B: A {  
    int z;  
};
```



```
A * pa;
```

```
C c, * pc = &c;
```

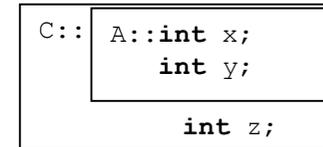
```
pc -> z; // ошибка: доступ к закрытому полю
```

```
pc -> x; // ошибка: доступ к защищённому полю извне класса
```

```
pa = ( A * ) pc;
```

```
pa -> x; // правильно: поле A::x – открытое
```

```
class C: protected A {  
    int z;  
};
```



---

```
A a, *pa;
```

```
B b, *pb;
```

```
pb = &b;
```

```
pa = pb; // допустимо, если наследование было открытым (public)
```

```
pb = ( B* ) pa; // обратное преобразование должно быть явным
```

# Перекрытие ( 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;          // Ошибка! функция h(int) – не L-value выражение  
pa = (A*) pb; pa -> f (1); // Ошибка! функция A::f имеет 2 параметра  
pb = &a; pb -> f (1); // Ошибка! расширяющее присваивание
```

## Видимость и доступность имен

```
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  
}
```

## 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 префикс **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, на который ссылается указатель, переделаем наши классы таким образом:

```
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

Для полиморфных классов следует делать деструкторы виртуальными

```
void f () {  
    student * ps = new student5 ("Morris", 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. работает динамический полиморфизм.

# Механизм виртуальных функций

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 профилем**.
2. Виртуальные функции выбираются по типу объекта, на который ссылается указатель (или ссылка).
3. У виртуальных функций должны быть одинаковые профили. Исключение составляют функции с одинаковым именем и списком формальных параметров, у которых тип результата есть указатель 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 замещения, сокрытия имен.

## Абстрактные классы

**Абстрактным** называется класс, содержащий хотя бы одну **чистую виртуальную** функцию.

**Чистая виртуальная** функция имеет вид: **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

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

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

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

Тогда C c; ~	a pvtbl b c	vtbl для A ~ &A:: f &A:: g &A:: h		для B ~ &A:: f &B:: g &A:: h &B:: m		для C ~ &A:: f [0] &B:: g [1] &C:: h [2] &B:: m [3] &C:: n [4]
--------------	----------------------	---	--	--	--	--

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

## Виртуальные функции. Пример 2.

```
#include <iostream>
using namespace std;

class A {
public:
    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";
    }
};

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

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

int main () {
    A a, *pa = &a;
    B b, *pb = &b;
    C c;

    pa = pb;
    pa -> f (1); // B::f();
    pa -> f (1, 2); // A::f(int, int)
    //pb -> f (1, 2); // ошибка! Эта f не видна
    A & ra = c;
    ra.f(1,1); // C::f(int, int)
    B & rb = c;
    //rb.f(0,0); // ошибка! Эта f не видна
    return 0;
}
```