

Лекции 3-4

- Друзья класса
- Перегрузка операций

Друзья класса

Друг класса – это функция, не являющаяся членом этого класса, но имеющая доступ к его **private** и **protected** членам.

Своих друзей класс объявляет сам в любой зоне описания класса с помощью служебного слова **friend**.

Функция-друг может быть описана внутри класса.

Если функций, имена которых совпадают с объявленной в классе функцией-другом, несколько, то другом считается только та, у которой в точности совпадает прототип.

Другом класса может быть:

- обычная функция:
- функция-член другого класса:
- весь класс:

```
friend void f (...);  
friend void Y::f (..);  
friend class Y;
```

Использование функций - друзей класса

```
class X {  
    int a;  
    friend void fff ( X *, int); // здесь нет this !  
public:  
    void mmm (int);  
};  
  
void fff ( X * p, int i) {  
    p -> a = i;  
}  
  
void X::mmm (int i) {  
    a = i;  
}  
  
void f () {  
    X obj;  
    fff (&obj, 10);  
    obj.mmm (10);  
}
```

Свойства друзей класса

Дружба не обладает ни наследуемостью, ни транзитивностью.

Примеры:

```
class A {  
    friend class B;  
    int a;  
};
```

```
class B {  
    friend class C;  
};
```

```
class C {  
    void f (A* p) {  
        p -> a++;  
    }  
};
```

```
class D: public B {  
    void f (A* p) {  
        p -> a++;  
    }  
};
```

Свойства друзей класса

Дружба не обладает ни наследуемостью, ни транзитивностью.
Примеры:

```
class A {  
    friend class B;  
    int a;  
};
```

```
class B {  
    friend class C;  
};
```

```
class C {  
    void f (A* p) {  
        p -> a++; // ошибка, нет доступа к закрытым членам класса A  
    }  
};
```

```
class D: public B {  
    void f (A* p) {  
        p -> a++; // ошибка, нет доступа к закрытым членам класса A  
    }  
};
```

Преимущества использования друзей класса

1. Эффективность реализации (можно обходить ограничения доступа, предназначенные для обычных пользователей).
2. Функция-друг нескольких классов позволяет упростить интерфейс этих классов.
3. Функция-друг допускает преобразование своего первого параметра-объекта, а метод класса - нет.

Перегрузка операций

- Для перегрузки встроенных операций C++ используется ключевое слово **operator**.
- Перегружать операцию можно с помощью
 - метода класса,
 - внешней функции, в частности, функции-друга (что менее эффективно).
- Нельзя перегружать:
‘.’, ‘::’, ‘?:’, ‘.*’, sizeof, и typeid !!!

Пример 1.

```
class complex {  
    double re, im;  
    public:  
        complex (double r = 0, double i = 0) {        re = r;  
                                                    im = i;  
        }  
        complex operator+ (const complex & a) {  
            complex temp (re + a.re, im + a.im);  
            return temp;  
        } ...  
        // operator double () { return re; } – функция преобразования  
};  
int main () {  
    complex x (1, 2), y (5, 8), z;  
    double t = 7.5;  
    z = x + y; // O.K. – x.operator+ (y);  
    z = z + t; // O.K. – z.operator+ (complex (t)); если есть ф-я преобр., то  
                // неоднозначность: '+' - double или перегруженный  
    z = t + x; // Er.! – т.к. первый операнд по умолчанию – типа complex.  
}
```


Пример 2.

```
class complex {  
    double re, im;  
    public:  
        complex (double r = 0, double i = 0) {  
            re = r;  
            m = i;  
        }  
    friend complex operator+ (const complex & a, const complex & b);  
    ...  
};  
  
complex operator+ (const complex & a, const complex & b) {  
    complex temp (a.re + b.re, a.im + b.im);  
    return temp;  
}  
  
int main () {  
    complex x (1, 2), y (5, 8), z;  
    double t = 7.5;  
    z = x + y; // O.K. – x.operator+ (y);  
    z = z + t; // O.K. – z.operator+ (complex (t));  
    z = t + x; //O.K. – complex (t).operator+ (x);  
}
```

Пример 3.

```
class complex {  
    double re, im;  
    public:  
        friend complex operator * (const complex & a, double b);  
    ...  
};  
complex operator * (const complex & a, double b) {  
    complex temp (a.re * b, a.im * b);  
    return temp;  
}  
int main () {  
    complex x (1, 2), z;  
    double t = 7.5;  
    z = x * t;      // O.K. – x.operator* (t);  
    z = t * x;      // Er.! т.к. нет функции преобразования x --> double, но  
                    // если бы была, была бы неоднозначность:  
                    // * - из double или из complex  
}
```

В таких случаях обычно определяют еще одного друга с прототипом:
complex operator * (**double** b, **const** complex & a);

Замечания

- **n-местные операции** перегружаются
 - а) **методом с (n-1) параметром,**
 - б) **внешней функцией с n параметрами;**

- в любом случае **сохраняется приоритет, ассоциативность и местность операций;**

- операции

= , [] , () и ->

можно перегрузить **ТОЛЬКО** нестатическими методами класса, что гарантирует, что первым операндом будет сам объект, к которому операция применяется;

Особенности перегрузки операций ++ и --

complex x;

префиксная ++: ++ x; ~ x.operator ++ ();

```
complex & operator ++ () {  
    re = re + 1;  
    im = im + 1;  
    return *this;  
}
```

постфиксная ++: x++; ~ x.operator ++ (0);

```
complex operator ++ (int) {  
    complex c = * this;  
    re = re + 1;  
    im = im + 1;  
    return c;  
}
```

Пример перегрузки операции «()» и операции вывода «<<»

```
class Matrix {  
    double M [ 3 ] [ 3 ];  
public:  
    Matrix ();  
    double & operator ( ) (int i, int j) const {  
        return M [ i ] [ j ];  
    }  
friend ostream & operator << (ostream & s, const Matrix & a) {  
    for (int i = 0; i < 3 ; i ++ ) {  
        for (int j = 0; j < 3; j ++ )  
            s << a (i, j) << ' ';  
        s << endl;  
    }  
    return s;  
}  
};
```

Перегрузка функций

О перегрузке можно говорить только для функций из одной области видимости!

Алгоритм поиска и выбора 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 это 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. - '+' 2 пар. - '+'
```

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

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

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

Разрешим ли вызов?

Пример 1.

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

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

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

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

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

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

Т.к. пересечение множеств, построенных для каждого параметра, состоит только из одной функция `f (X, int)`, — вызов разрешим.

Пример 2.

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

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

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

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

Разрешим ли вызов?

Пример 2.

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

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

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

```
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 при выборе 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

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:

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

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

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

Неоднозначности нет, хотя **short -> int & double,**
 float -> int & double.

в) Отождествление с помощью стандартных преобразований.

- Все оставшиеся стандартные целочисленные и вещественные преобразования, которые могут выполняться неявно, а также преобразование объекта однозначного доступного базового класса к объекту производного класса.
- Преобразования указателей:
 - 0 --> любой указатель,
 - любой указатель --> **void***,
 - derived* --> base* - для однозначного 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; // Er.!      т.к. может интерпретироваться двояко:  
                        // b.operator int () +1 – целочисленный ‘+’ или  
                        // b.operator+ (Boolean (1)) – Boolean ‘+’  
    k = b + 1; // Er.!      -- “ --  
}
```

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; // Er.! , т.к. требуется a.operator X ().operator
```

```
int ()
```

```
    ...
```

```
}
```

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

д) Отождествление по

Пример 1:

```
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, ...);  
}
```

Пример 2:

Многоточие может приводить к неоднозначности:

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

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

```
void g () {...
```

```
    f (1); // Er.! т.к. отождествление по  
           // первому параметру дает  
           // обе функции.
```

```
}
```