


Функции-члены классов

- функции **объявленные** внутри класса: методы класса

```
class Stack {  
    ...  
    float pop() {return v[--top];} // и объявление и определение  
    void push(float val);          // только объявление  
};  
// определение push() из класса Stack  
void Stack::push(float val) { // Stack:: обязательно!  
    v[top++]=val;  
}
```

- 👉 Если функции **определена** внутри класса то это встроенная (**inline**) функция

 Функции-члены вызываются только для объектов такого же класса

```
Stack S, T;  
Stack* pS = new Stack;  
S.push(10.);           // вызов функции для объекта S  
T.push(10.);           // вызов функции для объекта T  
pS->push(10.);         // вызов функции для указателя pS  
push(10.);           // ERROR: нет объекта для вызова  
push(S,10.);        // ERROR: компилятор вас не понимает
```

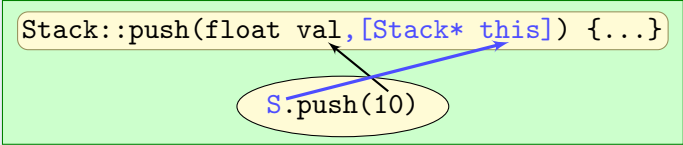
Как это работает?

Указатель «на себя» `this`

в функцию «передается» адрес вызываемого объекта: указатель `this`

```
Stack::push(float val, [Stack* this]) {...}
```

`S.push(10)`



явное и неявное использование `this`:

```
void push(float val) { this->v[this->top++] = val; } // explicit
void push(float val) { v[top++] = val; }           // implicit
```

🔔 `this` используется неявно в большинстве случаев

`this` – это не переменная, это служебное слово!

🔔 запрещено изменять: `(this = ...; // ERROR)`

🔔 невозможно получить адрес: `(&this; // ERROR)`

Константные функции-члены

```
int Size() const { return top;}
```

- модификатор `const` после списка аргументов означает, что функция не изменяет состояние объекта: в таких функциях `this` – константный указатель
- 👉 модификатор `const` входит в сигнатуру функции: могут существовать две функции отличающиеся только наличием-отсутствием `const`
- 👉 константные методы можно вызывать и для константных объектов, но такие функции не могут вызывать другие не константные методы
- 👉 в статических функциях класса нет `this`

Друзья классов `friend`

- Что бы внешние по отношению к классу функции или классы могли иметь полный доступ к её приватным членам их надо объявить `friend`
- Декларацию `friend` помещают внутрь класса в любое место

Например `operator<<()` часто объявляется другом класса:

- 1 `operator<<()` обычно использует приватные переменные
 - 2 `operator<<()` невозможно объявить методом класса
- ```
friend ostream& operator << (ostream& out, ...);
```

# Перегрузка операторов

Для переопределения операторов используется конструкция:

```
ret_type operator⊙ (list_of_arguments)
```

- ⊙ – «имя» оператора (+, \*, « ... »)
  - `ret_type` – возвращаемый тип; `list_of_arguments` – список аргументов, как в обычной функции
- 👉 Переопределить можно почти любой **существующий** оператор C++

## Запрещено перегружать

|          |                                     |
|----------|-------------------------------------|
| ::       | разрешение области видимости        |
| .        | выбор члена                         |
| .*       | выбор члена через указатель на член |
| ? :      | троичный оператор                   |
| sizeof() | оператор sizeof                     |

## Класс рациональных чисел для дальнейших примеров

```
class Rational { // number of the form x/y
private:
 int x;
 unsigned int y;
public:
 Rational(int a=0, int b=1) { // ctr
 if (b>0) {
 x=a; y=b;
 } else {
 x=-a; y=-b;
 }
 }
};
```

# Перегрузка операторов ввода-вывода

## добавляем в класс

```
friend istream& operator >> (istream& in, Rational& r);
friend ostream& operator << (ostream& out, const Rational& r);
```

## определение функций ввода и вывода: вне класса

```
istream& operator >> (istream& in, Rational& r) {
 int x,y;
 in >> x >> y;
 r = Rational(x,y); // ctor + assignment (copy or move)
 return in;
}

ostream& operator << (ostream& out, const Rational& r) {
 out << r.x;
 if(r.y > 1 && r.x != 0) out << "/" << r.y;
 return out;
}
```



## тестовая программа

```
Rational a;
Rational b(3);
Rational c(3,-4);
cout<<" a= "<<a<<" b= "<<b<<" c= "<<c<<endl; // a= 0 b= 3 c= -3/4
Rational d;
cout << " type d(x/y) as two integer numbers x and y:";
cin >> d; // вводим с клавиатуры: 1 -4
cout << " d= " << d << endl; // d= -1/4
```

- 👉 В операторе `<<` важно использовать `const reference`;  
если убрать `const` то будут проблемы печати временных объектов:  

```
cout << "a+b= " << a+b << endl; // invalid initialization
// of non-const reference
```
- 👉 В операторе `>>` нет внутренних переменных класса, однако оставляем `friend` в объявлении функции для улучшения читаемости

# Унарные операторы

## Примеры унарных операторов

`++a; -a; !a; &a;`

Выражение  $\odot A$  можно определить функциями:

|             | метод класса                                 | внешняя функция                             |
|-------------|----------------------------------------------|---------------------------------------------|
| вид функции | <code>A.operator<math>\odot</math> ()</code> | <code>operator<math>\odot</math> (A)</code> |

Что «лучше» определяется предпочтениями программиста

- ☞ Метод класса подчеркивает связь с классом, поэтому «предпочтительней»

# Префиксный и постфиксный операторы ++

## Как различить функции для A++ и ++A?

- ✓ Постфиксный оператор имеет «фиктивный» аргумент типа `int`:

|                  | <code>++A</code>             | <code>A++</code>                |
|------------------|------------------------------|---------------------------------|
| метод класса:    | <code>A.operator++ ()</code> | <code>A.operator++ (int)</code> |
| внешняя функция: | <code>operator++ (A)</code>  | <code>operator++ (A,int)</code> |

## Пример для методов класса ++A и A++

```
Rational& operator++ () { //++A
 x+=y;
 return *this;
}
```

```
Rational operator++ (int) { //A++
 Rational tmp(*this);
 x+=y;
 return tmp;
}
```

## тест

```
d = Rational(-1,4);
Rational q1 = d++;
Rational q2 = ++d;
cout << " d= " << d // d= 7/4
 << " q1= " << q1 // q1= -1/4
 << " q2= " << q2 << endl; // q2= 7/4
```

# Унарный минус


добавляем в public раздел класса

```
Rational operator- () const {
 return Rational(-x,y);
}
```

## тест

```
const Rational cd(-1,4);
Rational d1 = -cd;
cout << " cd= " << cd << " d1= " << d1 << endl; // cd= -1/4 d1= 1/4
```

## Обратите внимание

 `operator-()` константная функция: не меняет число, для которого вызывается

# Перегрузка операторов: бинарные операторы

## Примеры бинарных операторов

```
a+b; a-b; a = b; a += b; a < b; cout << a;
```

Выражение  $A \odot B$  можно определить функциями:

|             | метод класса          | внешняя функция       |
|-------------|-----------------------|-----------------------|
| вид функции | $A.operator\odot (B)$ | $operator\odot (A,B)$ |

## Что лучше определяется предпочтениями программиста

- ☞ Для  $(+ - * \dots)$  внешняя функция выглядит более логично из-за симметрии операндов
- ☞ Для  $(+= *= \dots)$  «предпочтительнее» функция-член класса

# Rational += Rational

добавляем в public раздел класса

```
Rational& operator += (const Rational& r); // member of class
```

```
Rational& Rational::operator += (const Rational& r) { // q += r
 x = x*r.y + y*r.x;
 y = y*r.y;
 return *this; // for expressions like c = (a += b);
}
```

👉 должно быть возвращаемое значение (\*this)

👉 необходимо выполнить проверку `c+=c`: аргумент совпадает с `this`!

```
cout << "c=" << c << " d=" << d << endl; // c=3/4 d=-1/4
cout << (c+=d) << " : " << (c+=c) << endl; // 8/16 : 256/256
```

# Rational + Rational

добавляем в класс

```
friend Rational operator+(const Rational& r1,const Rational& r2);
```

определяем функцию через operator += ()

```
Rational operator + (const Rational& r1, const Rational& r2) { // r1+r2
 Rational tmp = r1;
 return tmp += r2;
}
```

👉 В данном случае `friend` не нужен, оставлен «для читаемости» кода

ТЕСТ

```
Rational e = c+d;
cout<<" c="<<c<<" d="<<d<<" e="<<e<<endl; // c=8/16 d=-1/4 e=16/64
```



## Операции $\text{int} + \text{Rational}$ и $\text{Rational} + \text{int}$

добавляем в `public` раздел класса

```
Rational& operator += (int i);
friend Rational operator + (int i, const Rational& r);
friend Rational operator + (const Rational& r, int i);
```

```
Rational& Rational::operator += (int i) { // r += i
 x += i*y;
 return *this;
}

Rational operator + (int i, const Rational& r) { // i + r
 Rational tmp = r;
 return tmp += i;
}

Rational operator + (const Rational& r, int i) { // r + i
 Rational tmp = r;
 return tmp += i;
}
```

## ТЕСТ

```
e = (c+=1);
cout << " e= " << e << endl; // e= 24/16
Rational f1 = e+5;
Rational f2 = 5+e;
cout << " f1= " << f1 << " f2= " << f2 << endl; // f1= 104/16 f2= 104/16
```

## Зачем отдельно писать эти функции?

- ☞ для оптимизации: в три раза меньше операций умножения
- ☞ для разрешения неоднозначности неявного преобразования типов:  
// если нет функций для `int+Rational`, `Rational+int`  
`Rational f1 = r+5;` // => `r+Rational(int)` OK!  
`Rational f2 = 5+r;` // => `5+double(r)` и затем `Rational(double)`?
- ✓ второй случай можно запретить используя декларацию:  
`explicit Rational(double)`

## Другие операторы

- Оператор вызова функции: `operator () (...)`
- Индексация: `operator [] (int index)`
- Операторы `new` и `delete`
- Разыменование: `operator -> ()`
- Оператор «звёздочка» `operator * ()`

# Оператор функционального вызова ()

- Функция `operator () (...)` должна быть членом класса
- Количество и тип аргументов, а также тип возвращаемого значения могут быть любыми

```
Rational& operator () (int a, unsigned int b) {
 x=a; y=b;
 return *this;
}
```

## ТЕСТ

```
Rational t(1);
Rational f = d + t(-6,4);
cout<<" d= "<<d<<" t ="<<t<<" f= "<<f<< endl; // d= 7/4 t= -6/4 f= 4/16
```

# Оператор «взятия индекса» []

- `operator [] (...)` должен быть членом класса
- Количество и тип аргументов, а также тип возвращаемого значения могут быть любыми
- ☞ ожидается целочисленный аргумент и то что функция вернет ссылку:  
`element-type& operator[](integral type)`

**Пример:** только в демонстрационных целях

```
enum TypeR {NOM,DENOM};
int& Rational::operator [] (TypeR idx) {
 switch(idx) {
 case NOM: return x;
 case DENOM: return (int&)y;
 }
}
```

## TEST

```
a = Rational(1,5);
cout << " a= " << a // a= 1/5
 << " NOM= " << a[Rational::NOM] // NOM= 1
 << " DENOM= " << a[Rational::DENOM] << endl; // DENOM= 5
a[Rational::NOM]++;
a[Rational::DENOM]--;
cout << " NOM++; DENOM-- = " << a << endl; // NOM++; DENOM-- = 2/4
a[Rational::DENOM]=-1;
cout << " a= " << a << endl; // a= 2/4294967295
```

# Краткое резюме по перегрузке операторов

- ❶ Нельзя ввести новые операторы, менять можно только имеющиеся
- ❷ Операторы перегружаются только для пользовательских классов, за исключением операторов `new` и `delete`
- ❸ Контроль за адекватным поведением и согласованностью операторов лежит полностью на совести программиста
- ❹ Аккуратная разработка класса требует многочисленных тестирующих примеров

- 1 Позволяет вычислить константы на стадии компиляции
- 2 Позволяет задать `constexpr` функцию, которую можно использовать как для вычисления констант компилятором, так и далее во время исполнения программы

## 1 Вычисление констант на стадии компиляции

```
constexpr unsigned long Factorial(unsigned int n) {
 return n < 1 ? 1 : n*Factorial(n-1); // no loops in C++11
}

constexpr int F10 = Factorial(10); // compile time
constexpr double half_ln_2pi=log(2*M_PI)/2; // is std::log() constexpr?
constexpr array v {1,2,3}; // C++17
constexpr auto sum = v.front() + v.back(); // C++17
```



## 2 Использование constexpr функции

```
constexpr unsigned long Factorial14(unsigned int n) { // C++14
 unsigned long result = 1UL;
 for (; n>1; --n) { result *=n; }
 return result;
}

constexpr int F7 = Factorial14(7); // compile time
cout << "F7= " << F7 << endl; // F7= 5040
volatile int b = 8; // disallow optimization
cout << b << "!=" << Factorial14(b) << endl; // 8!=40320 - run time
```

☞ constexpr подразумевает const

☞ constexpr функция может вызываться как обычная функция

☞ вычисления должны быть «достаточно простыми»