Функции члены класса или методы класса

☞ Это функции объявленные как члены класса без `static` или `friend` спецификаторов

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

☞ функции определенные внутри класса объявляются встроенными (`inline`) функциями

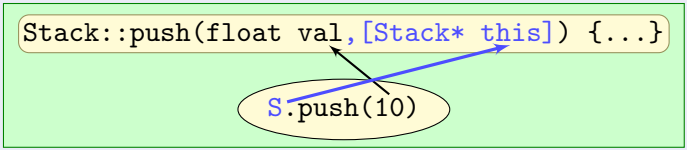
☞ Функции-члены вызываются только для объектов такого же класса

```
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



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;}
```

const после списка аргументов означает

- функция не меняет объект: в них `this` – константный указатель
- для константных объектов можно вызывать только такие функции
- квалификатор `const` входит в сигнатуру функции (могут существовать две функции отличающиеся только наличием-отсутствием `const`)

☞ про такие функции говорят, что они имеют cv-квалификатор: `const/volatile qualifier`

Ссылочные квалификаторы (C++11)

Ref-qualifiers

Функции члены могут иметь дополнительные ссылочные квалификаторы:

- `lvalue ref-qualifier` (токен `&` после списка параметров): вызывается для `lvalue` объектов
- `rvalue ref-qualifier` (токен `&&` после списка параметров): вызывается для `rvalue` (временных) объектов

☞ ссылочная квалификация не меняет `this`: `*this` всегда `lvalue`

lvalue ref-qualifier

☞ для lvalue объектов

```
Stack Stack::Rev() const & {
  cout<<__PRETTY_FUNCTION__<<'\n';
  Stack rs(*this);
  for (int i=0; i < top; ++i) {
    rs.v[i] = this->v[top-1-i];
  }
 return rs;
}
```

rvalue ref-qualifier

☞ для rvalue объектов

```
Stack Stack::Rev() && {
  cout<<__PRETTY_FUNCTION__<<'\n';
  for (int i=0; i < top/2; ++i) {
    std::swap(v[i], v[top-1-i]);
  }
 return std::move(*this);
}
```

```
Stack S {1,2,3};
Stack rs1 = S.Rev(); // Stack Stack::Rev() const &
cout << "rs1= " << rs1 << endl; // rs1= 3  2  1

Stack rs2 = Stack({6,7,8}).Rev(); // Stack Stack::Rev() &&
cout << "rs2= " << rs2 << endl; // rs2= 8  7  6
```

Друзья классов, `friend`

- декларация `friend` применяется к внешним по отношению к классу функциям или классам что бы те могли иметь полный доступ к её приватным членам
- `friend`-объявление помещают внутрь класса в любое место

`operator≪()` вывода на печать часто объявляется другом класса

1. `operator≪()` обычно использует приватные перемененные
2. `operator≪()` невозможно объявить методом класса

```
friend ostream& operator << (ostream& out, ...);
```

Перегрузка операторов

Для переопределения операторов используется конструкция:

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

- $\odot$ – «имя» оператора ($+$, $*$, $\ll \ldots$)
- `ret_type` – возвращаемый тип; `list_of_arguments` – список аргументов, как в обычной фунуции

☞ 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;
   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) {
  return out << r.x << "/" << r.y;
}
```

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;
```

Возможные функции для унарного оператора ⊙

	метод класса	внешняя функция
⊙A ==	A.operator⊙ ()	operator⊙ (A)

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`:

	`++A`	`A++`
метод класса:	`A.operator++ ()`	`A.operator++ (int)`
внешняя функция:	`operator++ (A)`	`operator++ (A,int)`

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;
```

Возможные функции для бинарного оператора:

	метод класса	внешняя функция
A⊙B ==	A.operator⊙ (B)	operator⊙ (A,B)

Что лучше определяется предпочтениями программиста

- ☞ Для ($+ - * \ldots$) внешняя функция выглядит более логично из-за симметрии операндов A и B
- ☞ Для (+= *= ...) «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);
}
...
cout << "c=" << c <<" d=" << d << endl;     // c=3/4 d=-1/4
cout << (c+=d) << " : " << (c+=c) << endl; // 8/16 : 256/256
```

- ☞ должно быть возвращаемое значение `(*this)`
- ☞ необходимо выполнить проверку `c+=c`: аргумент `r` совпадает с `this`

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

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

Операции int + Rational и Rational + 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; // не компилируется если operator+() метод кл.
    // а также возможно 5+double(r) и затем Rational(double)?
```

- ✓ неявное преобразование к/от `double` лучше запретить:

```
explicit operator double() const
explicit Rational(double)
```

Другие операторы

- Оператор вызова функции: `operator () (...)`
- Индексация: `operator [] (int index)`
- Операторы `new` и `delete`
- Оператор доступа к члену класса: `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;
}
...
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;
  }
}
```

тест

```
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]=0;
cout << " a= " << a << endl;                  // a= 2/0
```

Краткое резюме по перегрузке операторов

1. Нельзя ввести новые операторы, можно менять только имеющиеся
2. Операторы перегружаются только для пользовательских классов, за исключением операторов `new` и `delete`
3. Контроль за адекватным поведением и согласованностью операторов лежит полностью на совести программиста
4. Аккуратная разработка класса требует многочисленных тестирующих примеров