# ПЕРЕГРУЗКА ОПЕРАТОРОВ В C++

**Перегрузка операторов** в программировании — один из способов реализации полиморфизма, заключающийся в возможности одновременного существования в одной области видимости нескольких различных вариантов применения операторов, имеющих одно и то же имя, но различающихся типами параметров, к которым они применяются. В изданиях советского периода аналогичные механизмы назывались переопределением или повторным определением, перекрытием операций.

Если оператор «перегружен», то его можно использовать в других методах в обычном для него виде. Например, команды поэлементного суммирования двух массивов a1 и a2

```
a1.add(a2);
```

```
a3 = add(a1, a2);
```

лучше вызвать более естественном способом:

```
a1 = a1 + a2;
```

```
a3 = a1 + a2;
```

В данном примере оператор '+' считается перегруженным.

## Основы перегрузки операторов

Во-первых, почти любой существующий оператор в языке C++ может быть перегружен. Исключениями являются:

- тернарный оператор (?:);
- оператор sizeof;
- оператор разрешения области видимости (::);
- оператор выбора члена (.);
- указатель, как оператор выбора члена (.*).

Во-вторых, вы можете перегрузить только существующие операторы. Вы не можете создавать новые или переименовывать существующие.

В-третьих, по крайней мере один из операндов перегруженного оператора должен быть пользовательского типа данных. Это означает, что вы не можете перегрузить operator+() для выполнения операции сложения значения типа int со значением типа double. Однако вы можете перегрузить operator+() для выполнения операции сложения значения типа int с объектом класса MyComplexNumber.

В-четвертых, изначальное количество операндов, поддерживаемых оператором, изменить невозможно. Т.е. с бинарным оператором используются только два операнда, с унарным — только один, с тернарным — только три.

Наконец, все операторы сохраняют свой приоритет и ассоциативность по умолчанию (независимо от того, для чего они используются), и это не может быть изменено.

**Замечание.** Некоторые начинающие программисты пытаются перегрузить побитовый оператор XOR (^) для выполнения операции возведения в степень. Однако в языке C++ у оператора ^ приоритет ниже, чем у базовых арифметических операторов (+, -, *, /), и это приведет к некорректной обработке выражений.

**Правило: При перегрузке операторов старайтесь максимально приближенно сохранять функционал операторов в соответствии с их первоначальными применениями.**

## Особенности реализации

Для заданного класса операторную функцию в классе можно реализовать:

- внутри класса

В этом случае, операторная функция является методом класса.

- за пределами класса

В этом случае операторная функция объявляется за пределами класса как «дружественная» (с ключевым словом friend).

| Оператор | Рекомендуемая форма |
|---|---|
| Все унарные операторы | Член класса |
| = () [] -> ->* | Обязательно член класса |
| += -= /= *= ^= &= \|= %= >>= <<= | Член класса |
| Остальные бинарные операторы | Не член класса |

**Общая форма операторной функции, реализованной в классе, имеет следующий вид:**

```
type operator#(arguments_list) {

    // некоторые операции

    // ...

};
```

type – тип значения, которое возвращается операторной функцией;

operator# – ключевое слово, определяющее операторную функцию в классе. Символ # заменяется оператором языка C++, который перегружается. Например, если перегружается оператор +, то нужно указать operator+;

argument_list – список параметров, которые получает операторная функция. Если перегружается бинарный оператор, то argument_list содержит один аргумент. Если перегружается унарный оператор, то список аргументов пустой. Если функция внешняя и является дружественной, то аргументов может быть 1 и 2 соответственно. Параметрами

«дружественной» функции являются правыми операндами. Чтобы обратиться к левому операнду используется указатель на объект, для которого вызывается оператор – this. Часто указатель this опускается.

Операторная функция может возвращать объекты любых типов. Наиболее часто операторная функция возвращает объект типа класса, в котором она реализованная или с которыми она работает.

**Основные правила на тему «Аргументы и возвращаемые значения»**

- **Если аргумент не изменяется оператором**, в случае, например унарного плюса, его нужно передавать **как ссылку на константу**. Вообще, это справедливо для почти всех арифметических операторов (сложение, вычитание, умножение...)
- Тип возвращаемого значения зависит от сути оператора. Если оператор должен возвращать новое значение, то необходимо создавать новый объект (как в случае бинарного плюса). Если вы хотите запретить изменение объекта как l-value, то нужно возвращать его константным.
- **Для операторов присваивания** необходимо возвращать **ссылку на измененный элемент**. Также, если вы хотите использовать оператор присваивания в конструкциях вида (x=y).f(), где функция f() вызывается для для переменной x, после присваивания ей y, то не возвращайте ссылку на константу, возвращайте просто ссылку.
- Логические операторы должны возвращать в худшем случае int, а в лучшем bool.

**Оператор присваивания обязательно определяется в виде функции класса (внутри класса),** потому что он неразрывно связан с объектом, находящимся слева от "=".

Пример оператора присваивания (обратите внимание на подход к изменению объекта):

```
Integer& operator=(const Integer& right) {
    //проверка на самоприсваивание
    if (this == &right) {
        return *this;
    }
    value = right.value;
    return *this;
}
```

**Замечание.** Проверка на самоприсваивание нужна, чтобы не тратить время на операции типа `x = x;` ведь полей у объектов может быть много, в том числе целые массивы данных и данная операция очень трудозатратна!

Для понимания предлагается посмотреть пример.

## Пример

**Пример.** Задан класс Complex, в котором перегружаются два оператора:

- унарный оператор **‘+’**, возвращающий модуль комплексного числа (тип double);

- бинарный оператор ‘+’, возвращающий сумму комплексных чисел. Операторная функция возвращает объект типа Complex;
- бинарный оператор ‘+’, который добавляет к комплексному числу некоторое вещественное число. В этом случае операторная функция получает входным параметром вещественное число и возвращает объект типа Complex.

Текст класса следующий:

```
// класс Complex
class Complex
{
private:
    float real; // действительная часть
    float imag; // мнимая часть

public:
    // конструкторы
    Complex(void)
    {
        real = imag = 0;
    }

    Complex(float _real, float _imag)
    {
        real = _real;
        imag = _imag;
    }

    // методы доступа
    float GetR(void) { return real; }
    float GetI(void) { return imag; }

    void SetRI(float _real, float _imag)
    {
        real = _real;
        imag = _imag;
    }

    // объявление операторной функции, перегружающей бинарный
'+'
    // функция возвращает объект, содержащий сумму двух
комплексных чисел
    Complex& operator+(const Complex& c)
    {
        Complex c2; // временный объект

        // суммирование комплексных чисел
        c2.real = this->real + c.real;
        c2.imag = imag + c.imag;

        return *c2;
```

```
    }

    // объявление операторной функции, перегружающей унарный '+'
    // функция возвращает модуль комплексного числа
    float operator+(void)
    {
        float res;
        res = std::sqrt(real*real+imag*imag);
        return res;
    }

    // объявление операторной функции operator+()
    // функция добавляет к комплексному числу некоторое число,
которое есть входящим параметром
    Complex& operator+(const float real)
    {
        Complex c2; // результирующий объект
        c2.real = this->real + real;
        c2.imag = this->imag;
        return *c2;
    }
};
```

Далее демонстрируется использование класса Complex и перегруженных операторных функций в некотором другому методе

```
Complex c1(1,5);
Complex c2(3,-8);
Complex c3; // результирующий объект
double d;

// проверка
c3 = c1 + c2;
d = c3.GetR(); // d = 1 + 3 = 4
d = c3.GetI(); // d = 5 + (-8) = -3

// перегруженный унарный оператор '+'
d = +c1; // d = |1 + 5j| = 5.09902 - модуль числа
d = +c2; // d = |3 + (-8)j| = 8.544

// вызов перегруженного бинарного '+',
// добавить к комплексному числу число
c3 = c1 + 5.0;
d = c3.GetR(); // d = 1 + 5 = 6
```

Как уже было сказано, существует два способа перегрузки любого оператора:

- с помощью операторной функции, которая реализована внутри класса (это мы уже рассмотрели);
- **с помощью операторной функции, которая реализована как «дружественная» (friend) к классу.**

Первым параметром «дружественной» функции есть левый операнд, а вторым параметром правый операнд. Эти отличия возникают из-за того, что «дружественная» операторная функция не получает неявного указателя this.

## Пример

Задан класс Complex, реализующий комплексное число. В классе объявляются внутренние переменные, конструкторы, методы доступа и «дружественная» функция operator-(). «Дружественная» функция operator-(), реализованная за пределами класса, осуществляет вычитание комплексных чисел.

```
// класс Complex
class Complex
{
private:
     float real; // вещественная часть
     float imag; // мнимая часть

public:
  // конструкторы
     Complex(void)
     {
          real = imag = 0;
     }

     Complex(float _real, float _imag)
     {
          real = _real;
          imag = _imag;
     }

  // методы доступа
     float GetR(void) { return real; }
     float GetI(void) { return imag; }

     void SetRI(float _real, float _imag)
     {
          real = _real;
          imag = _imag;
     }

  // объявление "дружественной" к классу Complex операторной функции
     friend Complex& operator-(const Complex& c1, const Complex&
c2);
};

// "дружественная" к классу Complex операторная функция,
// реализована за пределами класса,
```

```
// осуществляет вычитание комплексных чисел
Complex& operator-( const Complex& c1, const Complex& c2)
{
     Complex c;  // создать объект класса Complex

   // вычитание комплексных чисел
     c.real = c1.real - c2.real;
     c.imag = c1.imag - c2.imag;

     return  *c;
}
```

Использование класса Complex в другом методе

```
// использование "дружественной" операторной функции

Complex c1(5,6);
Complex c2(3,-2);
Complex c3;  // результат
float  a, b;

// проверка
a = c1.GetR();  // a = 5
b = c1.GetI();  // b = 6

// вызов "дружественной" к классу Complex операторной функции
c3 = c1 - c2;

// результат
a = c3.GetR();  // a = 5-3 = 2
b = c3.GetI();  // b = 6-(-2) = 8
```

Вообще, если семантически нет разницы как определять оператор (как внутреннюю или как внешнюю функцию), то лучше его оформить в виде функции класса, чтобы подчеркнуть связь, плюс помимо этого функция будет подставляемой (inline).

К тому же, иногда может возникнуть потребность в том, чтобы представить левосторонний операнд объектом другого класса. Наверное, самый яркий пример — переопределение << и >> для потоков ввода/вывода, их необходимо реализовывать как ВНЕШНИЕ ФУНКЦИИ.