Ключевое **operator** слово объявляет функцию, указывающую, что *означает оператор-символ* при применении к экземплярам класса. Это дает оператору более одного значения — "перегружает" его. Компилятор различает разные значения оператора, проверяя типы его операндов.
Функцию большинства встроенных операторов можно переопределить глобально или для отдельных классов. Перегруженные операторы реализуются в виде функции. Имя перегруженного оператора - operatorX, где X - сам оператор, то есть перегрузить оператор сложения, надо определить ф-ию с именем operator+.

Пример:

```
// operator_overloading.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

struct Complex {
   Complex( double r, double i ) : re(r), im(i) {}
   Complex operator+( Complex &other );
   void Display( ) {   cout << re << ", " << im << endl; }
private:
   double re, im;
};

// Operator overloaded using a member function
Complex Complex::operator+( Complex &other ) {
   return Complex( re + other.re, im + other.im );
}

int main() {
   Complex a = Complex( 1.2, 3.4 );
   Complex b = Complex( 5.6, 7.8 );
   Complex c = Complex( 0.0, 0.0 );

   c = a + b;
   c.Display(); }
```

ПЕРЕГРУЗКА

Позволяет определить для объектов класса встроенные операторы. Создать новые для C++ операторы нельзя, можно только существующие переопределить.
Если функция оператора определена как отдельная функция и не является членом класса, то количество параметров такой функции совпадает с количеством операндов оператора. Например, у функции, которая представляет унарный оператор, будет один параметр, а у функции, которая представляет бинарный оператор, - два параметра. Если оператор принимает два операнда, то первый операнд передается первому параметру функции, а второй операнд - второму параметру. При этом как минимум один из параметров должен представлять тип класса.

Формальное определение операторов в виде функций-членов класса:

```
1 // бинарный оператор
2 ReturnType operator Op(Type
3 right_operand);
4 // унарный оператор
  ClassType& operator Op();
```

Формальное определение операторов в виде функций, которые не являются членами класса:

```
// бинарный оператор
ReturnType operator Op(const ClassType& left_operand, Type
right_operand);
// альтернативное определение, где класс, для которого создается
оператор, представляет правый операнд
ReturnType operator Op(Type left_operand, const ClassType&
right_operand);
// унарный оператор
ClassType& operator Op(ClassType& obj);
```

Здесь `ClassType` представляет тип, для которого определяется оператор. `Type` - тип другого операнда, который может совпадать, а может и не совпадать с первым. `ReturnType` - тип возвращаемого результата, который также может совпадать с одним из типов операндов, а может и отличаться. `Op` - сама операция.

ПЕРЕГУЗКА УНАРНОГО ОПЕРАТОРА

Рассмотрим примеры перегрузки унарных операторов для определенного выше класса Integer. Заодно определим их в виде дружественных функций и рассмотрим операторы декремента и инкремента:

```
class Integer
{
private:
    int value;
public:
    Integer(int i): value(i)
   {}

    //унарный +
    friend const Integer& operator+(const Integer& i);

    //унарный -
    friend const Integer operator-(const Integer& i);

    //префиксный инкремент
    friend const Integer& operator++(Integer& i);

    //постфиксный инкремент
    friend const Integer operator++(Integer& i, int);

    //префиксный декремент
    friend const Integer& operator--(Integer& i);

    //постфиксный декремент
    friend const Integer operator--(Integer& i, int);
};

//унарный плюс ничего не делает.
const Integer& operator+(const Integer& i) {
    return i.value;
}

const Integer operator-(const Integer& i) {
    return Integer(-i.value);
}

//префиксная версия возвращает значение после инкремента
const Integer& operator++(Integer& i) {
```

```
    i.value++;
    return i;
}

//постфиксная версия возвращает значение до инкремента
const Integer operator++(Integer& i, int) {
    Integer oldValue(i.value);
    i.value++;
    return oldValue;
}

//префиксная версия возвращает значение после декремента
const Integer& operator--(Integer& i) {
    i.value--;
    return i;
}

//постфиксная версия возвращает значение до декремента
const Integer operator--(Integer& i, int) {
    Integer oldValue(i.value);
    i.value--;
    return oldValue;
}
```

Теперь вы знаете, как компилятор различает префиксные и постфиксные версии декремента и инкремента. В случае, когда он видит выражение ++i, то вызывается функция operator++(a). Если же он видит i++, то вызывается operator++(a, int). То есть вызывается перегруженная функция operator++, и именно для этого используется фиктивный параметр int в постфиксной версии.

ПЕРЕГРУЗКА БИНАРНОГО ОПЕРАТОРА

Рассмотрим синтаксис перегрузки бинарных операторов. Перегрузим один оператор, который возвращает l-значение, один условный оператор и один оператор, создающий новое значение (определим их глобально)

```
class Integer
{
private:
    int value;
public:
    Integer(int i): value(i)
```

```
   {}
    friend const Integer operator+(const Integer& left, const
Integer& right);

    friend Integer& operator+=(Integer& left, const Integer& right);

    friend bool operator==(const Integer& left, const Integer&
right);
};

const Integer operator+(const Integer& left, const Integer& right) {
    return Integer(left.value + right.value);
}

Integer& operator+=(Integer& left, const Integer& right) {
    left.value += right.value;
    return left;
}

bool operator==(const Integer& left, const Integer& right) {
    return left.value == right.value;
}
```

Во всех этих примерах операторы перегружаются для одного типа, однако, это необязательно. Можно, к примеру, перегрузить сложение нашего типа Integer и по его подобию Float.

НЕПЕРЕГРУЖАЕМЫЕ ОПЕРАТОРЫ

Сделано хз почему, мб из-за безопасности. Перечень:

- оператор выбора члена класса **.**
- Оператор разыменования указателя на члена класса **.***
- **оператора возведения в степень в C++ нет (**)**
- нельзя изменять приоритеты операторов
- Свои операторы тоже нельзя создавать

Операторы new и delete МОЖНО перегузить и они могут принимать сколько угодно аргументов и возвращать тоже любое значение (вход и выход могут по типу не совпадать).

Примерчик:

```
void* MyClass::operator new(std::size_t s, int a)
{
    void * p = malloc(s*a);
    if(p == nullptr)
```

```
        throw "No free memory!";
    return p;
}
// ...
// Вызов:
MyClass * p = new(12) MyClass;
```

ЧУТЬ КОНТЕКСТА

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

Вот так рекомендуется юзать. Почему так? Во-первых, на некоторые операторы изначально наложено ограничение. Вообще, если семантически нет разницы как определять оператор, 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.Унарное выражение состоит либо из унарного оператора, за которым следует операнд, либо из ключевого слова **sizeof** или **_Alignof**, за которым следует выражение. Выражением может быть либо имя переменной, либо выражение приведения типа. В последнем случае выражение должно быть заключено в круглые скобки. Бинарное выражение состоит из 2 операндов, соединенных бинарным оператором.

Символ	Имя.
`-` `~` `!`	Операторы отрицания и дополнения
`*` `&`	Операторы косвенного обращения и взятия адреса
`_Alignof`	Оператор выравнивания (начиная с выпуска C11)
`sizeof`	Оператор определения размера
`+`	Оператор унарного плюса
`++` `--`	Унарные операторы инкремента и декремента

Это всё унарные операторы, работают с 1 переменной/чем-то типа того.

Символ	Имя.
* / %	Мультипликативные операторы
+ -	Аддитивные операторы
<< >>	Операторы сдвига
< > <= >= == !=	Реляционные операторы
& | ^	битовые операторы;
&& ||	Логические операторы
,	Оператор последовательного вычисления

Это всё бинарные, работают с 2 переменными/чем-то ещё.

ПРЕОБРАЗОВАНИЕ ТИПОВ

C++ позволяет определить функция оператора преобразования ИЗ типа текущего в другой (мб фундаментальный тип или тип класса). В общем случае формула такая:

```
Class MyClass
{
public:
    Operator OtherType const; //из MyClass в OtherType
.....................................................................
.....................
};
```

В отличие от остальных операторов, оперы преобразования должны быть ТОЛЬКО как функции-члены класса, единственный такой, у которого ключевому слову предшествует тип возвращаемого значения. Как обчные ф-ии нельзя определить.

Самый простой примерчик:

```
class Counter
{
public:
    Counter(int number)
    {
```

```
        value = number;
    }
    operator int() const { return value; }
private:
    int value;
};
```

А так используем:

```
Counter counter{25};
int n = counter;    // преобразуем от Counter в int - n=25
```

Благодаря оператору преобразования подобная конвертация типов выполняется неявно. Но преобразование типов также можно выполнять явно, например, с помощью функции static_cast:

```
Counter counter{25};

int n = static_cast<int>(counter); //явное преобразование Counter -> int
std::cout << n << std::endl; //25

//ИЛИ

int m {static_cast<int>(counter)};
std::cout << m << std::endl;
```

Еще один пример - преобразование в тип bool:

```cpp
#include <iostream>

class Counter
{
public:
    Counter(double n)
    {
        value = n;
    }
    void print() const
    {
        std::cout << "value: " << value << std::endl;
    }
    // Оператор преобразования в bool
    operator bool() const  { return value != 0; }
private:
    int value;
};
// тестируем операторы
void testCounter(const Counter& counter)
{
    counter.print();
    if (counter)
        std::cout << "Counter is non-zero." << std::endl;
    if (!counter)
        std::cout << "Counter is zero." << std::endl;
}

int main()
{
    Counter counter1{22};
    testCounter(counter1);

    Counter counter2{0};
    testCounter(counter2);
}
```

В данном случае в операторе преобразования, если значение value объекта Counter равно 0, возвращаем false, иначе возвращаем true:

```cpp
operator bool() const  { return value != 0; }
```

Благодаря чему мы можем использовать объект Counter в условных выражениях, подобно типу bool:

```cpp
if (counter)
if (!counter)
```

Стоит отметить, что хотя мы не определили явным образом оператор ! (оператор логического отрицания) для типа Counter, но выражение `!counter` будет успешно выполняться, потому что в данном случае объект `counter` будет неявно преобразован в `bool`.

Если преобразование должно быть явным, то можно отключить неявное вот так:

```cpp
explicit operator int() const { return value; } // Только явные преобразования
```

Преобразование между классами

Подобным образом можно выполнять преобразования между типами классов:

```cpp
#include <iostream>
class PrintBook;

// электронная книга
class Ebook
{
public:
    Ebook(std::string book_title)
    {
        title=book_title;
    }
    operator PrintBook() const;
    std::string getTitle(){return title;}
private:
    std::string title;
};
// печатная книга
class PrintBook
{
public:
    PrintBook(std::string book_title)
    {
        title=book_title;
    }
    operator Ebook() const;
    std::string getTitle(){return title;}
private:
    std::string title;
};

Ebook::operator PrintBook() const
{
    return PrintBook{title};
}
PrintBook::operator Ebook() const
{
    return Ebook{title};
}

int main()
{
    PrintBook book{"C++"};
    Ebook ebook{ book };    // оцифровываем книгу - из PrintBook в Ebook
    std::cout << ebook.getTitle() << std::endl;        // C++
    PrintBook print_book{ebook}; // распечатываем книгу из Ebook в PrintBook
    std::cout << print_book.getTitle() << std::endl;        // C++
}
```

ПЕРЕГРУЗКА ОПЕРАТОРОВ ВНЕ И ВНУТРИ КЛАССА

Вот такой пример с форума, пишите мне сразу, если это не то, что нужно, я поменяю:

В ТЗ моей работы указано, что перегрузка операторов должна выполняться как вне, так и внутри класса.

Я не совсем понимаю как это работает и хотел попросить Вашего совета. В моем понимании перегрузка оператора внутри класса - это:

```
class foo {
public:
  foo();
  foo operator + (foo arg);
}
```

Тогда как быть с выполнением вне класса..

c++ ooп классы перегрузка-операторов

Поделиться Улучшить вопрос
Отслеживать

изменён 13 мая 2022 в 8:38
Harry
226k • 15 • 125 • 240

задан 12 мая 2022 в 23:20
J1st
13 • 3

Добавить комментарий

ет

Сортировка: Наивысший рейтинг (по умолчанию)

У вас —

```
class foo{
public:
  foo();
  foo operator + (const foo& arg);
}
```

оператор объявлен в классе, как функция-член класса. Но его можно объявить и вне класса, как свободную функцию, просто записав его с двумя аргументами:

```
foo operator+(const foo& arg1, const foo& arg2)
{
    ...
}
```