




# Перегрузка операторов



**Для стандартных типов данных в языке С++ реализованы встроенные операторы: сложения (+), вычитания (-), умножения (\*), больше (>), меньше (<) и т.д.**

**Перегрузка операторов - позволяет заставить работать эти стандартные операторы с объектами пользовательских классов.**

**Для перегрузки оператора необходимо написать функцию, которая будет выполнять необходимое действие с объектами.**

# Что такое оператор?

Каждый оператор в C++ реализован как вызов функции со следующим прототипом:  
*Тип operator имя-оператора(список\_параметров)*

Поэтому `a = b+c`

ЭКВИВАЛЕНТНО

`operator = (a, operator + (b, c)) ;`

# Пример перегрузки операторов

**a << 2;**

- если **a** принадлежит одному из целочисленных типов, то это – оператор побитового сдвига влево (результат – **a\*4**, значение **a** не меняется;
- если **a** – выходной поток, в этот поток помещается десятичное представление числа 2. Поток **a** изменяется.

# Ограничения на перегрузку операторов

не могут быть перегружены операторы:

- . – доступ к члену класса;
- . \* – доступ к члену класса через указатель;
- :: – разрешения области видимости;
- ? : – условный оператор;
- #, ## – препроцессорные операторы;
- `static_cast`, `const_cast`, `dynamic_cast`, `reinterpret_cast` – операторы приведения типов (преобразования выражения из одного типа в другой);
- `sizeof` – определения размера;
- `typeof` – определения типа.


# Ограничения на перегрузку операторов

- нельзя вводить обозначения собственных операторов;
- не допускается перегрузка операторов для стандартных типов;
- при перегрузке операторов должно сохраняться число аргументов;
- не изменяются приоритеты перегруженных операторов;
- при перегрузке операторов нельзя задавать аргументы по умолчанию.

# Реализация перегрузки операторов при написании классов

Функция-оператор может быть определена как:

- метод класса
- дружественная функция класса
- обычная функция.



# Реализация перегрузки операторов при написании классов

- Один оператор может быть перегружен несколько раз;
- Если оператор перегружен как член класса (используется метод класса), то первым операндом по умолчанию является объект класса, для которого вызывается этот оператор.



# Операторы класса Point2D

- Сложение и вычитание точек (по правилам сложения и вычитания векторов);
- Унарный минус (по правилам векторов);
- Умножение точки на число и числа на точку (результат – растяжение или сжатие точки);
- Умножение двух точек (результат – скалярное произведение);
- Инкремент и декремент точки (результат – увеличение или уменьшение координат на 1);
- Сравнение на равенство и неравенство двух точек.

# Реализация унарных операторов

Если унарный оператор определяется как метод класса, он описывается без параметров (параметром считается вызвавший его объект), в противном случае параметром такой функции должна быть ссылка на объект класса (если операнд изменяется) или константная ссылка (в противном случае).

Функция должна возвращать ссылку на объект класса, если результат должен быть L-value. В противном случае результат зависит от сути функции.

# Примеры реализации унарных операторов для класса Point2D

## 1. Унарный минус:

```
class Point2D {  
...  
Point2D operator - () const;  
...  
};
```

```
Point2D Point2D::operator - () const {  
    return Point2D(-x, -y);  
}
```

# Примеры реализации унарных операторов для класса Point2D

## 2. Инкремент (префиксный) :

Отметим, что оператор префиксного инкремента возвращает новое значение, причем L-value

```
class Point2D {  
    ...  
    Point2D& operator ++ ();  
    ...  
};
```

```
Point2D& Point2D::operator ++ () {  
    x++;  
    y++;  
    return *this;  
}
```

# Примеры реализации унарных операторов для класса Point2D

## 3. Инкремент (постфиксный) :

Оператор постфиксного инкремента возвращает старое значение и не-L-value. Для различия постфиксного и инфиксного операторов добавляется фиктивный параметр.

```
class Point2D {  
...  
Point2D operator ++ (int);  
...  
}
```

```
Point2D Point2D::operator ++ (int a) {  
    Point2D p(*this);  
    x++;          y++;  
    return p;  
}
```

# Реализация бинарных операторов

Если бинарный оператор определяется как метод класса, он описывается с одним параметром, соответствующим второму операнду. Первым операндом считается вызвавший его объект. В противном случае такая функция должна иметь два параметра, соответствующих операндам оператора.

Функция должна возвращать ссылку на объект класса, если результат должен быть L-value. В противном случае результат зависит от сути функции.

# Примеры реализации бинарных операторов для класса Point2D

## 1. Сложение:

```
class Point2D {  
...  
Point2D operator + (const Point2D&) const;  
...  
};
```

```
Point2D Point2D::operator + (const Point2D& p) const  
{  
    return Point2D(x + p.x, y + p.y);  
}
```

# Примеры реализации бинарных операторов для класса Point2D

## 2. Скалярное произведение:

```
class Point2D {  
...  
double operator * (const Point2D&) const;  
...  
};
```

```
double Point2D::operator * (const Point2D& p) const  
{  
    return (x*p.x + y*p.y);  
}
```



# Примеры реализации бинарных операторов для класса Point2D

## 3. Сравнение :

```
class Point2D {  
...  
bool operator == (const Point2D&) const;  
bool operator != (const Point2D&) const;  
...  
};
```

```
bool Point2D::operator == (const Point2D& p) const  
{  
    return ((x==p.x) && (y==p.y));  
}  
bool Point2D::operator != (const Point2D& p) const  
{  
    return (! (this->operator==(p)));  
}
```

# Примеры реализации бинарных операторов для класса Point2D

## 4. Умножение на число :

```
class Point2D {  
...  
Point2D operator * (double) const;  
friend Point2D operator *(double, const Point2D&);  
...  
};
```

```
Point2D Point2D::operator * (double d) const  
{  
    return Point2D(d*x, d*y);  
}  
Point2D operator * (double d, const Point2D& p)  
{  
    return p.operator *(d);  
}
```

# Примеры реализации бинарных операторов для класса Point2D

## 4. Вывод в поток:

```
class Point2D {  
...  
friend ostream& operator <<(ostream&,  
                           const Point2D&);  
...  
};
```

```
ostream& operator <<(ostream& s, const Point2D& p)  
{  
    s << "(" << p.x << ", " << p.y << ")";  
    return s;  
}
```

# Перегрузка оператора присваивания

Для вновь создаваемых классов создаётся перегруженный *оператор присваивания по умолчанию*. Этот оператор копирует все поля объекта из левой части в соответствующие поля объекта из правой части.

Если этого достаточно, то собственную реализацию этого оператора можно не писать!

Для класса **Point2D** достаточно работы стандартного оператора присваивания. Для класса **Person** необходимо писать собственную реализацию, т.к. он захватывает ресурсы во время своего существования.

# Схемы работы перегруженного оператора присваивания $a = b$

- проверка на самоприсваивание;
- освобождение ресурсов, полученных объектом  $a$ ;
- получение ресурсов, закрепленных за объектом  $b$  (в случае, если ресурсом является динамическая память – клонирование)

# Реализация оператора присваивания для класса Person

```
class Person {  
...  
    const Person& operator =(const Person&);  
...  
};
```

```
const Person& Person::operator =(const Person& p)  
{  
    if (this == &p)  
        return *this;  
    delete [] name;  
    name = new char[strlen(p.name)+1];  
    strcpy(name, p.name);  
    return *this;  
}
```

# Методы Clone и Erase

Для эффективной реализации работы с ресурсами можно написать два защищённых метода:

- метод, освобождающий все занятые объектом ресурсы (**Erase**);
- метод, клонирующий ресурсы другого объекта (**Clone**)

```
class Person {  
private:  
    void Erase();  
    void Clone(const Person&);  
    ...  
}
```

# Реализация и использование методов Clone и Erase

```
void Person::Erase() {  
    delete [] name;  
}  
  
void Person::Clone(const Person& p) {  
    name = new char[strlen(p.name)+1];  
    strcpy(name, p.name);  
}  
  
Person::~~Person() {  
    Erase();  
}
```



# Реализация и использование методов Clone и Erase

```
Person::Person(const Person& p) : ID(++newID) {  
    Clone(p);  
}  
const Person& Person::operator=(const Person& p) {  
    if (this != &p) {  
        Erase();  
        Clone(p);  
    }  
    return *this;  
}
```

# Перегрузка оператора приведения типа

- Имеет следующий прототип:  
`operator тип()`
- Тип – задает тип данных, к которому приводится объект. Этот тип может быть как стандартным, так и пользовательским.
- Нет возвращаемого значения.

# Перегрузка оператора приведения типа

```
class Point2D {  
...  
operator double ();  
...  
};
```

```
Point2D::operator double() {  
    return Module();  
}
```

# Перегрузка оператора приведения типа

После написания этого кода возникают сложности с использованием других перегруженных операторов:

```
Point2D p(10,5);  
...  
cout << 3*p; //Ошибка C2666:  
// number overloads have similar conversions  
  
cout << operator*(3, p); //Ошибки нет  
  
cout << 3*(double)p; // Ошибки тоже нет
```

# Перегрузка оператора вызова функции

- Запись:

```
тип_возврата operator()  
(формальные_параметры) ;
```

- Класс, в котором определён хотя бы один оператор вызова функции, называется **функциональным классом**. Функциональный класс может не иметь других полей и методов.
- Функциональные классы используются в алгоритмах из библиотеки STL.

# Пример работы с функциональным классом

```
class IsBest {  
...  
    bool operator () (int a, int b) {  
        return (a<b || a%10==0);  
    }  
...  
};
```

```
IsBest best;  
...  
cout << best(10,4) << best(5,4);  
...  
int Mas[100];  
// заполняем массив  
sort(Mas, Mas+100, IsBest);
```



# Конец лекции