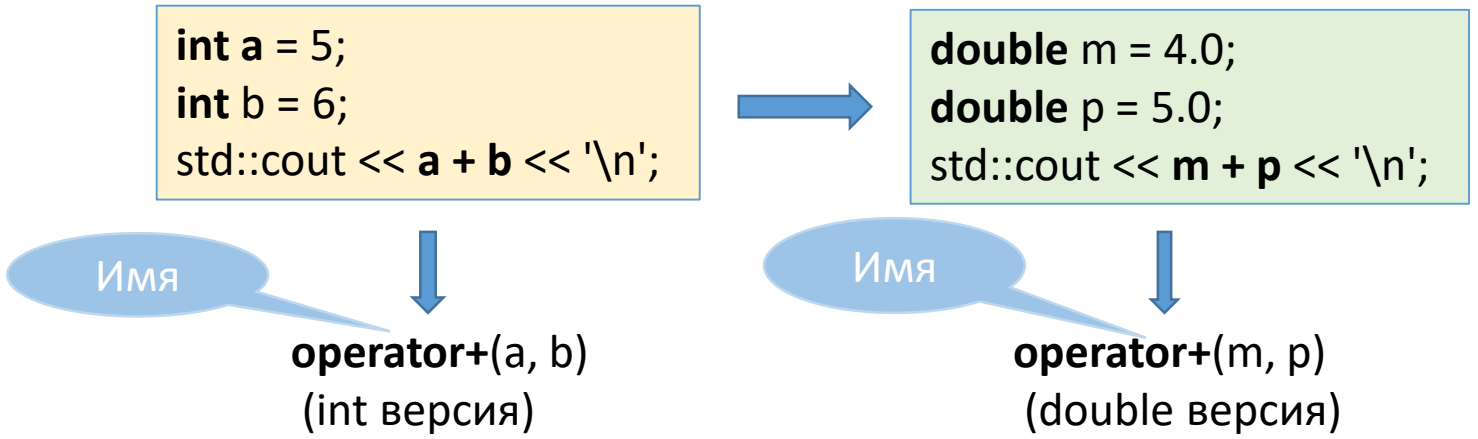
Перегрузка операторов в C++.

Лекция + практика

Операторы, как функции

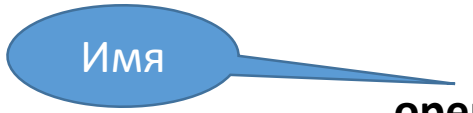
В языке C++ операторы реализованы в виде функций. Используя перегрузку функции оператора, вы можете определить свои собственные версии операторов, которые будут работать с разными типами данных (включая **классы**). Использование перегрузки функции для перегрузки оператора называется **перегрузкой оператора**.



Если все операнды фундаментальных типов данных, то вызываются встроенные соответствующие версии операторов (!если таковые существуют). Если таковых не существует, то компилятор выдаст ошибку.

Операторы для пользовательских типов

```
MyString hello = "Hello, ";
MyString world = "World!";
std::cout << hello + world << '\n';
```



operator+(hello, world)
(? версия)

- Если какой-либо из операндов - пользовательского типа данных , то компилятор будет искать версию оператора, которая работает с таким типом данных.
- Если компилятор не найдет ничего подходящего, то попытается выполнить конвертацию одного или нескольких операндов пользовательского типа данных в фундаментальные типы данных, чтобы использовать соответствующий встроенный оператор.
- Если это не сработает — компилятор выдаст ошибку.

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

Нельзя перегружать в C++:

- тернарный оператор **(?:);**
- оператор **sizeof**;
- оператор разрешения области видимости **(::)**;
- операторы выбора члена **.** и **.***.

- ✓ вы можете перегрузить только существующие операторы;
- ✓ по крайней мере один из операндов перегруженного оператора должен быть пользовательского типа данных;
- ✓ изначальное количество операндов, поддерживаемых оператором, изменить невозможно;
- ✓ все операторы сохраняют свой приоритет и ассоциативность по умолчанию

Способы перегрузки операторов.

При перегрузке бинарных операторов для работы с операндами разных типов, нужно писать две функции — по одной на каждый случай. (int, MyClass) и (MyClass, int)

Перегрузка через дружественные функции.

```
class A{
    int m_a;
  public:
    A(int a) {m_a = a;}
    // прототип
    friend A operator+(const A & a1, const A & a2);
};
  // реализация
A operator+(const A & a1, const A & a2){
    return A(a1.m_a + a2.m_a);
}
```

“Оператор +“

```
int main()
{
    A obj1(7);
    A obj2(3);
    A sum = obj1 + obj2;

return 0;
}
```

Перегрузка через обычные функции

```
class A{                                          "Оператор +"
    int m_a;
  public:
    A(int a) {m_a = a;}
    int getA() const {return m_a}; // геттер
};
// реализация
A operator+(const A & a1, const A & a2){
    return A(a1.getA() + a2.getA());
}
```

```
int main()
{
    A obj1(7);
    A obj2(3);
    A a_sum = obj1 + obj2;

return 0;
}
```

Перегрузка через методы класса.

```
class A{
                                                    "Оператор ="

    int m_a, m_b;
  public:        //констр. по умолчанию
    A(int a = 0, int b = 1): m_a(a), m_b(b){}
                // констр. копирования
    A(const A & copy): m_a(copy.m_a), m_b(copy.m_b){}
    A& operator= (const A & objA){
                    m_a = objA.m_a;
                    m_b = objA.m_b;
                    return *this;  // Возвращаем текущий объект
       }
};
```

```
int main()
{
    A obj1(1,7); // К. по умолч.
    A obj2;
    obj2 = obj1; // опер. присв.

 return 0;
}
```

Копирующая инициализация

Конструктор копирования должен быть задан явно. В нем должно быть описано: инициализация всех членов класса, выделение дин. памяти как под указатели так и под динамические переменные т.е. Описано “глубокое” копирование. **А так же должен быть перегружен оператор присваивания (=)**

Конструктор преобразования

Конструкторы, которые используются в неявных преобразованиях, называются **конструкторами преобразования**

```
// Конструктор по умолчанию
Drob(int numerator = 0, int denominator = 1) :
m_numerator(numerator), m_denominator(denominator)
{
assert(denominator != 0);
}
// Конструктор копирования
Drob(const Drob &copy) :
m_numerator(copy.m_numerator),
m_denominator(copy.m_denominator)
{
std::cout << "Copy constructor worked here!\n"; // просто, чтобы показать, что это работает
}
```

Явные конструкторы (с ключевым словом **explicit**) не используются для неявных конвертаций

```
Drob makeNegative(Drob d)
{
        d.setNumerator(-d.getNumerator());
        return d;
}
int main(){
        std::cout << makeNegative(7); // передаем целочисленное значение
        return 0;
}
```

-> Copy constructor worked here!
-7/1

У класса Drob есть конструктор, который может принимать одно целочисленное значение (конструктор по умолчанию), поэтому компилятор выполнит **неявную конвертацию литерала 7** в объект класса **Drob**. Это делается путем выполнения копирующей инициализации параметра **d** функции makeNegative() с помощью конструктора Drob(int, int).

Поверхностное копирование

Конструктор копирования и оператор присваивания, которые C++ предоставляет по умолчанию, используют **поверхностное копирование**. Это означает, что C++ выполняет копирование для каждого члена класса индивидуально. (Если нет членов с дин. выделенной памятью то все в порядке.)

При поверхностном копировании указателя копируется только адрес указателя — никаких действий по содержимому адреса указателя не предпринимается.

При выполнении поверхностного копирования, два указателя будут содержать в себе адрес одного и того же участка памяти!

Если объект-копия выходит из области видимости, то вызывается деструктор для этой копии. Он освобождает динамически выделенную память, на которую указывает копия и объект с помощью которого копия была создана! Следовательно, удаляя копию, мы также (случайно) удаляем и данные первоначального объекта. Объект сору затем уничтожается, но первоначальный объект остается указывать на освобожденную память!

Глубокое копирование

При глубоком копировании память сначала выделяется для копирования адреса, который содержит исходный указатель, а затем для копирования фактического значения.

Таким образом копия находится в отдельной, от исходного значения, памяти и они никак не влияют друг на друга. Для выполнения **глубокого копирования** нам необходимо написать свой собственный **конструктор копирования** и перегрузить **оператора присваивания**.

П1. Перегрузка операторов вывода

```
int main(){
    Point point1(5.0, 6.0, 7.0);
    std::cout << point1;
    return 0;
}
-> You entered: Point(4, 5.5, 8.37)
```

Цель

Оператор **<<** -бинарный,
Операнды : **point** – тип Point,
std::cout – тип std::ostream

Описание оператора

Рабочий класс

```
class Point{
private:
    double m_x, m_y;
public:
    Point(double x=0.0, double y=0.0)
            :m_x(x), m_y(y){}
  // место для прототипа друж. Ф-ции
};
```

```
std::ostream& operator<< (std::ostream &out, const Point &point)
{
    out << "Point(" << point.m_x << ", " << point.m_y << ")";
    return out;
}
```

Реализация перегрузки

прототип

```
friend std::ostream & operator<<(std::ostream &out, const Point &point);
```

П2. Перегрузка унарных операторв

```
int main(){
    const A a1(7);
    std::cout << (-a1).getA();
    return 0;
}
```

Цель

Описание оператора

Оператор (-)-унарный,
Операнд : a1 – тип A,

Перегружается только методом класса !

Рабочий класс

```
class A{
private:
    int m_a;
public:
    A(int a=0):m_a(a)
   { }
    int getA() const { return m_a; }
    A operator- ()const;
};
```

```
A A :: operator- () const
 {
      return A(-m_a);
}
```

Реализация перегрузки

ПЗ. Перегрузка операторов сравнения

```
int main(){
    Car car1(“Ford”,”Mustang”);
    Car car2(“KIA”,”RIO”);
    if (car1 == car2)
      std::cout << “cars the same !”;
    return 0;
}
```

Цель

Оператор == -бинарный,
Операнды : car1 – тип Car,
car2 – тип Car
Перегружается дружественной или обычной функцией!

Описание оператора

```
class Car{
private:
    std::string m_comp;
    std::string m_model;
public:
    Car(std::string m_comp std::string m_model)
            :m_comp(comp), m_model(model){}
friend bool operator== (const Car &c1, const Car &c2);
};
```

Рабочий класс

```
bool operator== (const Car &c1, const Car &c2)
{
    return (c1.m_company == c2.m_company &&
            c1.m_model== c2.m_model);
}
```

Реализация перегрузки

прототип

П4.Перегрузка инкремента/декремента

```
int main()
{
   N number(7);
   std::cout << number;
   std::cout << --number;
   std::cout << --number;
   std::cout << number++;
   return 0;
}
```

Цель

Оператор (- -)-унарный,
Оператор (++)-унарный,
Операнд : number – тип N,
Перегружается только методом класса !

Описание оператора

```
N & N:: operator- - ()
{
    if(m_n == 0)
       m_n = 8;
    else   - -m_n;
   return * this;
}  // префикс
```

```
N N:: operator++ (int)
{
     N tmp (m_n);
      ++(*this);
 return tmp;
}  // постфикс
```

Реализация перегрузки

```
class N{
   int m_n;
public:
   N(int n=0):m_n(n)
  { }
   N & operator- -();  // префикс
   N operator ++(int); // постфикс
};
```

Рабочий класс

Чтобы различить при перегрузке **пре-** и **пост-** версию, C++ использует **фиктивную переменную для операторов версии постфикс.**

Операторы версий префикс и постфикс выполняют одно и то же задание: оба увеличивают/уменьшают значение переменной объекта. Разница между ними только в значении, которое они возвращают.
Префикс – возвращает результат после изменения параметра.
Постфикс – возвращает результат перед изменением параметра (для этого исп. **tmp**)

П5. Перегрузка оператора ()

Оператор () является особенно интересным, поскольку позволяет изменять как тип параметров, так и их количество.

Перегрузка круглых скобок должна осуществляться через **метод класса**.
В случае с классами перегрузка круглых скобок выполняется в методе **operator()(){}** (в объявлении функции перегрузки находятся две пары круглых скобок).

Перегрузка оператора () используется в реализации функторов (или «функциональных объектов») — классы, которые работают как функции.
Преимущество функтора над обычной функцией заключается в том, что функторы могут хранить данные в переменных-членах

```
class Accumulator{
private:
    int m_counter = 0;
public:
    Accumulator()
    {}
    int operator() (int i) { return (m_counter += i); }
};
```

Рабочий класс

```
int main()
{
    Accumulator accum;
    std::cout << accum(30) << std::endl; // выведется 30
    std::cout << accum(40) << std::endl; // выведется 70
    return 0;
}
```

Цель

Реализация перегрузки

Использование **Accumulator** выглядит как вызов обычной функции, объект Accumulator может хранить значение, которое увеличивается

П6.Перегрузка оператора[]

```
int main(){
    IntArray array;
    array[4] = 5; // присваиваем значение
    std::cout << array[4]; // выводим значение
   return 0;
}
```

Цель

Оператор []-унарный,
Операнд : number – тип int,
Перегружается только методом класса !

```
class IntArray
{
private:
   int m_array[10];
public:
   int& operator[] (const int index);
};
int& IntArray::operator[] (const int index)
{
   return m_array[index];
}
```

Рабочий класс

Реализация перегрузки

Домашняя работа.

- Перегрузка оператора ввода (>>)
- Перегрузка оператора ввода (>)
- Перегрузка инкремента/декремента(++x/x--)
- Перегрузка оператора[] при работе с указателем.
- Перегрузка операций преобразования типов данных
- Перегрузка оператора присваивания (=)

Источники информации

- https://ravesli.com/uroki-cpp/#toc-0 (через VPN)