Языки программирования. Часть 2
Лекция 2

ПМИ

Демяненко Я.М.

2023

Перегрузка операций

Перегрузка бинарной операции

Перегрузка операции это описание операции с тем же именем, но работающей с другими типами.

@ - обозначение бинарной операции в рамках курса

Существует 2 способа

1.Как **функцию-член**
```
a.operator@(b)
```

2.Как **внешнюю функцию**
```
operator@(a, b)
```

Реализовывать необходимо **один из двух**.
При попытке реализовать оба варианта компилятор выдаст ошибку.

Перегрузка операции +=

```
class Date{
    private:
    void add_days(int n);
    ...
    public:
    void operator+=(int n){// inline
        add_days(n);
    }
};
```

```
d += 7;
d.operator+=(7);
//d.add_days(7); private
```

Перегрузка операции сравнения на равенство

```
class Date {
    int d, m, y;

public:
    //bool operator==(Date d1, Date d2) типичная ошибка

    bool operator==(Date const & other)   {
        return d == other.d &&
               m == other.m &&
               y == other.y;
    }
};
```

Если операция бинарная и определена функцией-членом , то у нее будет один аргумент.

Задание

Попробовать убрать скобки в операторе вывода в поток

```
cout<<(a==b)<<endl;
```

Необходимы ли они?
В любом случае объяснить, почему.

Перегрузка операции != (как внешняя функция)

```
class Date {
    …
    public:
    …
    friend bool operator!=(const Date &d1, const Date &d2)    {
        return !(d1==d2);
    }
};
```

Как видим, использование ранее определенной операции == позволяет не обращаться к внутренним полям класса напрямую.

Так как описанная функция является inline, и если и операция != тоже inline, то в результате будет произведена следующая замена.

```
d1!=d2 ~ !(d1==d2) ~ !(d1.d == d2.d && d1.m == d2.m && d1.y == d2.y)
```

Перегрузка операции != (как член класса)

```
class Date {
    …
    public:
    …
    bool operator!=(const Date &d2){
        return !(*this==d2);
    }
};
```

Ограничения перегрузки

Хотя C++ позволяет перегружать **почти все** операции, доступные в C++, **возможности** перегрузки **ограничены**.

В частности, **отсутствует** возможность **изменения приоритета** или **количества аргументов** у операций.

Кроме того, рекомендуется **не изменять семантику** операций.

Аргументы перегруженной операции

Количество аргументов в списке перегруженной операции зависит от двух факторов:

- категории операции – **унарная или бинарная**;
- способа определения операции – **в виде глобальной (внешней) функции** (один аргумент для унарной, два – для бинарных операций) **или функции класса** (для унарных операций аргументы отсутствуют, для бинарных операций – один аргумент).

При определении функции-члена класса объект, для которого она будет вызываться, всегда будет её левым операндом.

Перегрузка операции вывода в поток (чтения из потока)

operator<<(>>)

Функция-член vs Внешняя функция

```
/* date.h */
class Date {
    int d, m, y;
public:
    // …
    friend ostream & operator<<(ostream & os, const Date & d);
};

/* date.cpp */
ostream & operator<<(ostream & os, const Date & d) {
    os << d.d << '.' << d.m << '.' << d.y << '\n';
    return os;
}
```

Функция `operator<<` должна возвращать ссылку на полученный объект потока для допустимости цепочек типа

```
cout << d1 << d2;
```

Операция чтения из потока определяется аналогичным образом, только вместо `ostream` используется `istream`.

Более короткий вариант

Т.к. operator<< небольшой, то его лучше сделать inline. Перенесем реализацию в заголовочный файл.

```
/* date.h */
class Date {
    int d, m, y;
public:
    // …
    friend ostream & operator<<(ostream & os, Date const & d){
        return os << d.d << '.' << d.m << '.' << d.y << '\n';
    }
};
```

Если друг класса определен прямо в классе, то он остается внешней функцией и становится inline.

Перегрузка арифметических операций

Пусть @ ∈ {+, -, *, /}

Обычно определяют пару функций: `operator@=`, `operator@`

В таком случае `operator@=` определяется как функция-член,

а `operator@` — или как внешняя функция или как член-функция

Операция @= называется присваивающей формой операции @

Перегрузка арифметических операций класса **BigInteger**

```
class BigInteger {
    int data[1024];
public:
    BigInteger & operator+=(const BigInteger & other)  {
        // Цикл по data: суммирование и перенос
        // Возвращаем ссылку на себя
        return *this;
    }
};

BigInteger operator+(BigInteger const & bi1, BigInteger const & bi2) {
// функция не может возвращать ссылку, потому что возвращает новый объект
    BigInteger res(bi1);  // копия bi1
    res += bi2;           // res хранит bi1 + bi2
    return res;
}
```

Если сделать operator+ **дружественной**

```
class BigInteger {
    int data[1024];
public:
  BigIntiger & operator+=(BigInteger cons & other)  {
        // Цикл по data: суммирование и перенос
        // Возвращаем ссылку на себя
        return *this;
    }
  friend BigInteger operator+(BigInteger const & bi1, BigInteger const & bi2){
    // функция не может возвращать ссылку, потому что возвращает новый объект
       BigInteger res(bi1);          // копия bi1
       res += bi2;                  // res хранит bi1 + bi2
       return res;
    }

};
```

Если сделать operator+ **членом класса**

```
class BigInteger {
    int data[1024];
public:
    BigIntiger & operator+=(BigInteger cons & other)  {
        // Цикл по data: суммирование и перенос
        // Возвращаем ссылку на себя
        return *this;
    }
    BigInteger operator+(BigInteger const & bi2);
};

BigInteger BigInteger::operator+(BigInteger const & bi2) {
    //функция не может возвращать ссылку, потому что возвращает новый объект
    BigInteger res(*this); //копия того объекта, для которого вызвана функция
    res += bi2;            //res хранит bi1 + bi2
    return res;
}
```

Перегрузка унарных операций: @a ++a --a (общий случай)

Префиксная унарная операция может быть определена двумя способами:

- в виде функции-члена: `a.operator@()`
- в виде внешней функции: `operator@(a)`

Операции ++d для класса Date

```
class Date {
    private:
    …
    void add_days(int a);
    public:
    …
    // определение префиксного ++
    Date & operator++()  {
        add_days(1);
        return *this;
    }
};
```

Как разделить a++ и ++a ?

При перегрузке префиксной и постфиксной унарных операций встает вопрос, об интерпретации записи типа operator@.

В 1998 году для разделения был введен фиктивный параметр типа int.

Постфиксная унарная операция a@ a++ a--

Постфиксная унарная операция также может быть определена двумя способами:

- в виде функции-члена `a.operator@(int)`
- в виде внешней функции `operator@(a, int)`

Операции d++ для класса Date

```
class Date {
    private:
    …
    void add_days(int a);
    public:
    …
    // определение постфиксного ++
    // d++ возвращает значение до увеличения,
    // значит возвращать нужно копию, а не ссылку.
    Date operator++(int) {
        Date d = *this;
         //Date d = Date (*this);
        add_days(1); // ~ ++(*this);
        return d;
    }
};
```

i++ vs ++i

Из-за копирования всего объекта операция d++ будет выполняется дольше, поэтому надо выбирать в пользу ++d.

Это верно для своих типов, но для встроенных типов компилятор производит оптимизацию.

В результате записи i++ и ++i, например, для переменных типа int, равноценны.

Автоматическая генерация конструктора копии

Копирующие конструкторы настолько важны, что компилятор **автоматически генерирует копирующий конструктор**, если этого не делает программист.

Такой автоматически создаваемый конструктор копии является **конструктором с побитовым копированием** переменных-членов класса.

Такое **копирование** называется **поверхностным**.

Три случая, когда вызывается **конструктор копий**

- Явное создание нового объекта-копии:

```
myvector myv3(mv1);
myvector myv2 = myv1;
```

- Вызов функции с передачей параметра по значению:

```
void f(myvector st) {/* … */}
```

- Возврат объекта из функции по значению:

```
myvector g() {/* … return …;*/}
```

НЕ КОНСТРУКТОР КОПИИ

```
myvector mv2;

// операция присваивания
mv2=mv1;
```

Вопрос: сколько будет вызвано конструкторов копий?

```
class myvector {
    …
public:
};

myvector g() {
   return myvector();
 }

int main() {
    myvector myv1 = g();
}
```

Явное создание нового объекта-копии
Вызов функции с передачей параметра по значению
Возврат объекта из функции по значению

Ответ:

```
class myvector {
    …
public:
};

myvector g() {
   return myvector();
 }

int main() {
    myvector myv1 = g();
}
```

здесь присутствуют случаи 1 и 3 вызова конструктора копий, а значит должны создаваться две копии.

Явное создание нового объекта-копии
Вызов функции с передачей параметра по значению
Возврат объекта из функции по значению

Return Value Optimization (RVO)

На самом деле, запуск данного примера покажет, что во время выполнения программы не будет вызвано ни одного конструктора копий.

Это результат работы оптимизирующего компилятора.

Заметим, что такая оптимизация может существенно повлиять на поведение программы в случае, когда конструктор копии содержит побочные эффекты (например: вывод на консоль).

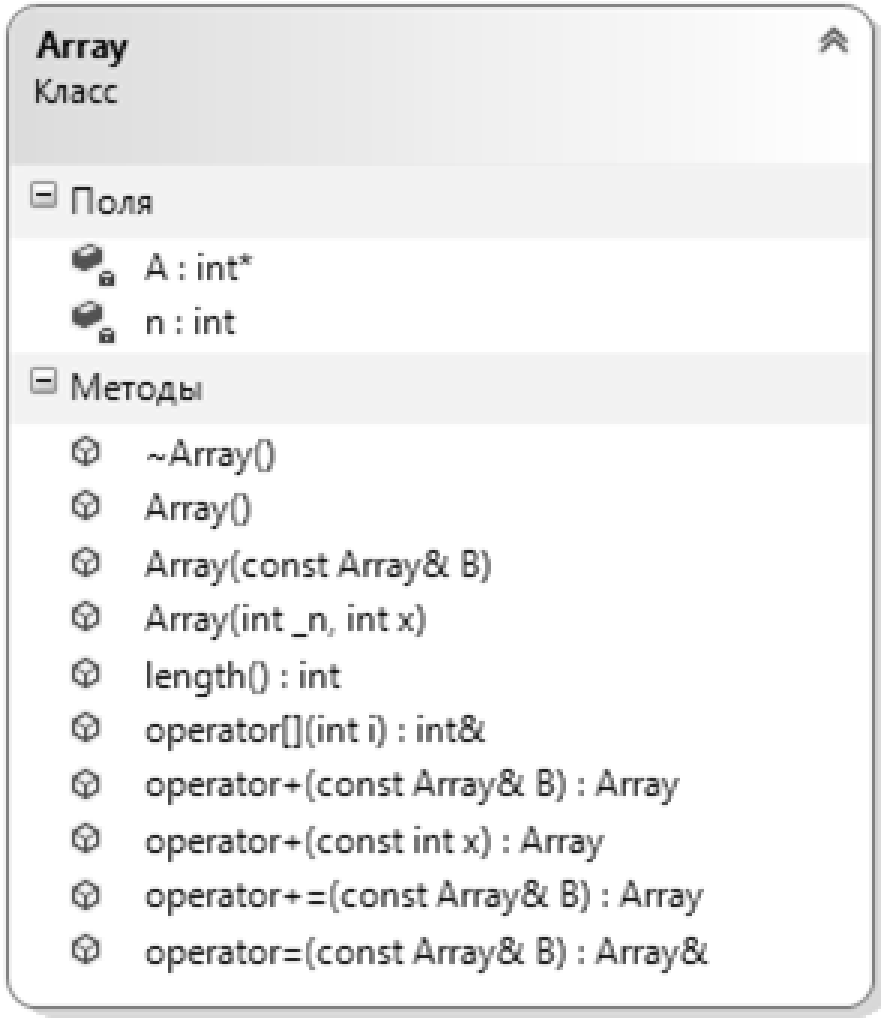
Однако она производится подавляющим большинством современных компиляторов по умолчанию, потому что явно оговорена в стандарте языка.

Эта оптимизация носит название Return Value Optimization (RVO).

Если специальными ключами компиляции запретить RVO, то будут вызвано ровно два конструктора копии, как и ожидалось.
В **реальных программах** просто стараются **не помещать дополнительный код в конструктор копии**.

Пример. Реализовать класс – динамический массив целых чисел.

Перегрузить операции:
+ для двух массивов одинакового размера;
+ для массива и целого числа;
+= для двух массивов одинакового размера;
присваивания;
обращение к элементу по индексу;
вывода в поток



UML диаграмма класса Array

```
class Array {
private:
  int n;
  int *A;
public:
  Array();                        //конструктор по умолчанию
  Array(int _n, int x = 0);       //конструктор с параметром по умолчанию
  Array(const Array &B);          //конструктор копии
  int length() const;             //функция для нахождения размера массива
  void resize(int nsize);         //изменение размера массива
  Array operator + (const Array &B);
  Array &operator += (const Array &B);
  Array operator + (const int x);
  Array &operator = (const Array &B);
  int& operator [] (int i);
  int operator [] (int i) const;
  ~Array();
  friend ostream & operator << (ostream &out, const Array &B);
};
```

```
int Array::length() const{
  return n;
}


void Array::resize(int nsize) {
  int * ndata = new int[nsize];
  int sz = (n < nsize) ? n : nsize;
  for (int i=0;i<sz;++i)
    ndata[i] = A[i];
  delete[] A;
  A = ndata;
  n = nsize;
}
```

Перегрузка операции []

Её нужно реализовывать **только** как **функцию-член класса**.
Важно, чтобы перегруженная операция доступа к элементу массива по индексу [ ] возвращала ссылку на элемент массива. Это обусловлено требованиями к соблюдению семантики.

```
int& Array::operator [] (int i){
  return A[i];
}
```

```
for (int i = 0; i < Q.length(); ++i)
    Q[i] = Q.length() - 1 - i;
  cout << "изменённый Q " << Q;
  cout << "срединные элементы массива Q " << Q[Q.length() / 2 - 1] << ' '
                                          << Q[Q.length() / 2] << endl;
  for (int i = 0; i < E.length(); ++i)
    E[i] = i;
```

Константные функции-члены

Возможно, операцию индексирования потребуется использовать в функциях, в которые объект класса Array передаётся как константный параметр по ссылке.

```
int printAndSum (const Array &v) {
  int sum=0;
  for (int i = 0; i < v.length(); i++){
    cout << v[i] << ' ';
    sum+=v[i];
  }
  return sum;
}
```

При компиляции этой функции возникнет ошибка
«IntelliSense: отсутствует оператор "[]", соответствующий этим операндам: const Array[int]»

Для корректной компиляции необходимо определить вторую функцию перегрузки операции индексирования.

```
int Array::operator [] (int i) const {
  return A[i];
}
```

Деструктор

Если в классе используется выделение памяти, то её необходимо освободить средствами этого же класса. Это должен делать деструктор.

```
Array::~Array( ){
  delete[] A;
}
```

Напомним, что деструктор вызывается компилятором автоматически для каждого созданного объекта.

Операция +

Для массива и целого числа

Операция некоммутативна

```
Array Array::operator+(const int x){
  Array C(n);
  //можно: Array C(n,0);
  for (int i = 0; i<n; i++)
    C.A[i] = A[i] + x;
  return C;
}
```

Для двух массивов

Операция коммутативна

```
Array Array::operator+(const Array &B){
  if (n != B.n)
    throw (1);
  Array C(n);
  //можно: Array C(n,0);
  for (int i=0; i<n; i++)
    C.A[i]=A[i]+B.A[i];
  return C;
}
```

Использование операции +

```
Array Z(15), Y(15);
  try {
          cout << "сумма " << Z+Y;
          cout<<Z+5;
          //cout<<5+Z;  //error
  }
  catch (int) {
    cout << "Массивы имеют разную длину" << endl;
  }
```

Операция +=

```
Array& Array::operator += (const int k){
  for (int i=0; i<n; i++)
    A[i]=A[i]+k;
  return *this;
}
```

```
Array& Array::operator += (const Array &B){
  if (n != B.n)
    throw (1);
  for (int i=0; i<n; i++)
    A[i]=A[i]+B.A[i];
  return *this;
}
```

Использование операции +=

```
Array Z(15), Y(15);
  try {
          Z+=Y;
          cout << " новый Z= " << Z;
  }
  catch (int) {
    cout << "Массивы имеют разную длину" << endl;
  }

Z+=5;
          cout << " новый Z= " << Z;
```

Операция + через операцию +=

```
Array Array::operator + (const Array &B){
  Array C(n);
  for (int i=0; i<n; i++)
    C.A[i]=A[i]+B.A[i];
  return C;
}
```

```
 Array Array::operator + (const Array &B){
   Array C(*this);
   C+=B;
 return C;
 }
```

Варианты использования конструкторов

```
Array Q(10,5);  //конструктор с двумя параметрами

Array P(10);    //конструктор с параметром по умолчанию. Второй параметр задан неявно

Array W,E;      //конструктор по умолчанию

Array R(Q);     //конструктор копии
```

Зачем создавать конструктор копии?

Допустим такое определение класса

```
class myvector {
    int size;
    int * data;
    string name;

public:
    myvector(string const & name = "id1"): size(8), name(name) {
        data = new int [size];
        cout << name << " created\n";
    }

    ~myvector()     {
        delete [] data;
        cout << name << " killed\n";
    }
};
```

Ожидание ⟶ Реальность

Проверим работу класса, выполнив следующий код в функции main:

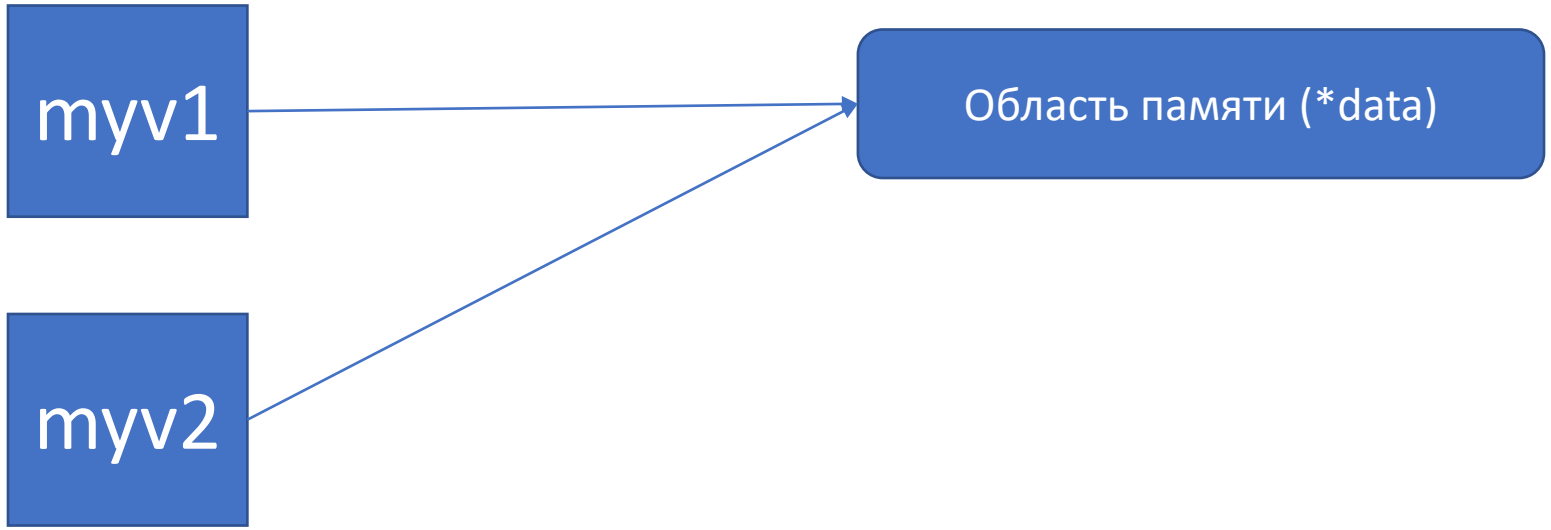
```
myvector myv1;
myvector myv2 = myv1;
```

В строке `myvector myv2 = myv1;` мы подразумеваем, что происходит копирование объекта `myv1`, а в конце выполнения функции `main()` — удаление двух объектов: `myv1` и `myv2`.

В действительности, при запуске программы происходит ошибка:

id1 created
id1 killed
*** Error in `./main': double free or corruption (fasttop): …

Что происходит в случае использования конструктора копии, создаваемого автоматически?



Вместо копии массива будет создана копия ссылки на него, т. е. ссылки двух разных объектов будут указывать на одно и то же место в памяти.