

Предефиниране на оператори.

Копиращ конструктор.

Оператор за присвояване

Любомир Чорбаджиев

Технологическо училище "Електронни системи"

Технически университет, София

lchorbadjiev@elsys-bg.org

*Revision : 1.4*

21 февруари 2005 г.

## Пример: Операции с вектори

- Основните операции, които могат да се извършват с вектори, са **събиране**, **изваждане** и **умножение по число**.
- Нека разгледаме вектори, дефинирани в равнината. Всеки вектор може да се представи като двойка числа  $\vec{a} = (a_x, a_y)$ , където  $a_x$  и  $a_y$  са съответно  $x$  и  $y$ -координатата на вектора  $\vec{a}$ .

## Пример: Операции с вектори

- Нека са дадени два вектора  $\vec{a} = (a_x, a_y)$  и  $\vec{b} = (b_x, b_y)$ . Операцията **събиране на вектори** дава нов вектор  $\vec{c} = (c_x, c_y)$ , такъв че:

$$c_x = a_x + b_x, c_y = a_y + b_y$$

- Нека са дадени два вектора  $\vec{a} = (a_x, a_y)$  и  $\vec{b} = (b_x, b_y)$ . Операцията **изваждане на вектори** дава нов вектор  $\vec{c} = (c_x, c_y)$ , такъв че:

$$c_x = a_x - b_x, c_y = a_y - b_y$$

- Нека се дадени вектор  $\vec{a} = (a_x, a_y)$  и число  $\alpha$ . Операцията **умножение на вектор по число** дава нов вектор  $\vec{b} = (b_x, b_y)$ , такъв че:

$$b_x = \alpha a_x, b_y = \alpha a_y$$

2

## Пример: Операции с вектори

- Нека дефинираме клас Point, който представя **вектор в равнината**.

```
4 class Point {
5     double x_, y_;
6 public:
7     Point(double x=0, double y=0)
8         : x_(x), y_(y)
9     {}
10    double get_x() const {return x_;}
11    double get_y() const {return y_;}
12
13    Point& add(const Point& p);
14    Point& sub(const Point& p);
15    Point& mul(double a);
16 };
```

3

## Пример: Операции с вектори

- Методът `Point& add(const Point& p)` реализира операцията събиране на вектори.

```
18 Point& Point::add(const Point& p) {  
19     x_+=p.x_;  
20     y_+=p.y_;  
21     return *this;  
22 }
```

- Нека са дадени два вектора  $\vec{p}_1$  и  $\vec{p}_2$ . Операцията  $\vec{p}_1 = \vec{p}_1 + \vec{p}_2$  може да се изпълни по следният начин:

```
Point p1, p2;  
//....  
p1.add(p2);
```

4

## Пример: Операции с вектори

- Методът `Point& sub(const Point& p)` реализира операцията изваждане на вектори.

```
23 Point& Point::sub(const Point& p) {  
24     x_-=p.x_;  
25     y_-=p.y_;  
26     return *this;  
27 }
```

- Нека са дадени два вектора  $\vec{p}_1$  и  $\vec{p}_2$ . Операцията  $\vec{p}_1 = \vec{p}_1 - \vec{p}_2$  може да се изпълни по следният начин:

```
Point p1, p2;  
//....  
p1.sub(p2);
```

5

## Пример: Операции с вектори

- Методът `Point& mul(double alpha)` реализира операцията умножение на вектор по число.

```
28 Point& Point::mul(double alpha) {
29     x_*=alpha;
30     y_*=alpha;
31     return *this;
32 }
```

- Нека е даден вектор  $\vec{p}$  и числото  $\alpha$ . Операцията  $\vec{p} = \alpha\vec{p}$  може да се изпълни по следният начин:

```
Point p;
double alpha;
//....
p.mul(alpha);
```

6

## Пример: Операции с вектори

- И трите разгледани метода връщат препратка към `Point`, като тази препратка препраща към обекта, върху който се изпълнява операцията (`*this`).

```
18 Point& Point::add(const Point& p) {
19     x_+=p.x_;
20     y_+=p.y_;
21     return *this;
22 }
```

- Това позволява тези операции да се прилагат последователно върху даден обект:

```
1 Point p1, p2, p3;
2 //...
3 p1.add(p2).sub(p3).mul(10.0);
```

- Ред 3 е еквивалентен на следният код:

```
1 p1.add(p2);
2 p1.sub(p3);
3 p1.mul(10.0);
```

7

# Пример: Операции с вектори

```
1 #include <iostream>
2 using namespace std;
3
4 class Point {
5     double x_, y_;
6 public:
7     Point(double x=0, double y=0)
8         : x_(x), y_(y)
9     {}
10    double get_x() const {return x_;}
11    double get_y() const {return y_;}
12
13    Point& add(const Point& p);
14    Point& sub(const Point& p);
15    Point& mul(double a);
16 };
17
```

8

```
18 Point& Point::add(const Point& p) {
19     x_ += p.x_;
20     y_ += p.y_;
21     return *this;
22 }
23 Point& Point::sub(const Point& p) {
24     x_ -= p.x_;
25     y_ -= p.y_;
26     return *this;
27 }
28 Point& Point::mul(double alpha) {
29     x_ *= alpha;
30     y_ *= alpha;
31     return *this;
32 }
33
```

```

34 int main(void) {
35     Point p1(1.0,1.0);
36     Point p2(2.0,2.0);
37     Point p3(3.0,3.0);
38
39     p3.add(p2).sub(p1).mul(10.0);
40
41     cout<<"p3="
42         <<p3.get_x()<<" ,□"
43         <<p3.get_y()<<" "<<endl;
44     return 0;
45 }

```

```

lubo@kid:~/school/notes> ./a.out
p3=(40, 40)

```

## Предефиниране на оператори

- Представената реализация на векторна аритметика е удобна, но щеше да бъде много по удобна, ако можем да използваме естествените математически оператори +, -, \*, +=, -=, \*=. Например:

```

1 Point p1, p2, p3;
2 //...
3 p1=p2+p3;
4 p1*=10.0;
5 p3-=p3;

```

- Една от важните концепции при създаването на езика C++ е, че класовете, трябва да бъдат равноправни на вградените (примитивни) типове.

## Предефиниране на оператори

- В езика C++ е предвидена възможност операторите да бъдат дефинирани за потребителските типове.
- Има само няколко оператора, които не могат да се предефинират от потребителя:
  - ◇ `::` – оператор за избор на област на видимост;
  - ◇ `.` – оператор за избор на член;
  - ◇ `.*` – оператор за избор на член чрез указател към член;
  - ◇ `sizeof` – оператор за размер на обект;
  - ◇ `typeid` – оператор за идентификация на типа;
  - ◇ `?:` – оператора за условен избор;
- Всички останали оператори могат да се предефинират.

10

## Бинарни и унарни оператори

- **Бинарен** оператор се нарича оператор, който действа върху два аргумента. **Унарен** е оператор, който действа върху един аргумент.
- Примери за бинарни оператори са операторите `+` (`a+b`), `*` (`a*b`), `-` (`a-b`), `/` (`a/b`) и т.н.
- Примери за унарни оператори са операторите `-` (`-a`), `!` (`!a`), `~` (`~a`), `++` (`a++`) и т.н.
- Вида на оператора определя начина, по който той може да бъде предефиниран.

11

## Бинарни оператори

- Бинарните оператори могат да се дефинират по два начина:

- ◇ Като нестатична член-функция на класа, която приема един аргумент – например:

```
Point Point::operator+(const Point& p)
```

- ◇ Като функция, която не е член на класа и приема два аргумента – например:

```
Point operator+(const Point& p1, const Point& p2)
```

12

## Бинарни оператори

- Нека разгледаме първият вариант за предефиниране на бинарен оператор. За пример ще използваме класът Point и бинарният оператор за събиране:

```
1 #include <iostream>
2 using namespace std;
3
4 class Point {
5     double x_, y_;
6 public:
7     Point(double x=0, double y=0)
8         : x_(x), y_(y)
9     {}
10    double get_x() const {return x_;}
11    double get_y() const {return y_;}
12    Point operator+(const Point& p) const;
13 };
```

13



```

14 Point Point::operator+(const Point& p) const {
15     Point result(get_x()+p.get_x(),get_y()+p.get_y());
16     return result;
17 }
18 int main(void) {
19     Point p1(1.0,1.0), p2(2.0,2.0), p3;
20
21     p3=p1+p2;
22     cout<<"p3=("
23         <<p3.get_x()<<" ,"
24         <<p3.get_y()<<")"<<endl;
25     return 0;
26 }

```

```
lubo@kid:~/school/notes> ./a.out
```

```
p3=(3, 3)
```

- Изразът в ред 21 е еквивалентен на следното:

```
p3=p1.operator+(p2);
```

## Бинарни оператори

- Нека разгледаме вторият вариант за предефиниране на бинарен оператор. Като пример отново използваме класът Point:

```

1 #include <iostream>
2 using namespace std;
3
4 class Point {
5     double x_, y_;
6 public:
7     Point(double x=0,double y=0
8         : x_(x), y_(y)
9         {}
10    double get_x() const {return x_;}
11    double get_y() const {return y_;}
12 };
13 Point operator+(const Point& p1, const Point& p2) {
14     Point result(p1.get_x()+p2.get_x(),
15                 p1.get_y()+p2.get_y());
16     return result;
17 }

```

```

18 int main(void) {
19     Point p1(1.0,1.0), p2(2.0,2.0), p3;
20     p3=p1+p2;
21
22     cout<<"p3=( "
23         <<p3.get_x()<<" ,□"
24         <<p3.get_y()<<" )"<<endl;
25     return 0;
26 }

```

```
lubo@kid:~/school/notes> ./a.out
```

```
p3=(3, 3)
```

- Изразът в ред 20 е еквивалентен на следното:

```
p3=operator+(p1, p2);
```

## Унарни оператори

- Унарните оператори могат да се дефинират по два начина:

- ◇ Като нестатична член-функция на класа, която не приема аргументи – например:

```
Point Point::operator-(void)
```

- ◇ Като функция, която не е член на класа и приема един аргумент – например:

```
Point operator-(const Point& p)
```

# Унарни оператори

- Нека разгледаме първият вариант за предефиниране на унарнен оператор. За пример ще използваме класът Point и унарният оператор -:

```
1 #include <iostream>
2 using namespace std;
3
4 class Point {
5     double x_, y_;
6 public:
7     Point(double x=0, double y=0)
8         : x_(x), y_(y)
9     {}
10    double get_x() const {return x_;}
11    double get_y() const {return y_;}
12    Point operator-(void) const;
13 };
```

16

```
14 Point Point::operator-() const {
15     Point result(-get_x(), -get_y());
16     return result;
17 }
18 int main(void) {
19     Point p1(1.0, 1.0), p2;
20
21     p2 = -p1;
22     cout << "p2=( "
23           << p2.get_x() << " , "
24           << p2.get_y() << " )" << endl;
25     return 0;
26 }
```

```
lubo@kid:~/school/notes> ./a.out
```

```
p2=(-1, -1)
```

- Изразът в ред 21 е еквивалентен на следното:

```
p2 = p1.operator-();
```

# Унарни оператори

- Нека разгледаме вторият вариант за предефиниране на унарен оператор. Като пример отново ще използваме класът Point и унарният оператор -:

```
1 #include <iostream>
2 using namespace std;
3
4 class Point {
5     double x_, y_;
6 public:
7     Point(double x=0, double y=0)
8         : x_(x), y_(y)
9     {}
10    double get_x() const {return x_;}
11    double get_y() const {return y_;}
12};
```

17

```
13 Point operator-(const Point& p) {
14     Point result(-p.get_x(), -p.get_y());
15     return result;
16 }
17 int main(void) {
18     Point p1(1.0, 1.0), p2;
19
20     p2 = -p1;
21     cout << "p2=( "
22         << p2.get_x() << " , "
23         << p2.get_y() << " )" << endl;
24     return 0;
25 }
```

```
lubo@kid:~/school/notes> ./a.out
```

```
p2=(-1, -1)
```

- Изразът в ред 20 е еквивалентен на следното:

```
p2 = operator-(p1);
```

# Предефиниране на оператори

- Всеки оператор може да се дефинира само за синтаксиса, който е определен за него в спецификацията на езика. Например:
  - ◇ Не може да се дефинира унарнен оператор за делене /, тъй като в спецификацията на езика този оператор е дефиниран като бинарен.
  - ◇ Не може да се дефинира бинарен оператор за логическо отрицание !, тъй като в спецификацията на езика този оператор е дефиниран като унарнен.
  - ◇ Операторът -, обаче, може да бъде предефиниран като унарнен и като бинарен оператор, тъй като в спецификацията на езика са дефинирани и двата варианта на оператора.

18

## Предефиниране на операторът за изход <<

- Операторът за изход << е бинарен оператор. Първият аргумент на оператора за изход задължително трябва да бъде от типа `ostream`.
- Типичният начин за предефиниране на операторът за изход е той да бъде дефиниран като функция извън рамките на класа по следният начин:

```
ostream& operator<<(ostream& out, const Point& p);
```

- Пример:

```
1 ostream& operator<<(ostream& out, const Point& p) {
2     out << "point(" << p.get_x() << ", "
3         << p.get_y() << ")";
4     return out;
5 }
```

19

# Пример: векторна аритметика

```
1 #include <iostream>
2 using namespace std;
3
4 class Point {
5     double x_, y_;
6 public:
7     Point(double x=0, double y=0)
8         : x_(x), y_(y)
9     {}
10    double get_x() const {return x_;}
11    double get_y() const {return y_;}
12    Point& operator+=(const Point& p);
13    Point& operator-=(const Point& p);
14    Point& operator*=(double alpha);
15 };
```

20

```
16 Point& Point::operator+=(const Point& p) {
17     x_+=p.get_x();
18     y_+=p.get_y();
19     return *this;
20 }
21 Point& Point::operator-=(const Point& p) {
22     x_-=p.get_x();
23     y_-=p.get_y();
24     return *this;
25 }
26 Point& Point::operator*=(double alpha) {
27     x_*=alpha;
28     y_*=alpha;
29     return *this;
30 }
```

```

31 Point operator+(const Point& p1, const Point& p2) {
32     Point result=p1;
33     result+=p2;
34     return result;
35 }
36 Point operator-(const Point& p1, const Point& p2) {
37     Point result=p1;
38     result-=p2;
39     return result;
40 }
41 Point operator*(const Point& p, double alpha) {
42     Point result=p;
43     result*=alpha;
44     return result;
45 }
46 Point operator*(double alpha, const Point& p) {
47     return p*alpha;
48 }

```

```

49 ostream& operator<<(ostream& out, const Point& p) {
50     out << "point(" << p.get_x() << ", " <<
51         << p.get_y() << ")";
52     return out;
53 }
54 int main(void) {
55     Point p1(1.0,1.0), p2(2.0,2.0), p3;
56     p3=p1+p2;
57     cout<< "p3=" << p3 << endl;
58     p3+=p1+p2;
59     cout<< "p3=" << p3 << endl;
60     p3=10.0*p1;
61     cout<< "p3=" << p3 << endl;
62     p3=p2*10.0;
63     cout<< "p3=" << p3 << endl;
64     return 0;
65 }

```

```

lubo@kid:~/school/notes> ./a.out
p3=point(3, 3)
p3=point(6, 6)
p3=point(10, 10)
p3=point(20, 20)

```

## Пример: масив с проверка на границите

```
1 #include <iostream>
2 #include <cstdlib>
3
4 using namespace std;
5
6 class Array {
7     int* data_;
8     unsigned int size_;
9 public:
10    Array(unsigned int size=10)
11        : size_(size), data_(new int[size])
12    {}
13    ~Array(void) {
14        delete [] data_;
15    }
```

21

```
16    int& element(unsigned int index) {
17        if(index<0 || index>=size_) {
18            cerr << "index out of bounds..." << endl;
19            exit(1);
20        }
21        return data_[index];
22    }
23    unsigned size() const {
24        return size_;
25    }
26 };
```



```

27 int main(void) {
28     Array v(3);
29
30     for(int i=0;i<3;++i) {
31         v.element(i)=i;
32     }
33     for(int i=0;i<3;i++) {
34         cout << "v[i]=" << v.element(i) << endl;
35     }
36
37     return 0;
38 }

```

```
lubo@kid:~/school/notes> ./a.out
```

```
v[i]=0
```

```
v[i]=1
```

```
v[i]=2
```

## Пример: масив с проверка на границите

```

1 #include <iostream>
2 #include <cstdlib>
3
4 using namespace std;
5
6 class Array {
7     int* data_;
8     unsigned int size_;
9 public:
10    Array(unsigned int size=10)
11        : size_(size), data_(new int[size])
12    {}
13    ~Array(void) {
14        delete [] data_;
15    }

```

```
16  int& operator[](unsigned int index) {
17      if(index<0 || index>=size_) {
18          cerr << "index out of bounds..." << endl;
19          exit(1);
20      }
21      return data_[index];
22  }
23  unsigned size() const {
24      return size_;
25  }
26  };
```

```
27  int main(void) {
28      Array v(3);
29
30      for(int i=0;i<3;++i) {
31          v[i]=i;
32      }
33      for(int i=0;i<3;i++) {
34          cout << "v[i]=" << v[i] << endl;
35      }
36
37      return 0;
38  }
```

```
lubo@kid:~/school/notes> ./a.out
```

```
v[i]=0
```

```
v[i]=1
```

```
v[i]=2
```

## Копиращ конструктор

- По подразбиране всички обекти могат да бъдат копирани. Всеки клас притежава копиращ конструктор, който е отговорен за копирането на обектите от съответният клас.
- Копиращият конструктор за класа `X` има сигнатура `X::X(const X&)`.
- Ако за даден клас не е дефиниран копиращ конструктор, то компилаторът генерира копиращ конструктор по подразбиране. Семантиката на този конструктор е да копира всички член-променливи на класа.

23

## Копиращ конструктор

- За класа `Point` поведението на подразбиращият се конструктор е еквивалентно на следното:

```
1 class Point {  
2     double x_, y_;  
3 public:  
4     Point(const Point& p)  
5         : x_(p.x_), y_(p.y_)  
6     {}  
7     //...  
8 };
```

- Ако подразбиращото се поведение на този конструктор е неподходящо за даден клас, то потребителят трябва да дефинира сам копиращ конструктор.

24

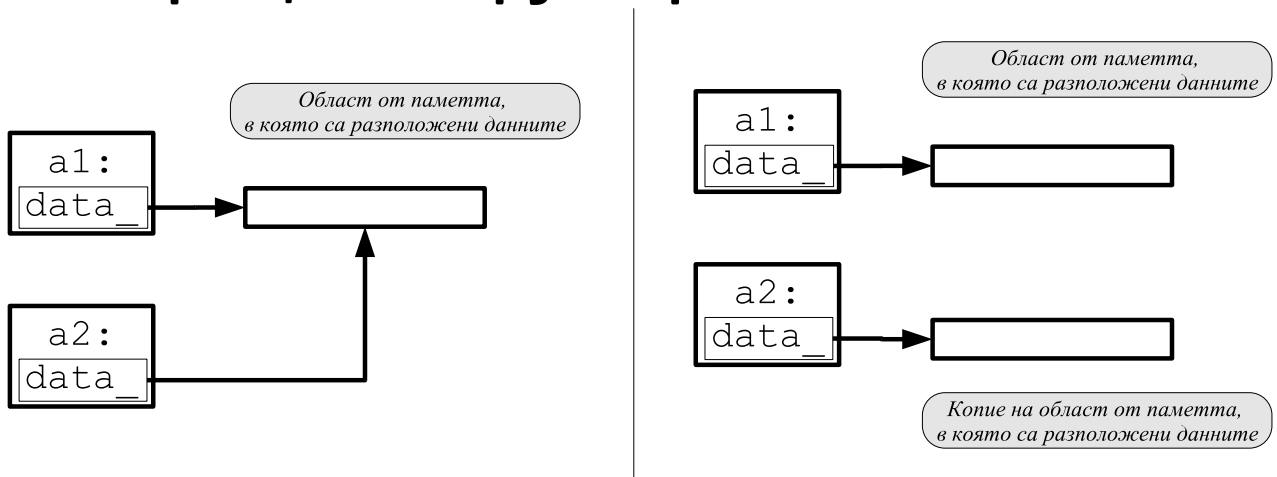
# Копиращ конструктор

- В повечето случаи подразбиращото се поведение на копиращият конструктор е напълно удовлетворително.
- Нека отново да разгледаме дефинираният от нас масив, с проверката на границите.

```
6 class Array {  
7     int* data_;  
8     unsigned int size_;  
9 public:  
10    Array(unsigned int size=10)  
11        : size_(size), data_(new int [size])  
12    {}  
13    ~Array(void) {  
14        delete [] data_;  
15    }
```

25

# Копиращ конструктор



- Подразбиращият се копиращ конструктор копира член-променливите на класа. Това означава, че ще се копират член променливите `data_` и `size_`. Областта от паметта, към която сочи `data_`, няма да бъде копирана.

26

- За да се обезпечи коректно поведение на масива при копиране е необходимо да се предефинира копиращият конструктор.

```
6 class Array {
7     int* data_;
8     unsigned int size_;
9 public:
10    Array(unsigned int size=10)
11        : size_(size), data_(new int[size_])
12    {}
13    Array(const Array& other)
14        : size_(other.size_), data_(new int[size_])
15    {
16        for(unsigned int i=0; i< size_; i++)
17            data_[i]=other.data_[i];
18    }
19    ~Array(void) {
20        delete [] data_;
21    }
```

27

```
22     int& operator[](unsigned int index) {
23         if(index<0 || index>=size_) {
24             cerr << "index_out_of_bounds..." << endl;
25             exit(1);
26         }
27         return data_[index];
28     }
29     unsigned size() const {
30         return size_;
31     }
32 };
```

```

33 int main(void) {
34     Array a1(3);
35     for(int i=0;i<3;++i) {
36         a1[i]=i;
37     }
38     Array a2=a1;
39     for(int i=0;i<3;i++) {
40         cout << "a2[i]=" << a2[i] << endl;
41     }
42     return 0;
43 }

```

```

lubo@kid:~/school/notes> ./a.out
a2[i]=0
a2[i]=1
a2[i]=2

```

## Копиращ конструктор

- Обърнете внимание, че като аргумент на копиращият конструктор се използва препратка — `X::X(const X& x)`.
- Ако в дефиницията на копиращият конструктор не се използва препратка — `X::X(X x)`, — то това ще доведе до безкрайна рекурсия. Проблемът е, че при предаване на аргумента по стойност, се извършва копиране, което води до извикване на копиращ конструктор.
- Ако искаме да забраним копирането на обектите на даден клас е необходимо да дефинира **private** копиращ конструктор.

# Оператор за присвояване

- По подразбиране за всички обекти може да се използва оператор за присвояване. Всеки клас притежава оператор за присвояване, който е отговорен за присвояване на обекти от съответния клас.
- Операторът за присвояване на класа  $X$  има сигнатура  $X\& X::operator=(const X\&)$ .
- Ако за даден клас не е дефиниран оператор за присвояване, то компилаторът генерира оператор за присвояване по подразбиране. Семантиката на този оператор е да копира всички член-променливи на класа.

29

# Оператор за присвояване

- За класа `Point` поведението на подразбиращият се оператор за присвояване е еквивалентно на следното:

```
1 class Point {  
2     double x_, y_;  
3 public:  
4     //...  
5     Point& operator=(const Point& other){  
6         x_=other.x_;  
7         y_=other.y_;  
8         return *this;  
9     }  
10 };
```

- Ако подразбиращото се поведение на този оператор е неподходящо за даден клас, то потребителят трябва да дефинира сам оператор за присвояване.

30

# Оператор за присвояване: пример

- В повечето случаи подразбиращото се поведение на оператора за присвояване е напълно удовлетворително.
- За да се обезпечи коректно поведение на масива при присвояване е необходимо да се предефинира оператора за присвояване.

```
25   Array& operator=(const Array& other) {  
26       if(this!=&other) {  
27           delete [] data_;  
28           size_=other.size_;  
29           data_=new int[size_];  
30           for(unsigned i=0;i<size_;i++)  
31               data_[i]=other.data_[i];  
32       }
```