

Конструктори

Жизнен цикъл на обект

- За обекта се заделя памет и се свързва с неговото име
- Извиква се подходящ конструктор на обекта
- ... (достъп до компоненти на обект, изпълняване на операции)
- Достига се края на областта на действие на обекта
- Извиква се деструкторът на обекта
- Заделената за обекта памет се освобождава

За какво служат конструкторите?

- Инициализират паметта за обекта
- Осигуряват, че преди да почне да се работи с обекта, той е във валидно състояние
- Позволяват предварително задаване на стойности на полетата

Видове конструктори

- Обикновен конструктор
- Конструктор по подразбиране
- Конструктор с параметри по подразбиране
- Конструктор за копиране
- Системно генерирани конструктори
 - по подразбиране
 - за копиране
- Конструктор за преобразуване на тип

Дефиниция на конструктор

- `<конструктор> ::=`
 `<име_на_клас>::<име_на_клас>(<параметри>)`
 `[: <член-данна>(<израз>) {, <член-данна>(<израз>) }]`
 `{ <тяло> }`
- Пример:
 `Rational :: Rational (int n, int d) : numer(n), denom(d) {`
 `if (denom == 0)`
 `cerr << „Нулев знаменател!“;`
 `}`
- Инициализиращият списък се изпълнява преди тялото на конструктора!

Използване на конструктори

- $\langle \text{описание_на_обект} \rangle ::=$
 $\langle \text{име_на_обект} \rangle [= \langle \text{израз} \rangle] |$
 $\langle \text{име_на_обект} \rangle (\langle \text{параметри} \rangle) |$
 $\langle \text{име_на_обект} \rangle = \langle \text{име_на_клас} \rangle (\langle \text{параметри} \rangle)$

- Примери:

```
Rational r1, r2 = Rational(), r3(1, 2), r4 = Rational(3,4);  
Rational r5 = r1, r6(r2), r7 = Rational(r3)
```

Конструктор по подразбиране

- Конструктор без параметри
`<име_на_клас>()`
- Извиква се при дефиниция на обект без параметри
`Rational r1;`
~~`Rational r2();`~~
`Rational r3 = Rational();`
- Инициализира обекта с „празни“, но валидни стойности
- Пример:
`Rational::Rational() : numer(0), denom(1) {}`
- Ако в един клас не дефинирате нито един конструктор, системно се създава конструктор по подразбиране с празно тяло

Подразбиращи се параметри

- В C++ е позволено да се задават стойности по подразбиране на някои или всички параметри на функции
- `<функция_с_подразбиращи_се_параметри> ::= <тип> <име> (<параметри> <подразбиращи_се_параметри>)`
- `<параметри> ::= void | <празно> | <параметър> {, <параметър> }`
- `<подразбиращи_се_параметри> ::= <празно> | <параметър> = <израз> {, <параметър> = <израз> }`
- Пример:
`int f(int x, double y, int z = 1, char t = 'x')`
`void g(int *p = NULL, double x = 2.3)`
~~`int h(int a = 0, double b)`~~

Конструктор с подразбиращи се параметри

- Конструкторите могат да бъдат с подразбиращи се параметри като всички останали функции
- Пример:
`Rational(int n = 0, int d = 1)`
- Дефинираме три конструктора наведнъж!
 - `Rational()` ↔ `Rational(0,1)` (конструктор по подразбиране)
 - `Rational(n)` ↔ `Rational(n,1)`
 - `Rational(n, d)`
- Подразбиращите параметри се задават в декларацията на конструктора, ако има такава

Конструктор за копиране

- Конструкторът за копиране служи за инициализиране на обект като се ползва като образец друг обект
- `<име_на_клас>(<име_на_клас> const&)`
- Образецът не трябва да може да се променя!
- Пример:
`Rational(Rational const& r) : numer(r.numer), denom(r.denom) {}`
- Ако не напишете конструктор за копиране се създава системен такъв, който копира дословно полетата на образца
- Конструкторът за копиране обикновено се пише, ако при копирането на обекта е нужно да се случи нещо допълнително

Извикване на конструктор за копиране

- `<име_на_клас> <обект>(<образец>)`
- `<име_на_клас> <обект> = <образец>`
- `<име_на_клас> <обект> = <име_на_клас>(<образец>)`
- Конструктор за копиране се извиква автоматично и при:
 - предаване на обекти като параметри на функции
 - връщане на обекти като резултат от функции
- Примери

Копиране на обекти със статични полета

```
Student s1(40000, „Надя Петрова“, 4.5);  
Student s2 = s1;  
s2.setName(„Надя Иванова“);
```

s1

40000	Надя Петрова	4.5
-------	--------------	-----

s2

40000	Надя Иванова	4.5
-------	--------------	-----

```
void anonymousPrint(Student s) {  
    s.setName(„Анонимен“);  
    cout << „Информация за оценка:“;  
    s.print();  
}
```

s

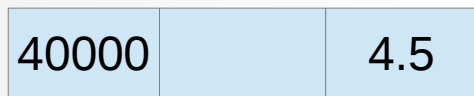
40000	Анонимен	4.5
-------	----------	-----

Копиране на обекти с динамични полета

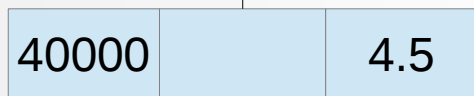
```
Student s1(40000, „Надя Петрова“, 4.5);  
Student s2 = s1;  
s2.setName(„Надя Иванова“);
```

```
void anonymousPrint(Student s) {  
    s.setName(„Анонимен“);  
    cout << „Информация за оценка:“;  
    s.print();  
}
```

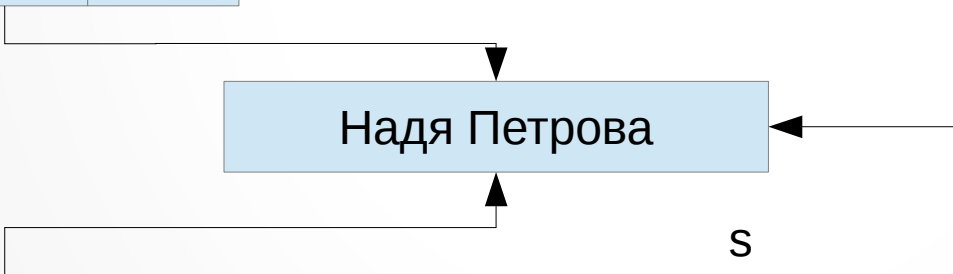
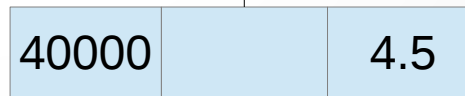
s1



s2



s



Копиране на обекти с динамични полета

```
Student s1(40000, „Надя Петрова“, 4.5);  
Student s2 = s1;  
s2.setName(„Надя Иванова“);
```

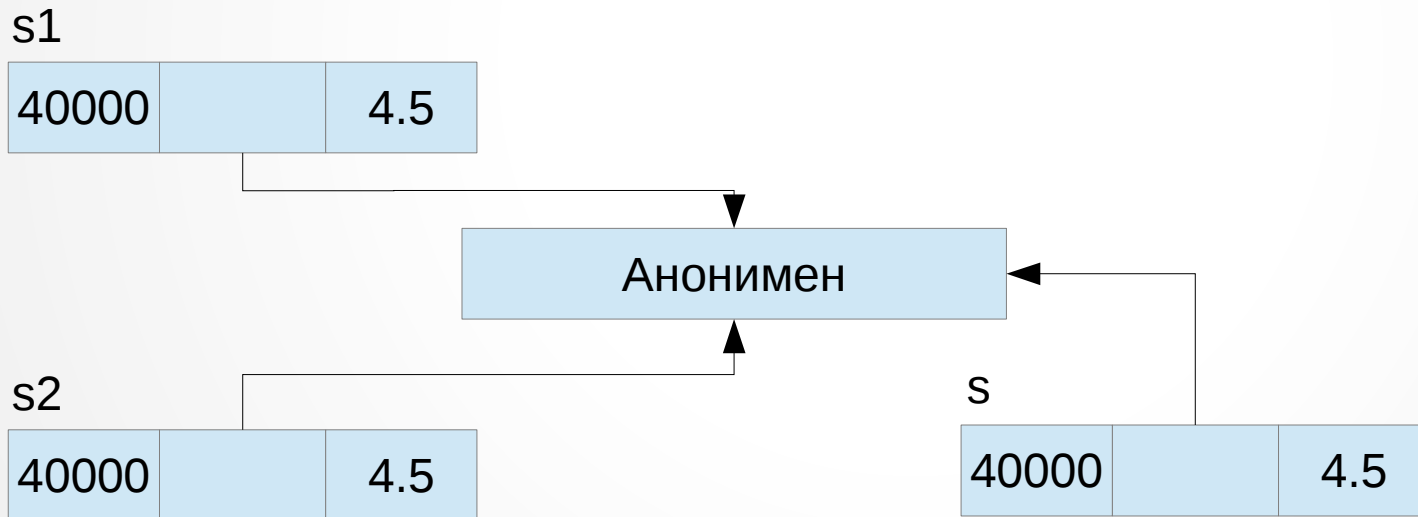
```
void anonymousPrint(Student s) {  
    s.setName(„Анонимен“);  
    cout << „Информация за оценка:“;  
    s.print();  
}
```



Копиране на обекти с динамични полета

```
Student s1(40000, „Надя Петрова“, 4.5);  
Student s2 = s1;  
s2.setName(„Надя Иванова“);
```

```
void anonymousPrint(Student s) {  
    s.setName(„Анонимен“);  
    cout << „Информация за оценка:“;  
    s.print();  
}
```



Конструктор за копиране за динамични полета

- Системният конструктор сякаш копира полетата
- При работа с динамична памет трябва да напишем собствен конструктор за копиране
- Трябва да се погрижим да заделим нова динамична памет и да копираме съдържанието на оригинала

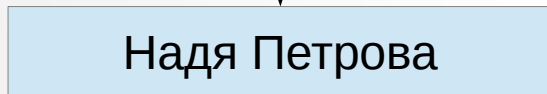
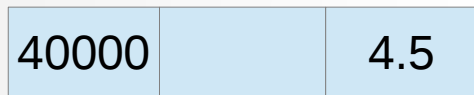
```
Student(Student const& s) : fn(s.fn), grade(s.grade) {  
    name = new char[strlen(s.name)+1];  
    strcpy(name, s.name);  
}
```


Копиране на обекти с динамични полета

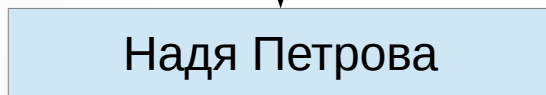
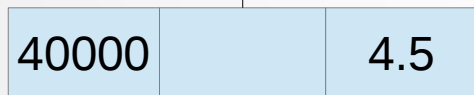
```
Student s1(40000, „Надя Петрова“, 4.5);  
Student s2 = s1;  
s2.setName(„Надя Иванова“);
```

```
void anonymousPrint(Student s) {  
    s.setName(„Анонимен“);  
    cout << „Информация за оценка:“;  
    s.print();  
}
```

s1



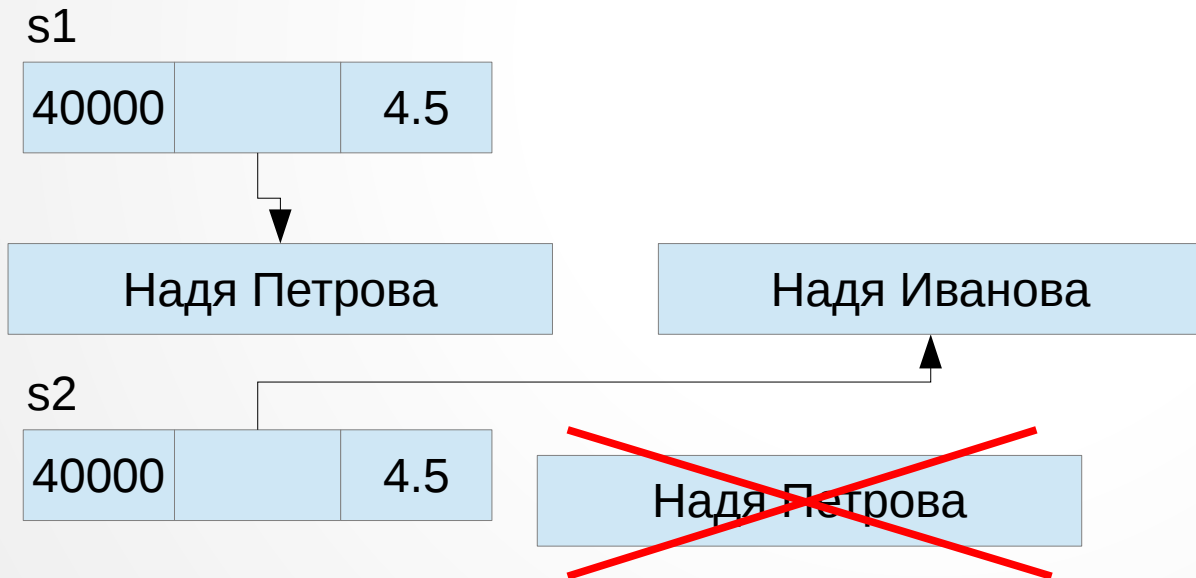
s2



Копиране на обекти с динамични полета

```
Student s1(40000, „Надя Петрова“, 4.5);  
Student s2 = s1;  
s2.setName(„Надя Иванова“);
```

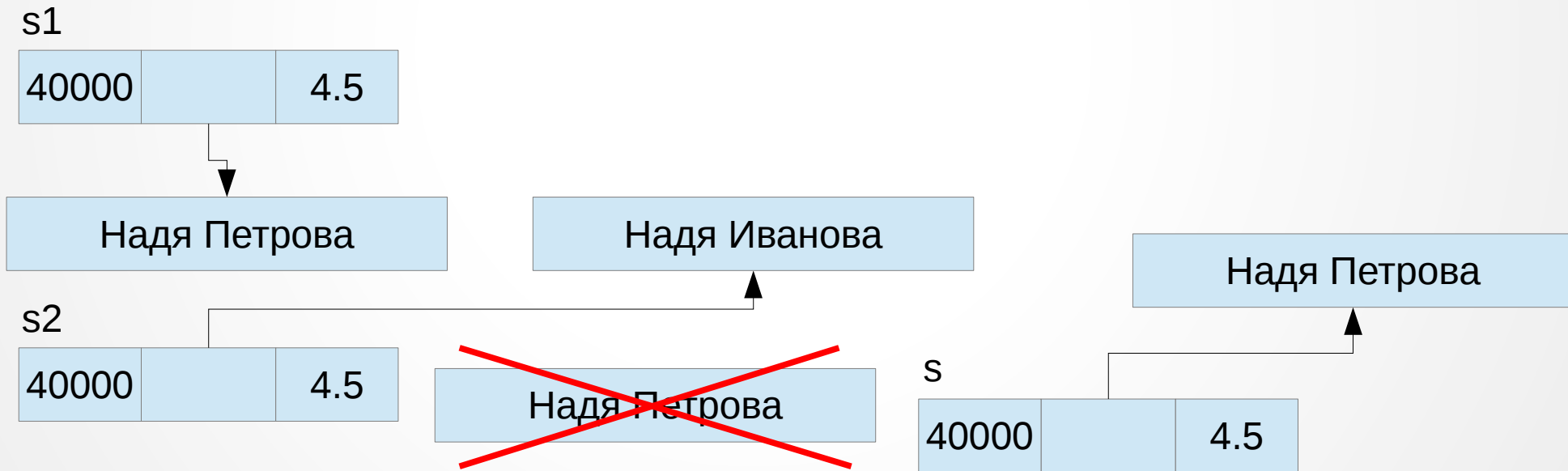
```
void anonymousPrint(Student s) {  
    s.setName(„Анонимен“);  
    cout << „Информация за оценка:“;  
    s.print();  
}
```



Копиране на обекти с динамични полета

```
Student s1(40000, „Надя Петрова“, 4.5);  
Student s2 = s1;  
s2.setName(„Надя Иванова“);
```

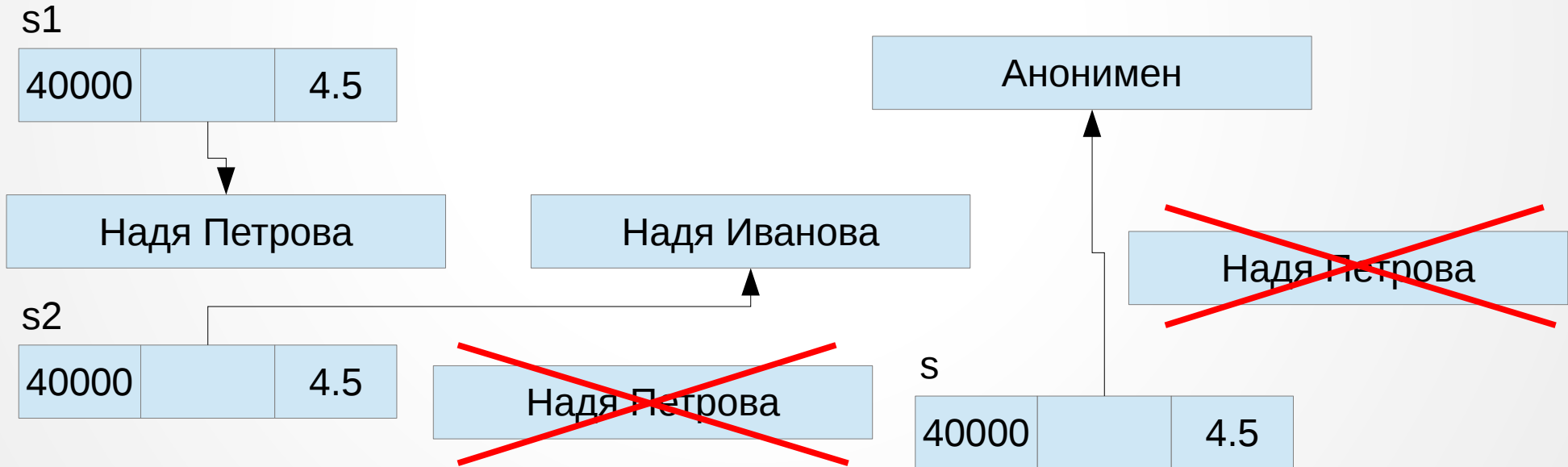
```
void anonymousPrint(Student s) {  
    s.setName(„Анонимен“);  
    cout << „Информация за оценка:“;  
    s.print();  
}
```



Копиране на обекти с динамични полета

```
Student s1(40000, „Надя Петрова“, 4.5);  
Student s2 = s1;  
s2.setName(„Надя Иванова“);
```

```
void anonymousPrint(Student s) {  
    s.setName(„Анонимен“);  
    cout << „Информация за оценка:“;  
    s.print();  
}
```



Конструктор за преобразуване на тип

- Конструкторите с точно един параметър са специални
- `<име_на_клас>(<тип_за_преобразуване>)`
- Задават правило за конструиране на обект от класа по обект от друг клас, или от стойност от вграден тип
- Навсякъде, където се очаква `Rational`, но се подава стойност от друг тип, C++ се опитва да използва конструктор за преобразуване на тип

- Примери

```
Rational r = 5; // ↔ Rational r(5); r = 5/1
```

```
add(3, Rational(2, 3)).print(); // ↔ add(Rational(3), Rational(2, 3)).print();
```

```
Rational round(Rational r) {
```

```
    int wholePart = r.getNumerator() / r.getDenominator();
```

```
    return wholePart; // ↔ return Rational(wholePart);
```

```
}
```

Временни обекти

- Конструкторите могат да се използват за създаване на временни анонимни обекти
- `<временен_обект> ::= <име_на_клас>(<параметри>)`
- Пример:

```
Rational(2, 3).print();  
cout << add(Rational(1,2), Rational(1,4));
```
- Тези обекти се създават само за да бъдат веднага използвани
- Временните обекти се унищожават веднага след като бъдат използвани

Обектите като член-данни

- Член-данните на даден клас биха могли да бъдат обекти от друг клас
- Всяка член-данна, която е обект се инициализира автоматично с конструктор по подразбиране
class RationalPoint {
 Rational x, y;
 RationalPoint() {} // x = 0/1, y = 0/1
- Ако искаме да инициализираме с друг конструктор, трябва да зададем параметрите му в инициализацията списък
 RationalPoint(Rational p) : x(p), y(3, 5) }

Обектите като член-данни

- Системният конструктор за копиране автоматично извиква конструкторите за копиране на всички обекти член-данни
RationalPoint p(Rational(2,3)); // p = (2/3, 3/5)
RationalPoint q = p; // q = (2/3, 3/5)
- **Внимание!** Ако пишем собствен конструктор за копиране, трябва ръчно да извикаме конструкторите за копиране на всички член-данни, които са обекти!

```
RationalPoint(RationalPoint const& p) : x(p.x), y(p.y) {}
```


Масиви и обекти

- Можем да дефинираме масиви от обекти от един и същи клас
<клас> <име>[<брой>
[= { <описание_на_обект> {, <описание_на_обект> } }] ;
- Дефинира масив <име> от <брой> обекта от <клас>, всеки от които се инициализира със съответен конструктор
- Примери:
Rational p(1,3), q(3, 5);
Rational a[6] = { Rational(), Rational(5, 7), p, Rational(q), 1 };

Достъп до елементите на масив от обекти

- Достъпът става по същия начин като с обикновени масиви
- Примери:
- `a[2].print();`
- `cout << a[3].getDenominator();`
- `Rational r = a[1];`
- `Rational* p = a + 1; (++p)->print();`
- `(a + 4)->read();`

Динамични обекти

- Можем да създаваме обекти в динамичната памет
- **new** <клас>
връща указател към нов обект, инициализиран с конструктор по подразбиране
- **new** <клас>(<параметри>)
връща указател към нов обект, инициализиран със съответния конструктор (в зависимост от параметрите)
- **new** <клас>[<брой>]
връща указател към масив от обекти, инициализирани с конструктор по подразбиране