

# Обектно ориентирано програмиране

---

КЛАСОВЕ. КОНСТРУКТОРИ. УКАЗАТЕЛИ КЪМ  
ОБЕКТИ

# Конструктори

---

Създаването на обекти е свързано с отделяне на памет, запомняне на текущо състояние, задаване на начални стойности и др. дейности, които се наричат **инициализация на обекта**.

В езика C++ тези дейности се изпълняват от специален вид член-функции на класовете – конструкторите.

# Конструктори ...

---

## Дефиниране на конструктор (най-често използвана форма)

<дефиниция\_на\_конструктор> ::=

<име\_на\_клас>::<име\_на\_клас>(<параметри>)

{<тяло>}

<тяло> ::= <редица\_от\_оператори\_и\_дефиниции>

<параметри> се определя както формални параметри на функция.

# Конструктори ...

---

Ще напомним, че конструкторът е член-функция, която притежава повечето характеристики на другите член-функции, но има и редица особености, като:

- Името на конструктора съвпада с името на класа.
- Типът на резултата е указател към новосъздадения обект (типът на указателя `this`) и явно не се указва.
- Изпълнява се автоматично при създаване на обекти.
- Не може да се извиква явно.

Освен това в един клас може явно да не е дефиниран конструктор, но може да са дефинирани и няколко конструктора.

# Конструктори ...

---

Типична дефиниция :

```
class Example
{
public:
    Example(int, int, int);
    void print() const;
private:
    int a, b, c;
};

Example::Example(int x, int y, int z) { // конструктор с три параметъра
    a = x;
    b = y;
    c = z;
}
```

# Конструктори ...

---

По-обща дефиниция на конструктор.

## Дефиниране на конструктор

<дефиниция\_на\_конструктор> ::=

<име\_на\_клас>::<име\_на\_клас>(<параметри>):

    <член\_данна>(<израз>){,<член\_данна>(<израз>)}

{<тяло>}

<тяло> ::= <редица\_от\_оператори\_и\_дефиниции>

Забелязваме, че е възможно член\_данна да се свърже с инициализираща стойност в заглавието на конструктора.

# Конструктори ...

---

**Пример:** Конструкторът на класа Example може да се дефинира и по следния начин

```
Example::Example(int x, int y, int z): a(x), b(y), c(z)
{ }
```

Ще отбележим, че не е задължително всички член-данни да са инициализирани само пред тялото или само вътре в тялото на конструктора.

**Пример:**

```
Example::Example(int x, int y, int z): a(x) {
    b = y;
    c = z;
}
```

# Конструктори ...

---

Смисълът на обобщената синтактична конструкция е, че инициализацията на член-данните в заглавието предшества изпълнението на тялото на конструктора. Това я прави изключително полезна.

Използването ѝ увеличава ефективността на класа поради следните съображения.

- Когато член-данни на клас са обекти, в дефиницията на конструктора на класа при инициализацията се използват конструкторите на класовете, от които са обектите (член-данни).
- Преди да започне изпълнението на указаните конструктори, автоматично се извикват конструкторите по подразбиране на всички член-данни, които са обекти.
- Веднага след това тези член-данни се инициализират, резултат от изпълнението на извиканите конструктори. Тази двойна инициализация намалява ефективността на програмата.



# Конструктори ...

---

```
class Vector
{
private:
    Point start;
    Point end;
public:
    // конструктор
    Vector(Point, Point);
    ...
};
```

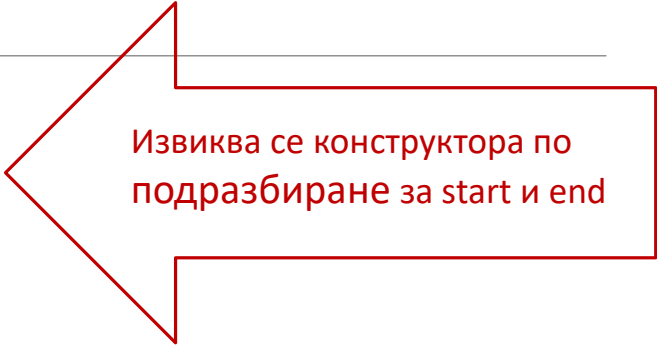
# Конструктори ...

---

```
Vector::Vector(Point a, Point b)
{
    start = a;
    end = b;
}
```



Копиране на обектите



Извиква се конструктора по подразбиране за start и end

# Конструктори ...

---

По-добре е да се дефинира така:

```
Vector::Vector(Point a, Point b): start(a), end(b)
{ }
```



# Предефинирани функции

---

## Предефинирани функции

Обект (в общия смисъл) е предефиниран, ако за него има няколко различни дефиниции, задаващи различни негови интерпретации. За да бъдат използвани такива конструкции е необходим критерии, по който те да се различат.

В рамките на една програма може да се извършва предефиниране на функции. Възможно е:

*а) да се използват функции с едно и също име с различни области на видимост*

В този случай не възниква проблем с различаването.

*б) да се използват функции с едно и също име в една и съща област на видимост*

# Предефинирани функции

---

В този случай компилаторът търси функцията с възможно най-добро съвпадане. Като критерии за добро съвпадане са въведени следните нива на съответствие:

- точно съответствие (по брой и тип на формалните и фактическите параметри)
- съответствие чрез разширяване на типа.

Извършва се разширяване по веригата

char -> short -> int -> long int или

float -> double

- други съответствия (правила въведени от потребителя).

# Предефинирани конструктори

---

В един клас може да са дефинирани няколко конструктора. Всички те имат едно име (името на класа), но трябва да се различават по броя и/или типа на параметрите си. Наричат се **предефинирани конструктори**. При създаването на обект на класа се изпълнява само един от тях. Определя се съгласно критерия за най-добро съвпадане.

**Пример:** В класа Rational дефинирахме два конструктора Rational() и Rational(int, int), които се различават по броя на параметрите си.

# Подразбиращ се конструктор

---

В клас може явно да е дефиниран, но може и да не е дефиниран конструктор. Ако явно не е дефиниран конструктор, автоматично се създава един т. нар. **подразбиращ се конструктор**. Този конструктор реализира множество от действия като: заделяне на памет за данните на обект, инициализиране на някои системни променливи и др.

Подразбиращият се конструктор може да бъде предефиниран от потребителя. За целта е необходимо в класа да бъде дефиниран конструктор без параметри.

**Пример:**

```
Rational::Rational() : numer(0), denom(1) {}
```

# Подразбиращи се параметри

---

Функциите в езика C++ могат да имат подразбиращи се параметри. За тези параметри се задават подразбиращи се стойности, които се използват само ако при извикването на функцията не бъде зададена стойност за съответния параметър.

Задаването на подразбираща се стойност се извършва чрез задаване на конкретна стойност в прототипа на функцията или в нейната дефиниция.



# Подразбиращи се параметри

---

<функция\_с\_подразбиращи\_се\_параметри> ::=  
<тип> <име> ( <параметри> <подразбиращи\_се\_параметри> )

<параметри> ::= **void** | <празно> | <параметър> {, <параметър> }

<подразбиращи\_се\_параметри> ::= <празно> | <параметър> =  
<израз> {, <параметър> = <израз> }

Примери:

```
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)
```

# Подразбиращи се параметри

---

```
#include <iostream>

using namespace std;

void f(double, int = 10, char* = "example1"); //интервал между * и =

int main() {
    double x = 1.5;
    int y = 5;
    char z[] = "example 2";
    f(x, y, z);
    f(x, y);
    f(x);
    return 0;
}

void f(double x, int y, char* z) {
    cout << "x= " << x << " y= " << y << " z= " << z << endl;
}
```

# Подразбиращи се параметри

---

В тази програма е дефинирана функцията  $f$  с три формални параметъра. От прототипа ѝ се вижда, че два от тях (вторият и третият) са подразбиращи се със стойности по подразбиране 10 и “example 1” съответно. Тъй като в обръщението към  $f$

$f(x, y);$  и  $f(x);$

са указани по-малко от три фактически параметъра, за стойности на липсващите параметри се вземат указаните стойности от прототипа на функцията.

В резултат от изпълнението на програмата се получава:

$x = 1.5 \ y = 5 \ \text{example 2}$

$x = 1.5 \ y = 5 \ \text{example 1}$

$x = 1.5 \ y = 10 \ \text{example 1}$

# Подразбиращи се параметри

---

Ще отбележим също, че ако за даден подразбиращ се параметър е зададена стойност при обръщението към функцията, за всички параметри *пред него* също трябва да са указани такива.

Всичко казано за подразбиращите се параметри на функции се отнася и за конструкторите.

# Конструктор с подразбиращи се параметри

---

Ако променим дефиницията на класа Rational по следния начин:

```
class Rational {  
private:  
    int numer, denom;  
    int gcd(int, int);  
public:  
    // конструктор с подразбиращи се параметри  
    Rational(int n = 0, int d = 1);  
    int getNumerator() const;  
    int getDenominator() const;  
    void print() const;  
    void read();  
};
```

# Конструктор с подразбиращи се параметри

---

Дефинираме три конструктора наведнъж!

- $\text{Rational}() \leftrightarrow \text{Rational}(0,1)$  (конструктор по подразбиране)
- $\text{Rational}(n) \leftrightarrow \text{Rational}(n,1)$
- $\text{Rational}(n, d)$

Подразбиращите параметри се задават в декларацията на конструктора.

Не може да имаме конструктор без параметри (подразбиращ се конструктор) и конструктор, на който всички параметри са подразбиращи се.

# Конструктор с подразбиращи се параметри

---

Допустими са следните дефиниции на обекти:

```
Rational p,           // p се инициализира с 0/1  
q = Rational(),       // q се инициализира с 0/1  
r = Rational(5),      // r се инициализира с 5/1  
s = Rational(13, 21), // s се инициализира с 13/21  
t(2);                // t се инициализира с 2/1
```

# Конструктор за копиране

---

Инициализацията на новосъздаден обект на даден клас може да зависи от друг обект на същия клас. За да се укаже такава зависимост се използва знакът за присвояване или кръгли скоби.

**Пример:** Нека

```
Rational p = Rational(1, 4);
```

Чрез еквивалентните конструкции

```
Rational q = p;
```

```
Rational q(p);
```

се създава обектът *q* от клас *rat*, като инициализацията на *q* зависи от *p*. Тази инициализация се създава от специален конструктор, наречен **конструктор за копиране**.



# Конструктор за копиране ...

---

Конструкторът за копиране конструктор, поддържащ формален параметър от тип <име\_на\_клас> **const &**.

Ако в един клас явно не е дефиниран конструктор за копиране, компилаторът автоматично създава такъв, в момента когато новосъздаден обект се инициализира с обекта, намиращ се от дясната страна на знака за присвояване или в кръглите скоби. Този конструктор се нарича конструктор за копиране.

**Пример:** В класа Rational не беше дефиниран конструктор за копиране. Затова при създаване на обект q чрез дефиницията

**Rational** q = p;

автоматично се извиква създадения от транслятора конструкторът за копиране.

# Конструктор за копиране ...

---

Последният има вида:

```
Rational::Rational(Rational const & r)
{
    numer = r.numer;
    denom = r.denom;
}
```

Дефиницията `Rational q = p;` създава нов обект `q` (*без викане на конструктор*), в който се копират съответните стойности на обекта `p`.

Ако в класа е дефиниран конструктор за копиране, компилаторът го използва.

# Конструктор за копиране ...

---

Пример:

```
class Rational {  
private:  
    int numer, denom;  
    int gcd(int, int);  
public:  
    Rational(int = 0, int = 1);  
    Rational(Rational const &); // Конструктор за копиране  
    int getNumerator() const;  
    int getDenominator() const;  
    void print() const;  
    void read();  
};
```

# Конструктор за копиране ...

---

...

```
Rational::Rational(Rational const & r)
```

```
{
```

```
    numer = r.numer + 1;
```

```
    denom = r.denom + 1;
```

```
}
```

...

```
Rational p,    // p се инициализира с 0/1
```

```
    q = p,      // q се инициализира с 1/2
```

```
    r = q,      // r се инициализира с 2/3
```

```
    s = r,      // s се инициализира с 3/4
```

```
    t(s);       // t се инициализира с 4/5.
```

# Конструктор за копиране ...

---

*Предефиниране на служебния конструктор за копиране се налага когато обектите имат член-данни, указващи динамична памет.*

Конструкторът за копиране се използва и:

- при предаване на обект като аргумент на функция, когато предаването е по стойност,
- при връщане на обект като резултат от изпълнение на функция.

Обектите могат да се предават като параметри на функциите по един от известните вече три начина: по стойност, по указател и по псевдоним. При предаване по стойност функциите работят с *копия* на параметрите, а не със самите параметри. При другите два начина за предаване на параметрите не се правят копия (функциите работят с оригиналните параметри).

# Конструктор за копиране ...

---

**Пример:** Нека останем в означенията на класа Rational с „глупавия“ конструктор за копиране от предишния пример и нека функцията sum, намираща сума на две рационални числа, е дефинирана по следния начин:

a)

```
Rational add(Rational p, Rational q) {  
    return Rational(p.getNumerator() * q.getDenominator()  
        + p.getDenominator() * q.getNumerator(),  
        p.getDenominator() * q.getDenominator());  
}
```

# Конструктор за копиране ...

---

Обръщението `add(p, q).print()` извежда **8/7**. Този резултат може да се обясни по следния начин. Тъй като `p` и `q` са параметри стойности, те се свързват с фактическите си параметри чрез копиране. В резултат `p` се свързва с  $1/2$  (не с  $0/1$ ), а `q` – с  $2/3$  (не с  $1/2$ ).

След изпълнението на инициализацията

```
Rational(p.getNumerator() * q.getDenominator()  
        + p.getDenominator() * q.getNumerator(),  
        p.getDenominator() * q.getDenominator())
```

чрез двуаргументния конструктор `Rational(int, int)` се създава обект, който се свързва със  $7/6$ .

Тъй като функцията `add` е от тип `Rational`, при изпълнение на `return` се извиква отново конструктора за копиране и се получава  $8/7$ .

# Конструктор за копиране ...

---

6)

```
Rational add(Rational const & p, Rational const & q) {  
    return Rational(p.getNumerator() * q.getDenominator()  
        + p.getDenominator() * q.getNumerator(),  
        p.getDenominator() * q.getDenominator());  
}
```



# Конструктор за копиране ...

---

Сега обръщението `add(p, q).print()` извежда **2/3**. Този резултат може да се обясни по следния начин. Тъй като `p` и `q` са параметри псевдоними, те директно се свързват с фактическите си параметри (не се извършва присвояване), т.е. `p` се свързва с `0/1`, а `q` – `1/2`. След изпълнението на инициализацията

```
Rational(p.getNumerator() * q.getDenominator()  
        + p.getDenominator() * q.getNumerator(),  
        p.getDenominator() * q.getDenominator())
```

се създава обект, който се свързва със `1/2`. Аналогично на случай а), при изпълнение на `return` се извиква конструктора за копиране и се получава `2/3`.

# Конструктор за копиране ...

---

B)

```
Rational add(Rational *p, Rational *q) {  
    return Rational(p->getNumerator() * q->getDenominator()  
        + p->getDenominator() * q->getNumerator(),  
        p->getDenominator() * q->getDenominator());  
}  
  
...  
  
Rational *p1 = &p, *q1 = &q;  
add(p1, q1).print();
```

# Конструктор за копиране ...

---

Обръщението `add(p1, q1).print()` извежда  $2/3$ .

Този резултат може да се обясни по следния начин. Тъй като `p` и `q` са параметри указатели, те се свързват с фактическите си параметри чрез адрес. В резултат `p` се свързва с  $0/1$ , а `q` – с  $1/2$ . След изпълнението на инициализацията се създава обект, който се свързва със  $1/2$ . Тъй като функцията `add` е от тип `Rational`, при изпълнение на `return` се извиква конструктора за копиране и се получава  $2/3$ .

# Указатели към обекти на класове

---

Дефинират се по същия начин както се дефинират указатели към променливи от стандартните типове данни.

**Пример:**

```
Rational p;
```

```
Rational *ptr = &p;
```

В резултат ще се отделят 4В ОП, които ще се именуват с ptr и ще се инициализират с адреса на обекта p.

# Указатели към обекти на класове

---

Достъпът до компонентите на рационалното число, сочено от `ptr`, се осъществява по общоприетия начин:

```
(*ptr).getNumerator()
```

```
(*ptr).getDenominator()
```

Синтактичната конструкция `(*ptr).` е еквивалентна на `ptr->`. Така горните обръщения могат да се запишат и по следния начин:

```
ptr->getNumerator();
```

```
ptr->getDenominator();
```

Ще напомним, че `this` е указател от тип `<име_на_клас>*`.

# Указатели към обекти на класове

---

Функция или член-функция може да връща указател към обект.

**Внимание:** Функцията не трябва да връща адрес на дефиниран във функцията обект. Защо?

// Събиране на 2 рационални числа

```
Rational* add(Rational p, Rational q) {  
    Rational r = Rational(p.getNumerator() * q.getDenominator()  
        + p.getDenominator() * q.getNumerator(),  
        p.getDenominator() * q.getDenominator());  
    return &r;  
}
```