

Конструктори

Трифон Трифонов

Обектно-ориентирано програмиране,
спец. Компютърни науки, 1 поток,
спец. Софтуерно инженерство,
2016/17 г.

16–23 март 2017 г.

Жизнен цикъл на обект

- За обекта се заделя памет и се свързва с неговото име

Жизнен цикъл на обект

- За обекта се заделя памет и се свързва с неговото име
- Извиква се подходящ конструктор на обекта

Жизнен цикъл на обект

- За обекта се заделя памет и се свързва с неговото име
- Извиква се подходящ конструктор на обекта
- Работа с обекта (достъп до компоненти на обект, изпълняване на операции)

Жизнен цикъл на обект

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

Жизнен цикъл на обект

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

Жизнен цикъл на обект

- За обекта се заделя памет и се свързва с неговото име
- Извиква се подходящ конструктор на обекта
- Работа с 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

Ролята на 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 с параметри
- 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> }
```

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!";
}

```

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!";
}

```

Инициализацията списък се изпълнява преди тялото на конструктора!

Извикване на конструктори

```
<описание на обект> ::=  
    <име-на-обект> [ = <израз> ] |  
    <име-на-обект> (<параметри>) |  
    <име-на-обект> = <име-на-клас> (<параметри>)
```

Извикване на конструктори

```
<описание на обект> ::=  
    <име-на-обект> [ = <израз> ] |  
    <име-на-обект> (<параметри>) |  
    <име-на-обект> = <име-на-клас> (<параметри>)
```

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> ()

Конструктор по подразбиране

- Конструктор без параметри: <име-на-клас> ()
- Извиква се при дефиниция на обект без параметри

Конструктор по подразбиране

- Конструктор без параметри: `<име-на-клас>()`
- Извиква се при дефиниция на обект без параметри
 - `Rational r1;`

Конструктор по подразбиране

- Конструктор без параметри: `<име-на-клас>()`
- Извиква се при дефиниция на обект без параметри
 - `Rational r1;`
 - ~~`Rational r2();`~~

Конструктор по подразбиране

- Конструктор без параметри: `<име-на-клас>()`
- Извиква се при дефиниция на обект без параметри
 - `Rational r1;`
 - ~~`Rational r2();`~~
 - `Rational r3 = Rational();`

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

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) {}

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 се дефинира **ниито 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

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>)`

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

- В C++ е позволено да се задават стойности по подразбиране на някои или всички параметри на функции
- $\langle \text{функция-с-подразбиращи-се-параметри} \rangle ::= \langle \text{тип} \rangle \langle \text{име} \rangle (\langle \text{параметри} \rangle \langle \text{подразбиращи-се-параметри} \rangle)$
- $\langle \text{параметри} \rangle ::= \text{void} \mid \langle \text{празно} \rangle \mid \langle \text{параметър} \rangle \{ , \langle \text{параметър} \rangle \}$
- $\langle \text{подразбиращи-се-параметри} \rangle ::= \langle \text{празно} \rangle \mid \langle \text{параметър} \rangle = \langle \text{израз} \rangle \{ , \langle \text{параметър} \rangle = \langle \text{израз} \rangle \}$

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 = nullptr, 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

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)`

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!

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()` \iff `Rational(0,1)` (конструктор по подразбиране)

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()` \iff `Rational(0,1)` (конструктор по подразбиране)
 - `Rational(n)` \iff `Rational(n,1)`

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()` \iff `Rational(0,1)` (конструктор по подразбиране)
 - `Rational(n)` \iff `Rational(n,1)`
 - `Rational(n, d)`

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()` \iff `Rational(0,1)` (конструктор по подразбиране)
 - `Rational(n)` \iff `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

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&)`

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!

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) {}
```

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

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
- Конструктор за копиране **не се извиква** при:
 - предаване и връщане на обекти по указател
 - предаване и връщане на обекти по псевдоним

Копиране на обекти със статични полета

```
Player p1("Гандалф Сивия", 45); void anonymousPrint(Player p) {  
Player p2 = p1;                 p.setName("Анонимен");  
p2.setName("Гандалф Белия");   cout << "Играч:";  
anonymousPrint(p2);           p.print();  
                                }
```

Копиране на обекти със статични полета

```
Player p1("Гандалф Сивия", 45); void anonymousPrint(Player p) {  
Player p2 = p1;                    p.setName("Анонимен");  
p2.setName("Гандалф Белия");      cout << "Играч:";  
anonymousPrint(p2);               p.print();  
}
```

p1

Гандалф Сивия	45
---------------	----

Копиране на обекти със статични полета

```
Player p1("Гандалф Сивия", 45); void anonymousPrint(Player p) {  
Player p2 = p1;                    p.setName("Анонимен");  
p2.setName("Гандалф Белия");      cout << "Играч:";  
anonymousPrint(p2);              p.print();  
}
```

p1

Гандалф Сивия	45
---------------	----

p2

Гандалф Сивия	45
---------------	----

Копиране на обекти със статични полета

```
Player p1("Гандалф Сивия", 45); void anonymousPrint(Player p) {  
Player p2 = p1;                    p.setName("Анонимен");  
p2.setName("Гандалф Белия");      cout << "Играч:";  
anonymousPrint(p2);               p.print();  
}
```

p1

Гандалф Сивия	45
---------------	----

p2

Гандалф Белия	45
---------------	----

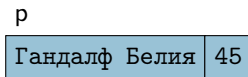
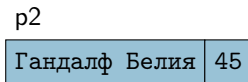
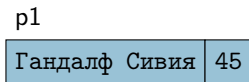
Копиране на обекти със статични полета

```

Player p1("Гандалф Сивия", 45);
Player p2 = p1;
p2.setName("Гандалф Белия");
anonymousPrint(p2);

void anonymousPrint(Player p) {
    p.setName("Анонимен");
    cout << "Играч:";
    p.print();
}

```



Копиране на обекти със статични полета

```

Player p1("Гандалф Сивия", 45);
Player p2 = p1;
p2.setName("Гандалф Белия");
anonymousPrint(p2);

void anonymousPrint(Player p) {
    p.setName("Анонимен");
    cout << "Играч:";
    p.print();
}

```

p1

Гандалф Сивия	45
---------------	----

p2

Гандалф Белия	45
---------------	----

p

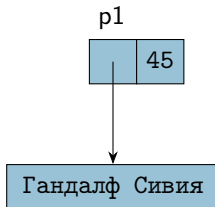
Анонимен	45
----------	----

Копиране на обекти с динамични полета

```
Player p1("Гандалф Сивия", 45); void anonymousPrint(Player p) {
Player p2 = p1;                   p.setName("Анонимен");
p2.setName("Гандалф Белия");     cout << "Играч:";
anonymousPrint(p2);             p.print();
                                }
```

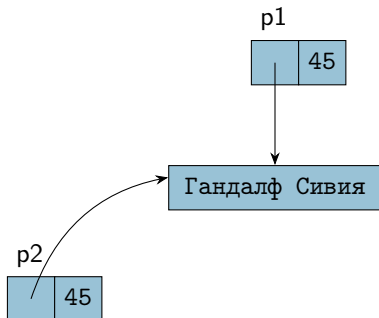

Копиране на обекти с динамични полета

```
Player p1("Гандалф Сивия", 45); void anonymousPrint(Player p) {  
Player p2 = p1;                   p.setName("Анонимен");  
p2.setName("Гандалф Белия");     cout << "Играч:";  
anonymousPrint(p2);              p.print();  
}
```



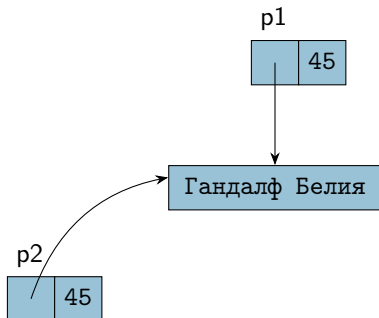
Копиране на обекти с динамични полета

```
Player p1("Гандалф Сивия", 45); void anonymousPrint(Player p) {  
Player p2 = p1;                   p.setName("Анонимен");  
p2.setName("Гандалф Белия");     cout << "Играч:";  
anonymousPrint(p2);              p.print();  
}
```



Копиране на обекти с динамични полета

```
Player p1("Гандалф Сивия", 45); void anonymousPrint(Player p) {  
Player p2 = p1;                   p.setName("Анонимен");  
p2.setName("Гандалф Белия");     cout << "Играч:";  
anonymousPrint(p2);              p.print();  
                                   }
```

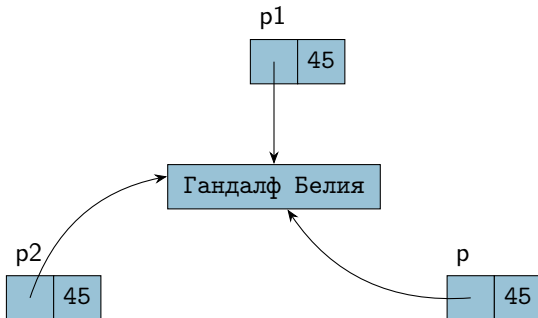


Копиране на обекти с динамични полета

```

Player p1("Гандалф Сивия", 45); void anonymousPrint(Player p) {
Player p2 = p1;                  p.setName("Анонимен");
p2.setName("Гандалф Белия");    cout << "Играч:";
anonymousPrint(p2);            p.print();
                                }

```

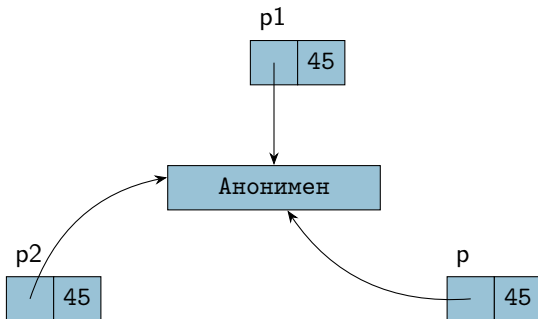


Копиране на обекти с динамични полета

```

Player p1("Гандалф Сивия", 45); void anonymousPrint(Player p) {
Player p2 = p1;                  p.setName("Анонимен");
p2.setName("Гандалф Белия");    cout << "Играч:";
anonymousPrint(p2);             p.print();
                                }

```



Конструктор за копиране на динамични полета

- Системният конструктор сяко 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
- **Пример:**

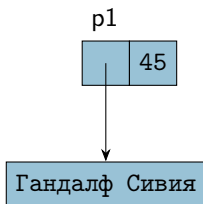
```
Player(Player const& p) : score(p.score) {  
    name = new char[strlen(p.name)+1];  
    strcpy(name, p.name);  
}
```

Коректно копиране на обекти с динамични полета

```
Player p1("Гандалф Сивия", 45); void anonymousPrint(Player p) {  
Player p2 = p1;                 p.setName("Анонимен");  
p2.setName("Гандалф Белия");   cout << "Играч:";  
anonymousPrint(p2);           p.print();  
                                }
```

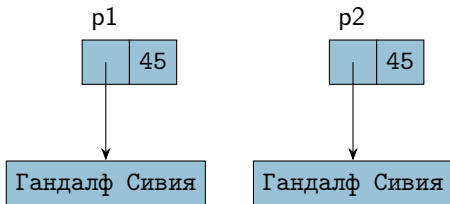
Коректно копиране на обекти с динамични полета

```
Player p1("Гандалф Сивия", 45); void anonymousPrint(Player p) {  
Player p2 = p1;                   p.setName("Анонимен");  
p2.setName("Гандалф Белия");     cout << "Играч:";  
anonymousPrint(p2);              p.print();  
}
```



Коректно копиране на обекти с динамични полета

```
Player p1("Гандалф Сивия", 45); void anonymousPrint(Player p) {  
Player p2 = p1;                   p.setName("Анонимен");  
p2.setName("Гандалф Белия");     cout << "Играч:";  
anonymousPrint(p2);              p.print();  
}
```

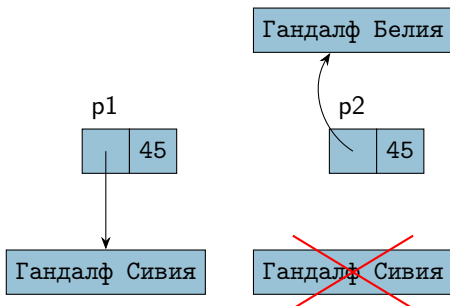


Коректно копиране на обекти с динамични полета

```

Player p1("Гандалф Сивия", 45); void anonymousPrint(Player p) {
Player p2 = p1;                   p.setName("Анонимен");
p2.setName("Гандалф Белия");     cout << "Играч:";
anonymousPrint(p2);              p.print();
                                  }

```

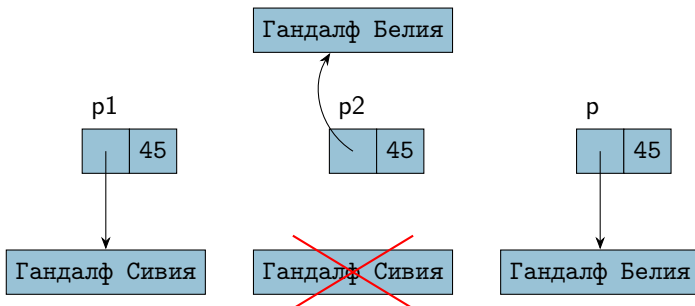


Коректно копиране на обекти с динамични полета

```

Player p1("Гандалф Сивия", 45); void anonymousPrint(Player p) {
Player p2 = p1;                  p.setName("Анонимен");
p2.setName("Гандалф Белия");    cout << "Играч:";
anonymousPrint(p2);             p.print();
                                }

```

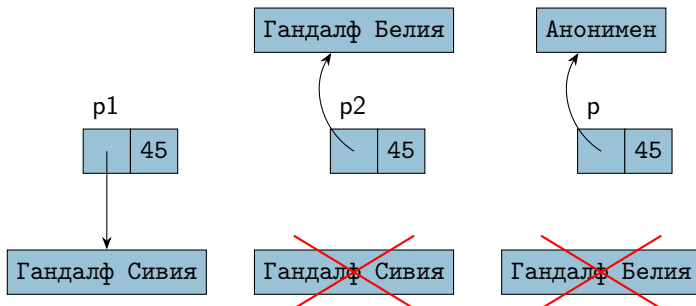


Коректно копиране на обекти с динамични полета

```

Player p1("Гандалф Сивия", 45); void anonymousPrint(Player p) {
Player p2 = p1;                  p.setName("Анонимен");
p2.setName("Гандалф Белия");    cout << "Играч:";
anonymousPrint(p2);             p.print();
                                }

```



Конструктор за преобразуване на тип

- Конструкторите с точно един параметър са специални
- `<име-на-клас> (<тип-за-преобразуване>)`

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

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 обект от клас А, но се подава стойност от тип В, С++ се опитва да използва конструктор за преобразуване на тип от вида `A(B)`

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 обект от клас А, но се подава стойност от тип В, С++ се опитва да използва конструктор за преобразуване на тип от вида А(В)
- **Примери:**

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 обект от клас А, но се подава стойност от тип В, С++ се опитва да използва конструктор за преобразуване на тип от вида $A(B)$
- **Примери:**
 - `Rational r = 5;` \iff `Rational r(5);` $r = \frac{5}{1}$

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 обект от клас А, но се подава стойност от тип В, С++ се опитва да използва конструктор за преобразуване на тип от вида А(В)
- **Примери:**
 - `Rational r = 5; \iff Rational r(5); r = $\frac{5}{1}$`
 - `add(3, Rational(2, 3)).print(); \iff
add(Rational(3), Rational(2, 3)).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 обект от клас A, но се подава стойност от тип B, C++ се опитва да използва конструктор за преобразуване на тип от вида A(B)

- **Примери:**

- `Rational r = 5; \iff Rational r(5); r = $\frac{5}{1}$`
- `add(3, Rational(2, 3)).print(); \iff
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
- $\langle \text{временен-обект} \rangle ::= \langle \text{име-на-клас} \rangle (\langle \text{параметри} \rangle)$

Временни обекти

- Конструкторите могат да се използват за създаване на временни анонимни обекти
- $\langle \text{временен-обект} \rangle ::= \langle \text{име-на-клас} \rangle (\langle \text{параметри} \rangle)$
- **Примери:**

Временни обекти

- Конструкторите могат да се използват за създаване на временни анонимни обекти
- `<временен-обект> ::= <име-на-клас> (<параметри>)`
- **Примери:**
 - `Rational(2, 3).print();`

Временни обекти

- Конструкторите могат да се използват за създаване на временни анонимни обекти
- `<временен-обект> ::= <име-на-клас> (<параметри>)`
- **Примери:**
 - `Rational(2, 3).print();`
 - `cout << add(Rational(1,2), Rational(1,4));`

Временни обекти

- Конструкторите могат да се използват за създаване на временни анонимни обекти
- `<временен-обект> ::= <име-на-клас> (<параметри>)`
- **Примери:**
 - `Rational(2, 3).print();`
 - `cout << add(Rational(1,2), Rational(1,4));`
- Тези обекти се създават само за да бъдат използвани веднага

Временни обекти

- Конструкторите могат да се използват за създаване на временни анонимни обекти
- `<временен-обект> ::= <име-на-клас> (<параметри>)`
- **Примери:**
 - `Rational(2, 3).print();`
 - `cout << add(Rational(1,2), Rational(1,4));`
- Тези обекти се създават само за да бъдат използвани веднага
- Временните обекти се унищожават непосредствено след като бъдат използвани

Обектите като член-данни

- Член-данните на даден клас биха могли да бъдат обекти от друг клас

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

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
- **Пример:**

```
class RationalPoint {  
    Rational x, y;  
    RationalPoint() {} // x = 0/1, y = 0/1  
};
```

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
- **Пример:**

```
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**

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
- **Пример:**

```
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**
- **Пример:** `RationalPoint(Rational p) : x(p), y(3, 5) {}`

Обектите като член-данни: копиране

- Системният конструктор за копиране автоматично извиква конструкторите за копиране на всички обекти член-данни

Обектите като член-данни: копиране

- Системният конструктор за копиране автоматично извиква конструкторите за копиране на всички обекти член-данни
- **Примери:**

Обектите като член-данни: копиране

- Системният конструктор за копиране автоматично извиква конструкторите за копиране на всички обекти член-данни
- **Примери:**
 - `RationalPoint p(Rational(2,3)); // p = (2/3, 3/5)`

Обектите като член-данни: копиране

- Системният конструктор за копиране автоматично извиква конструкторите за копиране на всички обекти член-данни
- **Примери:**
 - `RationalPoint p(Rational(2,3)); // p = (2/3, 3/5)`
 - `RationalPoint q = p; // q = (2/3, 3/5)`

Обектите като член-данни: копиране

- Системният конструктор за копиране автоматично извиква конструкторите за копиране на всички обекти член-данни
- **Примери:**
 - `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!

Обектите като член-данни: копиране

- Системният конструктор за копиране автоматично извиква конструкторите за копиране на всички обекти член-данни
- **Примери:**
 - `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!
- **Пример:**
`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> } }];`

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

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:**

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);`

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 обекти в масив

- Достъпът става по същия начин като с масиви от вграден тип

Достъп до обекти в масив

- Достъпът става по същия начин като с масиви от вграден тип
- **Примери:**

Достъп до обекти в масив

- Достъпът става по същия начин като с масиви от вграден тип
- **Примери:**
 - `a[2].print();`

Достъп до обекти в масив

- Достъпът става по същия начин като с масиви от вграден тип
- **Примери:**
 - `a[2].print();`
 - `cout << a[3].getDenominator();`

Достъп до обекти в масив

- Достъпът става по същия начин като с масиви от вграден тип
- **Примери:**
 - `a[2].print();`
 - `cout << a[3].getDenominator();`
 - `Rational r = a[1];`

Достъп до обекти в масив

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

Достъп до обекти в масив

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

Обекти в динамичната памет

- Три начина за създаване на обекти в динамичната памет:
- `new <клас>`
- `new <клас> (<параметри>)`
- `new <клас> [<брой>]`

Обекти в динамичната памет

- Три начина за създаване на обекти в динамичната памет:
- **new** <клас>
 - връща указател към нов обект, инициализиран с конструктор по подразбиране
- **new** <клас> (<параметри>)

- **new** <клас> [<брой>]

Обекти в динамичната памет

- Три начина за създаване на 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>]

Обекти в динамичната памет

- Три начина за създаване на 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