Обектно ориентирано програмиране

Класове продължение

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 CL
{
 public:
   CL(int, int, int);
   void print();
   ...
 private:
   int a, b, c;
};
CL::CL(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 CL може да се дефинира и по следния начин

```
CL::CL(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:**

```
CL::CL(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 …

- **Пример:** Ще дефинираме класа prat, като член-данна в него е обект на класа rat.

```
class prat
{
 public:
   prat(int, int, int);
   ...
 private:
   int a;
   rat r;
};
prat::prat(int x, int y, int z)
{
   a = x;
   r = rat(y, z);
}
```

Конструктори …

- Дефинираме обект q

```
prat q=prat(1,2,3);
```

- Преди да започне изпълнението на конструктора prat, автоматично се извиква конструкторът по подразбиране на rat и член-данната r на prat се инициализира с 0/1. Веднага след това r се свързва с обекта rat(y,z) за текущите y и z.
- По-ефективно е данната r да се свърже с правилната стойност направо, без междинна инициализация.

Конструктори …

- Това може да се реализира чрез дефиницията:

```
prat::prat(int x, int y, int z): r(rat(y, z))
{
    a = x;
}
```

или съкратено

```
prat::prat(int x, int y, int z): r(y, z)
{
    a = x;
}
```

Конструктори …

- **Предефинирани конструктори**
- 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

Конструктори …

- 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).

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 rat дефинирахме два конструктора rat() и rat(int, int), които се различават по броя на параметрите си.

Конструктори …

- 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. За целта е необходимо в класа да бъде дефиниран конструктор без параметри.
- **Пример:**

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

Конструктори …

- **Конструктори с 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.

Конструктори …

```
#include <iostream.h>
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;
}
```

Конструктори …

- В тази програма е дефинирана функцията f с три формални параметъра. От прототипа й се вижда, че два от тях (вторият и третият) са подразбиращи се със стойности по подразбиране 10 и “example 1” съответно. Тъй като в обръщенията 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 example 2
x = 1.5 y = 5 example 1
x = 1.5 y = 10 example 1
```

Конструктори …

- При използване на подразбиращи се параметри, важна роля играе редът на параметрите. Прието е, че ако параметър на функция е подразбиращ се, всички параметри след него също са подразбиращи се.
- Прототип на функция

  ```
  void f(double = 1.5, int, char* “example 1”);
  ```

  предизвиква грешка, тъй като първият формален параметър е обявен за подразбиращ се, а вторият не е такъв.
- 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.

Конструктори …

- Ако променим дефиницията на класа rat по следния начин:

```
class rat
{
 private:
    int numer;
    int denom;
 public:
    rat(int=0, int=1);
    void read();
    int get_numer() const;
    int get_denom() const;
    void print() const;
};
```

Конструктори …

- Допустими са следните дефиниции на обекти:

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

Конструктори …

- **Конструктори за присвояване и 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

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

  Чрез еквивалентните конструкции

  ```
  rat q = p;
  rat 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 присвояване**.

Конструктори …

- Конструкторът за присвояване е конструктор, поддържащ формален параметър от тип <име_на_клас> const &.
- Ако в един клас явно не е дефиниран конструктор за присвояване, 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 за присвояване се нарича конструктор за копиране.
- **Пример:** В класа rat не беше дефиниран конструктор за присвояване. Затова при създаване на обект p чрез дефиницията

```
rat p = q;
```

автоматично се извиква конструкторът за копиране.

Конструктори …

- Последният има вида:

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

- Дефиницията rat p=q създава нов обект p *(без викане на конструктор)*, в който се копират съответните стойности на обекта q.
- Ако в класа е дефиниран конструктор за присвояване, компилаторът го използва.

Конструктори …

- **Пример:**

```
class rat
{
 private:
    int numer;
    int denom;
 public:
    rat(rat const &); // конструктор за присвояване
    rat(int=0, int=1);
    void read();
    int get_numer() const;
    int get_denom() const;
    void print() const;
};
```

Конструктори ...

```
...
rat::rat(rat const & r)
{
   numer = r.numer + 1;
   denom = r.denom + 1;
}
...
rat 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.
```

Конструктори …

- *Предефиниране на служебния конструктор за копиране се налага когато обектите имат член-данни, указващи динамична памет.*
- Освен в горните случаи, конструкторът за присвояване (копиране) се използва и при предаване на обект като аргумент на функция, когато предаването е по стойност, а също 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).
- Когато обектите се предават като параметри на функции, спецификаторите на достъп public и private имат същия смисъл, т.е. функциите имат достъп само до public компонентите на обектите, когато им се подават като параметри.

Конструктори …

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

а)

```
rat sum(rat r1, rat r2)
{
   rat r(r1.get_numer()*r2.get_denom()+
         r2.get_numer()*r1.get_denom(),
         r1.get_denom()*r2.get_denom());
 return r;
}
```

Обръщението sum(p, q).print() извежда 8/7. Този резултат може да се обясни по следния начин. Тъй като r1 и r2 са параметри стойности, те се свързват с фактическите си параметри чрез присвояване. В резултат r1 се свързва с 1/2 (не с 0/1), а r2 – с 2/3 (не с1/2).

Конструктори …

След изпълнението на инициализацията

```
rat r(r1.get_numer()*r2.get_denom()+
      r2.get_numer()*r1.get_denom(),
      r1.get_denom()*r2.get_denom());
```

/чрез двуаргументния конструктор rat(int, int)/ обектът r се свързва със 7/6. Тъй като функцията rat е от тип rat, при изпълнение на return r; се прави още едно прилагане на глупавото присвояване и се получава 8/7.

б)

```
rat sum(rat const & r1, rat const & r2)
{
   rat r(r1.get_numer()*r2.get_denom()+
         r2.get_numer()*r1.get_denom(),
         r1.get_denom()*r2.get_denom());
   return r;
}
```

Конструктори …

Сега обръщението sum(p, q).print() извежда 2/3. Този резултат може да се обясни по следния начин. Тъй като r1 и r2 са параметри псевдоними, те директно се свързват с фактическите си параметри (не се извършва присвояване), т.е. r1 се свързва с 0/1, а r2 – 1/2. След изпълнението на инициализацията

```
rat r(r1.get_numer()*r2.get_denom()+
      r2.get_numer()*r1.get_denom(),
      r1.get_denom()*r2.get_denom());
```

r се свързва със 1/2. Аналогично на случай а), при изпълнение на return r; се прилагане “глупавото” присвояване и се получава 2/3.

Конструктори …

в)

```
rat sum(rat* r1, rat* r2)
{
   rat r(r1->get_numer()*r2->get_denom()+
         r2->get_numer()*r1->get_denom(),
         r1->get_denom()*r2->get_denom());
   return r;
}
...
rat* p1 = &p,
   * q1 = &q;
```

Обръщението sum(p1, q1).print() извежда 2/3. Този резултат може да се обясни по следния начин. Тъй като r1 и r2 са параметри указатели, те се свързват с фактическите си параметри чрез адрес. В резултат r1 се свързва с 0/1, а r2 – с 1/2. След изпълнението на инициализацията r се свързва със 1/2. Тъй като функцията rat е от тип rat, при изпълнение на return r; се прилага присвояването 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:**

```
rat p;
rat * ptr = &p;
```

  В резултат ще се отделят 4В ОП, които ще се именуват с ptr и ще се инициализират с адреса на обекта p.
- Достъпът до компонентите на рационалното число, сочено от ptr, се осъществява по общоприетия начин:

```
(*ptr).get_numer()
(*ptr).get_denom()
```

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

```
ptr -> get_numer()
ptr -> get_denom()
```

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

Масиви и обекти

- Елементите на масив могат да са обекти, но разбира се от един и същ клас. Дефинират се по общоприетия начин.
- **Дефиниция на масив от обекти**

  <дефиниция_на_променлива_от_тип_масив_от_обекти> ::=

  T <променлива>[size] [= {<инициализиращ_списък>}];

  където

  - T е име или декларация на клас;
  - <променлива> е идентификатор;
  - size е константен израз от интегрален или изброен тип с *положителна* стойност;
  - <инициализиращ_списък> се дефинира по следния начин:

    <инициализиращ_списък> ::= <стойност>{, <стойност>}

    {, <име_на_конструктор>(<фактически_параметри>)}

Масиви и обекти …

- **Пример:**

```
rat table[10];
```

определя масив от 10 обекта от клас rat.

- Достъпът до елементите на масива е пряк и се осъществява по стандартния начин – чрез индексирани променливи.
- **Пример:** Чрез индексираните променливи

```
table[0], table[1], ..., table[9]
```

се осъществява достъп до първия, втория и т.н. до десетия елемент на table.

Тъй като table[i] (i = 0, 1, ..., 9) са обекти, възможни са следните обръщения към техни компоненти:

Масиви и обекти …

```
table[i].read();       // въвежда стойност на table[i]
table[i].print();      // извежда стойността на table[i]
table[i].get_numer(); // намира числителя на table[i]
table[i].get_denom(); // намира знаменателя на table[i].
```

- Връзката между масиви и указатели е в сила и в случая когато елементите на масива са обекти. Името на масива е указател към първия му елемент, т.е. ако

```
rat * p = table;       // p сочи към table[0]
                       // т.е. p==&table[0]
   // *(p+i) == table[i], i = 0, 1,...,9
```

Тогава

```
(*(p+i)).print();  // е еквивалентно на table[i].print();
```

Масиви и обекти …

- Масивът може да е член-данна на клас.
- **Пример:**

```
class example
{
   int a;
   int table[10];
 public:
   int array[10];
} x[5];
```

дефинира масив с 5 компоненти, които са от тип example. Достъпът до компонентите на масива array ще се осъществи по следния начин:

`x[i].array[j], i = 0, 1, ..., 4; j = 0, 1, ..., 9.`

Масиви и обекти …

- Конструкторите (в частност конструкторът по подразбиране) играят важна роля при дефинирането и инициализирането на масиви от обекти. Масив от обекти, дефиниран в програма, се инициализира по два начина:
  - *неявно* (чрез извикване на системния конструктор по подразбиране за всеки обект – елемент на масива);
  - *явно* (чрез инициализиращ списък).
- **Примери:**

а) Класът

```
const NUM = 5;
class student
{
 public:
   void read_student();
   void print_student() const;
   bool is_better(student const &) const;
   double average() const;
```

Масиви и обекти …

```
 private:
    int facnom;
    char name[26];
    double marks[NUM];
};
```

няма явно дефиниран конструктор. Дефиницията

```
student table[30];
```

на масива table от 30 обекта от клас student е правилна.

Инициализацията се осъществява чрез извикване на “системния” конструктор по подразбиране за всеки обект – елемент на масива.

Масиви и обекти …

б) Класът rat, дефиниран по-долу

```
class rat
{
 private:
    int numer;
    int denom;
 public:
    rat(int=0, int=1);
    void read();
    int get_numer() const;
    int get_denom() const;
    void print() const;
};
```

Масиви и обекти …

притежава явно дефиниран конструктор с два подразбиращи се параметъра. В този случай са допустими дефиниции от вида:

```
rat x[10]; // x[i] се инициализира с 0/1, за всяко i=0,1,...9.
rat x[10] = {1,2,3,4,5,6,7,8,9,10}; //x[i] == i+1/1
rat x[10] = {rat(1,21),rat(2),rat(3, 5),4,5,6,7,8,9,10};
   // x[0] == 1/21; x[1] == 2/1; x[2] == 3/5, x[3]== 4/1, ...
```

- Ако променим конструктора на класа rat от

```
rat(int=0, int=1);
```

в

```
rat(int, int);
```

т.е. без подразбиращи се параметри и трите дефиниции от по-горе ще съобщят за грешка. Единствено допустима дефиниция на x[10] е с инициализация с 10 обръщения към двуаргументния конструктор rat с явно указани два аргумента.

Стек

- Стекът е линейна динамична структура от данни.
- **Задача.** Да се напише програма, която извежда двоичното представяне на естествено число.

  Операторът

```
do
{
    cout << k%2;
    k/=2;
} while (k);
```

  извежда двоичното представяне на числото k, но в обратен ред. За решаване на задачата ще използваме динамичната структура от данни **стек**.

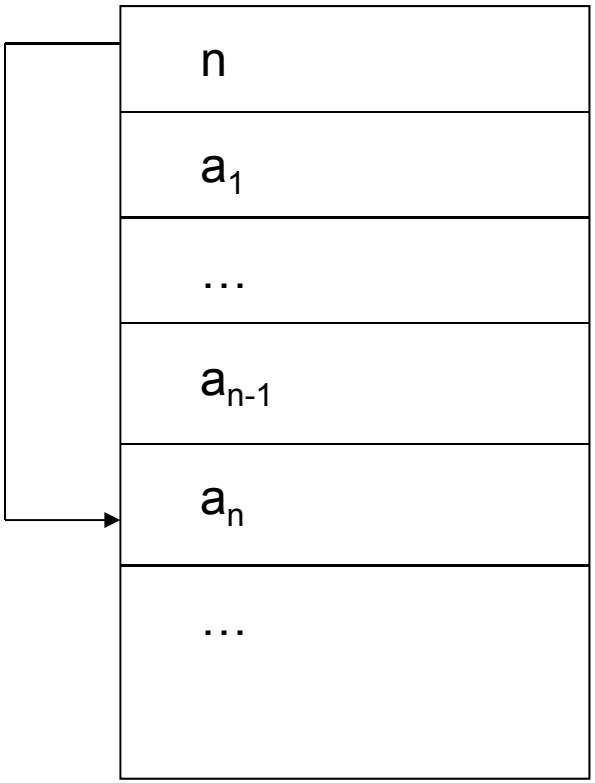
Стек …

- Стекът е крайна редица от елементи от един и същ тип. Операциите включване и изключване на елемент са допустими само за единия край на редицата, който се нарича **връх на стека**. Възможен е достъп само до елемента, намиращ се на върха на стека като достъпът е пряк.
- При тази организация на логическите операции, последният включен елемент се изключва пръв. Затова стекът се определя още като структура "*последен влязъл – пръв излязъл*" (FIFO).
- Широко се използват два основни начина за физическо представяне (представяне в ОП) на стек: *последователно* и *свързано*. За целите на тази задача ще използваме последователното представяне.
- При това представяне се запазва блок от паметта, вътре в който стекът да расте и да се съкращава. Ако редицата от елементи от един и същ тип

  $a_n, a_{n-1}, ..., a_1$

  е стек с връх a_n, последователното представяне на стека има вида:

Стек …


n
a_1
…
a_{n-1}
a_n
…
Указател към върха на стека
Връх на стека
Неизползвана част

Стек …

- При включване на елементи в стека, те се поместват в последователни адреси в неизползваната част веднага след върха на стека.
- Това физическо представяне ще реализираме като използваме структурата от данни масив. За указател към върха на стека ще служи цяла променлива. Ще използваме подхода абстракция със структури от данни. Първите две нива на абстракция реализираме чрез класа stack:

Стек ...

```
const int NUM = 100;
class stack
{public:
  stack();
  void push(int);
  void pop();
  void print();
  int top() const;
  bool empty() const;
 private:
  int n;              // указател към върха на стека
  int arr[NUM];       // представяне на стека
};
```

Стек …

- Примитивните операции са реализирани чрез следните член-функции:
  - void push(int); - включва елемент в стека
  - void pop(); - изключва елемент от стека
  - void print(); - извежда елементите на стека като разрушава стека
  - int top() const; - намира елемента от върха на стека
  - bool empty() const; - проверява дали стекът е празен

Стек …

```
#include <iostream.h>
const NUM = 100;
class stack
{public:
  stack();
  void push(int);
  void pop();
  int top() const;
  bool empty() const;
  void print();
 private:
  int n;
  int arr[NUM];
};
```

Стек …

```
stack::stack()
{
   n = 0;
   arr[0]=0;
}
void stack::push(int x)
{
   n++;
   arr[n] = x;
}
void stack::pop()
{
   if (!empty()
      n--;
   else
      cout << “Empty stack!\n”;
}
```

Стек ...

```
int stack::top() const
{
   return arr[n];
}
bool stack::empty() const
{
   return n == 0;
}
void stack::print()
{ while (!empty())
  {
     cout << top();
     pop();
  }
  cout << endl;
}
```

Стек ...

```
// конструира стек от двоичното представяне
// на указано цяло число
stack num_stack(int x)
{
    stack st;
    while (x)
    {
        st.push(x%2);
        x/=2;
    }
    return st;
}
```

Стек ...

```
void main()
{
    cout << "number: ";
    int n;
    cin >> n;
    num_stack(n).print();
}
```

Динамични обекти

- Вече разгледахме в най-общ план разпределението на ОП по време на изпълнението на програма. Всяка програма има три “места” за памет:
  - *област на статичните данни*
  - *програмен стек (стек)*
  - *област за динамичните данни (динамична памет, heap).*
- ***Стекът*** е област за временно съхранение на информация. Той е кратковременна памет. C++ използва стека основно за реализиране на обръщения към функции. Всяко обръщение към функция предизвиква конструиране на нова стекова рамка, която се установява на върха на стека. По такъв начин когато функция A извика функция B, която от своя страна вика функция C, стекът нараства. Когато пък всяка от тези функции завършва, стековите рамки на тези функции автоматично се разрушава. Така стекът се свива.

Динамични обекти …

- ***Хийпът*** е по-постоянна област за съхранение на данни. Той е един вид дълготрайна памет. Особеност на тази памет е, че тя не се свързва с имена на променливи. С разположените в нея обекти се работи косвено – чрез указатели. Обикновено се използва при работа с т. нар. **динамични структури от данни**.
- *Динамичните данни са такива обекти (в широкия смисъл на думата), чийто брой не е известен в момента на проектирането на програмата. Те се създават и разрушават по време на изпълнението на програмата. След разрушаването им, заетата от тях памет се освобождава и може да се използва отново. Така паметта се използва по-ефективно.*

Динамични обекти …

- Използването на динамичната памет досега не се налагаше, тъй като структурите от данни, с които работехме, бяха статични. По нататък ще дефинираме и използваме динамичните структури от данни свързан списък, стек, опашка, дърво, граф и др., използването на тези средства е задължително.
- Създаването и разрушаването на динамични обекти в C++ се осъществява чрез операторите new и delete.
- Извикването на new заделя в хийпа необходимата памет и връща указател към нея. Този указател може да се съхрани в някаква променлива и да се пази докато е необходимо. За разлика от стека, заделянето на памет в хийпа е явно – чрез new.

Динамични обекти …

- Освобождаването на паметта от хийпа също става явно, чрез delete. Всяко извикване на new трябва да бъде балансирано чрез извикване на delete. Последното се налага, тъй като за разлика от стека, хийпът не се изчиства автоматично. В C++ няма система за “събиране на боклуци”. Затова трябва явно да бъдат изтрити създадените в хийпа обекти.
- **Оператор new**

  Синтаксис

  new <име_на_тип> [[size]] |

  new <име_на_тип> (<инициализация>)

  където
  - *<име_на_тип>* е име на някой от стандартните типове int, double, char и др. или е име на клас;
  - *size* е израз с произволна сложност, но трябва да може да се преобразува до цял. Показва броя на компонентите от тип <име_на_тип>, за които да се задели памет в хийпа и се нарича **размерност**;
  - *<инициализация>* е израз от тип <име_на_тип> или инициализация на обект според синтаксиса на конструктора на класа, ако <име_на_тип> е име на клас.

Динамични обекти …

Семантика

Заделя в хийпа (ако е възможно):

- sizeof(<име_на_тип>) байта, ако не са зададени size и <инициализация> или
- sizeof(<име_на_тип>)*size байта, ако явно е указан size или
- sizeof(<име_на_тип>) байта, ако е специфицирана <инициализация>, която памет се инициализира с <инициализация>

и връща указател към заделената памет.

Динамични обекти …

- **Забележка:**
  - Ако <име_на_тип> е име на клас и след него има кръгли скоби, в тях трябва да стоят фактически параметри (аргументи) на конструктора на класа.
  - Ако скобите липсват, класът трябва да притежава конструктор по подразбиране или да няма явно дефиниран конструктор.
  - Ако след името на класа е поставен заграден в квадратни скоби израз, new заделя място за масив от обекти на указания клас и извиква конструктора по подразбиране за инициализиране на отделената памет.
- **Примери:**

а) `int* q = new int(2+5*5);`

отделя (ако е възможно) 4B памет в хийпа, инициализира я с 27 - стойността на израза 2+5*5 и свързва q с адреса на тази памет

Динамични обекти …

б) `int* p = new int[10];`

отделя (ако е възможно) 40B в хийпа (за 10 елемента от тип int) и свързва p с адреса на тази памет

в) `rat* r = new rat(1,5);`

отделя памет в хийпа за един обект от клас rat, свързва r с адреса на тази памет и извиква конструктора rat(1,5) за да я инициализира

г) `rat* r = new rat;`

отделя памет в хийпа за обект от тип rat, записва адреса на тази памет в r и извиква конструктора по подразбиране на класа rat за инициализиране на тази памет

д) `rat* r = new rat[10];`

отделя памет в хийпа за 10 обекта от класа rat, записва адреса на тази памет в r, извиква конструктора по подразбиране на класа rat и инициализира отделената памет;

Динамични обекти …

е) `rat** parr = new rat*[5];`

отделя 20B памет в хийпа за масив от 5 указателя към стойности от тип rat и записва в parr адреса на тази памет

- Заделянето на памет по време на компилация се нарича **статично** заделяне на памет, заделянето на памет по време на изпълнение на програмата - **динамично разпределение на паметта**. Паметта за променливите q, p, r и parr, от примерите по-горе, е заделена статично, а всяка една от тези променливи има за стойност адрес от хийпа. Казва се още, че q, p, r и parr адресират динамична памет.

Динамични обекти …

- Под период на **активност на една променлива** се разбира частта от времето за изпълнение на програмата, през което променливата е свързана с конкретно място в паметта.
- Паметта за глобалните променливи се заделя в началото и остава свързана с тях до завършването на изпълнението на програмата.
- Паметта за локалните променливи се заделя при влизане в локалната област и се освобождава при напускането ѝ.
- Паметта на динамичните променливи се заделя от оператора new. Заделената по този начин памет остава свързана със съответната променлива докато не се освободи явно от програмиста. Явното освобождаване на динамична променлива се осъществява чрез оператора delete, приложен към указателя, който адресира съответната променлива.

Динамични обекти …

- **Оператор delete**

  Синтаксис

  delete <указател_към_динамичен_обект>;

  където <указател_към_динамичен_обект> е указател към динамичен обект (в широкия смисъл на думата), създаден чрез оператора new.

  Семантика

  Разрушава обекта, адресиран от указателя, като паметта, която заема този обект, се освобождава. Ако обектът, адресиран от указателя, е обект на клас, отначало се извиква деструкторът на класа и след това се освобождава паметта.

Динамични обекти …

- ко в хийпа е заделена памет, след което тази памет не е освободена чрез delete, се получава загуба на памет. Парчето памет, което не е освободено, е като остров в хийпа, заемащо пространство, което иначе би могло да се използва за други цели.
- За да се разруши масив, създаден чрез new по следния начин:

```
int* arr = new int[5];
```

  трябва да се запише:

```
delete [] arr;
```

- **Забележка:** Някои реализации на езика допускат разрушаването в горния случай да стане и чрез `delete arr`.
- Ако обаче масивът съдържа в себе си указатели, първо трябва да бъде обходен и да бъде извикан операторът delete за всеки негов елемент.

Динамични обекти …

- Операторът delete трябва да се използва само за освобождаване на динамична памет, заделена с new. В противен случай действието му е непредсказуемо. Няма забрана за прилагане на delete към указател със стойност 0. Ако стойността на указателя е 0, той е свободен и не адресира нищо.
- **Пример:**

```
void example()
{...
 int a = 7;
 char *str = “abv”;
 int *pa = &a;
 rat *ptr = 0;
 double *x = new double;
```

Динамични обекти …

```
   delete str; //некоректно обръщение,
               //str не е адресирано чрез new
   delete pa;  //некоректно обръщение,
               //pa не е адресирано чрез new
   delete ptr; //некоректно обръщение,
               //ptr не е адресирано чрез new
   delete x;   // коректно обръщение
  ...
}
```

- Динамичната памет не е неограничена. Тя може да се изчерпи по време на изпълнение на програмата. Ако наличната в момента динамична памет е недостатъчна, new връща нулев указател. Затова се препоръчва след всяко извикване на new да се прави проверка за успешността ѝ.

Динамични обекти …

- Чрез оператора new могат да се създават т. нар. **динамични масиви** – масиви с променлива дължина. Динамичните масиви се създават в динамичната памет. Следващата програма илюстрира този процес.
- **Задача.** Да се напише програма, която създава динамичен масив от цели числа. Да се изведе масивът.

```
#include <iostream.h>
int main()
{
   int size; // дължина на масива
   do
   {
      cout << "size of array: ";
      cin >> size;
   } while (size<1);
```

Динамични обекти …

```
    // създаване на динамичен масив arr от size
    // елемента от тип int
    int* arr = new int[size];
    int i;
    for (i = 0; i <= size-1; i++)
       arr[i] = i;
    // извеждане на елементите на arr
    for (i = 0; i <= size-1; i++)
       cout << arr[i] << " ";
    cout << endl;
    // освобождаване на заетата динамична памет
    delete [] arr;
    return 0;
}
```

Динамични обекти …

- Методите на класовете също могат да използват динамична памет, която се заделя и освобождава по време на изпълнението им, чрез операторите new и delete.

```
class product
{private:
   char* name;
   double price;
    ...
public:
   void read();
   void print() const;
    ...
};
```

Динамични обекти …

```
void product::read()
{
    static char s[40];
    cout << "name: ";
    cin >> s;
    name = new char[strlen(s)+1];
    strcpy(name, s);
    cout << "price: ";
    cin >> price;
    ...
}
```

Динамични обекти …

- Ще отбележим също, че заделената от член-функциите динамична памет не се освобождава автоматично при разрушаване на обектите на класове. Освобождаването на тази памет трябва да стане явно чрез оператора delete, който трябва да се изпълни преди разрушаването на обекта. Този процес може да бъде автоматизиран чрез използване на специален вид методи, наречени **деструктори**.

Деструктори

- Разрушаването на обекти на класове в някои случаи е свързано с извършване на определени действия, които се наричат **заключителни**. Най-често тези действия са свързани с освобождаване на заделена преди това динамична памет, възстановяване на състояние на програмата и др. Ефектът от заключителните действия е противоположен на ефекта на инициализацията. Естествено е да се даде възможност заключителните действия да се извършат автоматично при разрушаването на обекта. Това се осъществява от деструкторите.
- Деструкторът е член-функция, която се извиква при:
  - разрушаването на обект чрез оператора delete,
  - излизане от блок, в който е бил създаден обект от класа.

Деструктори …

- Един клас може да има явно дефиниран точно един деструктор. Името му съвпада с името на класа, предшествано от символа ‘~’ (тилда), типът му е void и явно не се задава в заглавието. Деструкторът няма формални параметри.
- **Забележка:** Използването на явно дефинирани деструктори не винаги е належащо, тъй като всички член-променливи се разрушават при разрушаването на обекта и без използването на деструктор. Ако конструкторът или някоя член-функция реализира динамично заделяне на памет за някоя член-данна, използването на деструктор е задължително, тъй като в този случай той трябва да освободи заетата памет.

Деструктори …

```
class product
{
 private:
   char* name;
   double price;
   ...
 public:
   product();
   ~product();
   void print() const;
   char* get_name() const;
   ...
};
```

Деструктори ...

```
product::product()
{
    cout << "name: ";
    cin >> s;
    name = new char[strlen(s)+1];
    strcpy(name, s);
    cout << "price: ";
    cin >> price;
    ...
}
product::~product()
{
    delete name;
}
```

Деструктори …

```
void product::print() const
{
   cout << setw(25) << name
        << setw(10) << price;
}
char* product::get_name() const
{
   return name;
}
…
void sorttable(int n, product* a[])
{
…
}
```

Деструктори …

```
int main()
{
   cout << "size: "; // размерност на масива
   int size;
   cin >> size;
   //създава динамичен масив от size обекта на product
   product* table = new product[size];
   // заделя памет за динамичен масив от указатели
   // към size обекта на product
   product** ptable = new product*[size];
   int i;
   cout << "table: \n";
   for (i = 0; i < size; i++)
   {
      table[i].print();
      cout << endl;
      ptable[i] = &table[i];
   }
```

Деструктори …

```
    sorttable(size, ptable);
    cout << "\n New Table: \n";
    for (i = 0; i < size; i++)
    {
       ptable[i]->print();
    }
    delete [size] table; // някои реализации допускат
                         // пропускането на size
    delete [] ptable;    // някои реализации допускат
                         // пропускането на []
    return 0;
}
```

Деструктори …

```
class stack
{
 public:
   stack(int sz = 100);
   ~stack();
   void push(int);
   int pop();
   int top() const;
   bool empty() const;
   void print();
  private:
   int n;
   int *arr;
   int size;
};
```

Деструктори …

```
stack::stack(int sz = 100)
{
   arr = new int[sz];
   n = 0;
}
stack::~stack()
{
   deelete [] arr;
}
void stack::push(int x)
{
   arr[n] = x;
   n++;
}
int stack::pop()
{
   return arr[--n];
}
```

Деструктори ...

```
int stack::top() const
{
   return arr[n-1];
}
bool stack::empty() const
{
   return n == 0;
}
void stack::print()
{
   for (int i = n-1; i >= 0; i--)
      cout << arr[i];
   cout << endl;
}
```

Деструктори ...

```
void main()
{
    stack a(200);
    for (int i = 1; i <= 50; i++)
       a.push(i);
    a.print();
    stack *p = new stack;
    while (!a.empty())
       p->push(a.pop());
    p->print();
    delete p;
}
```

Деструктори …

- **Забележка:** Ако се освобождава памет, заета от динамичен масив, чийто елементи са обекти на клас, трябва явно да се посочи дължината на масива. Тя е необходима за да се определи броят на извикванията на деструктора.
- По повод на това, че за всяко обръщение към new трябва да има съответен delete, възниква въпросът: *Когато функция върне указател или псевдоним към обект, създаден чрез new, кой носи отговорността за извикването на delete за този указател?*

Деструктори …

```
struct object
{
   int a, b;
}
object& myfunc();

int main()
{
   object& rmyobj = myfunc();
   cout << rmyobj.a << rmyobj.b << endl;
   return 0;
}

object& myfunc()
{
   object *o = new object;
   o->a = 20;
   o->b = 40;
   return *o;       // връща самия обект
}
```

Деструктори …

- Например, къде в програмния фрагмент да бъде изтрит rmyobj?
- Функцията, която създава указателя или псевдонима нищо не може да направи, защото когато указателят или псевдонимът бъде върнат, тя вече ще е завършила. Така че, който е извикал функцията, той след това трябва да извика и delete. Възможно е извикващият да е програма, принадлежаща на друг програмист или ваша стара програма и да не помните тази подробност.
- Затова като цяло *се смята за лошо програмиране връщането на указател, който после трябва да бъде изтрит. По-добре е да се върне обекта по стойност.* В примерната програма по-горе в main, след извеждането на rmyobj трябва да се включи операторът delete &rmyobj.

Създаване и разрушаване на обекти на класове

- Съществуват два начина за създаване на обекти:
  - чрез дефиниция;
  - чрез функциите за динамично управление на паметта.
- **В първия случай** обектът се създава при срещане на дефиницията (във функция или блок) и се разрушава при завършване на изпълнението на функцията или при излизане от блока. Дефиницията може да бъде поставена където и да е в тялото на функцията или блока, като пред и след нея може да има изпълними оператори. Дефиницията, чрез която се създава обект, може да бъде допълнена с инициализация, която може да се основава на извикване на обикновен конструктор или на конструктора за присвояване.

Създаване и разрушаване на обекти на класове …

- Разрушаването на обекта е свързано с извикването на деструктора на класа, ако такъв явно е дефиниран или с извикването на “системния” деструктор (деструктора по подразбиране), ако в класа явно не е дефиниран деструктор.
- **Пример:** Нека в класа rat добавим деструктора

```
rat::~rat()
{
    cout << “destruct number: “
         << number << “/”
         << denum << endl;
}
```

Създаване и разрушаване на обекти на класове …

```
void main()
{
   rat p(1,8);      // създава обект p и го инициализира с 1/8
   rat q=rat(2,9); // създава обект q и го инициализира с 2/9
   for(int i=1; i<=5; i++)
   {
      rat r(i, i+1); // създава обект r и го инициализира
                     // с i/(i+1)
      r.print();     // за i = 1, ..., 5
   }
   p.print();
   q.print();
}
```

Създаване и разрушаване на обекти на класове …

- В резултат от изпълнението на main се получава:

```
1/2
destruct number:1/2
2/3
destruct number:2/3
3/4
destruct number:3/4
4/5
destruct number:4/5
5/6
destruct number:5/6
1/8
2/9
destruct number:2/9
destruct number:1/8
```

Създаване и разрушаване на обекти на класове …

- От изпълнението се вижда, че деструкторът на класа rat е извикан толкова пъти, колкото пъти са извършвани дейности по създаване на обекти на класа rat. Пътвите пет извиквания на деструктора са при завършване на изпълнението на блока на оператора for, а последните две – при завършване на изпълнението на функцията main.
- **Във втория случай** създаването и разрушаването на обекти се управлява от програмиста. Създаването става с new, а разрушаването чрез delete. Операциите new се включват в конструкторите, а операциите delete – в деструктора на класа.

Създаване и разрушаване на обекти на класове …

- **Пример:**

```
rat *p = new rat(3,7); // търси в хийпа 8B, свързва адреса
                       // им с p, извиква конструктора
                       // rat(9,13) и инициализира паметта
(*p).print();
delete p;              // извиква деструктора, след което
                       // разрушава обекта
...
```

- В този случай деструкторът само регистрира присъствието си. Получаваме:

```
3/7
destruct number: 3/7
```

Инициализиране на обекти на класове

- Езикът C++ позволява обектите на класове (както и обикновените променливи) да бъдат инициализирани при дефиницията си и при извикването на функцията new. При обикновените променливи инициализаторът задава стойност на променливата, а при обектите - осигурява аргументи на конструкторите. Инициализацията на обект на клас се извършва по следните начини:

  <име_на-клас> <обект>(<инициализатор>); |

  <име_на-клас> <обект> = <инициализатор>;

- Възможни са:

а) инициализаторът не е обект на класа

В този случай се създава обекта, след това се намира стойността на израза–инициализатор и се подава на подходящия конструктор (ако има такъв).

Инициализиране на обекти на класове …

- **Пример:** Нека в класа rat конструкторът е с прототип:

```
rat(int = 0; int = 1);
```

Инициализацията

```
rat p = 7;
```

ще създаде обекта p и ще извика конструктора rat(7), с който ще инициализира p. Ако в класа rat не беше дефиниран конструктор с един аргумент, опитът за тази инициализация щеше да е неуспешен.

б) инициализаторът е обект на класа

В този случай съществуват някои особености. Ако съществува явно дефиниран конструктор за присвояване, той се използва. В противен случай се използва подразбиращия се конструктор за копиране.

Инициализиране на обекти на класове …

- *Конструктор за присвояване явно не е дефиниран*

  Тогава се извиква подразбиращия се системен конструктор за копиране.
- **Пример:**

```
rat p(1,9);
rat q = p;
```

  Създава се нов обект q с член-данни абсолютни копия на съответните член-данни на p.
- В този случай възникват проблеми ако някоя член-данна на обекта е указател към динамичната памет, тъй като член-променливата на новия обект, който е указател към динамичната памет, е със същата стойност като на стария (указва към същата памет). В този случай става поделяне на компонента на обектите.

Инициализиране на обекти на класове …

```
class product
{
 private:
   char* name;
   double price;
   ...
 public:
   product();
   ~product();
   void print() const;
   char* get_name() const;
   ...
};
```

Инициализиране на обекти на класове …

```
product::product()
{
   cout << "name: ";
   cin >> s;
   name = new char[strlen(s)+1];
   strcpy(name, s);
   cout << "price: ";
   cin >> price;
   ...
   cout << "new data: " << this << endl;
}
product::~product()
{
   delete name;
   cout << "destruct data for: " << this << endl;
}
```

Инициализиране на обекти на класове …

```
void main()
{
   product p;
   product q = p;
}
```

- В този случай дефинираният конструктор по подразбиране product() се извиква веднъж – при инициализирането на p. Тъй като няма явно дефиниран конструктор за присвояване, генерираният от системата конструктор за копиране откопирва член-данните на обекта p в q като член-данната name е поделена. При завършване на блока – тяло на main, първо се разрушава обектът q. За него се извиква деструкторът. Поделената памет се освобождава, след което се разрушава и q. Забележете, q е разрушен, но е разрушена и част от p - поделената динамична памет. После започва процедурата по разрушаването и на обекта p. Извиква се деструкторът, който се опитва да освободи вече освободена памет. Това води до грешка, чийто последствия са непредвидими.

Инициализиране на обекти на класове …

- *Конструктор за присвояване явно е дефиниран*

  Вече дефинирахме един глупав конструктор за присвояване за класа rat и правихме експерименти с него. Ще напомним, че конструкторът за присвояване е член-функция от вида:

  <име_на_клас>(<име_на_клас> const&)

  {<тяло>}
- Ще дефинираме подходящ конструктор за присвояване в класа product и ще го извикаме за да реализираме коректно инициализацията от последния пример.

Инициализиране на обекти на класове …

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

и включваме прототипа му

```
   product(product const & p);
```

в public-частта на класа product.

Тогава функцията

```
void main()
{
   product p;
   product q = p;
}
```

вече работи добре.

Инициализиране на обекти на класове …

- Дефинираният конструктор за присвояване решава проблемите, възникващи при инициализацията на обект на клас product. Използва се при предаване на обект по стойност, а също при връщане на обект като стойност на функция. Като цяло обаче той не решава проблемите на операцията за присвояване.
- **Пример:** Ако променим main по следния начин:

```
void main()
{
    product p, q;
    q = p;
}
```

отново възникват проблеми. Присвояването q = p; ще промени член-данните на q, но q вече има част в динамичната памет, която *първо трябва да бъде освободена*.

Инициализиране на обекти на класове …

- Налага се да бъде дефинирана нова операторна функция за присвояване. Последното ще направим по следния начин:

```
product& product::operator=(product const & p)
{
   cout << "assignment!\n";
   if(this != &p)
   {
      delete name;
      name = new char[strlen(p.name)+1];
      strcpy(name, p.name);
      price = p.price;
      ...
   }
   return *this;
}
```

Инициализиране на обекти на класове …

- Ще включим прототипа й

```
product& operator=(product const &);
```

в public-частта на класа.

- Забелязваме, че операторната функция за присвояване извършва аналогични действия на тези на конструктора за присвояване. Разликата е, че тя извършва тези действия върху съществуващ обект, а не върху току що създаден. Това налага предварително да бъде освободена динамичната памет, отделена за обекта.

Инициализиране на обекти на класове …

- Дефинираната операторна функция има за формален параметър константен псевдоним от клас product. По този начин се избягва създаването на нов обект и извикването на конструктора за присвояване. Въпреки, че не е задължително, използването на константен псевдоним е препоръчително.
- Освен, че променя обекта, указван от this, операторната функция от примера връща като резултат псевдонима му. Като следствие, конструкцията p = q може да се разглежда като израз (p = q връща p), а също да бъде лява страна на израз. Изразът p = q = r е допустим и е еквивалентен на q = r; p = q;
- В този пример реализирахме като член-функции на класа product конструктор за присвояване, операторна функция за присвояване и деструктор. Тези три функции се наричат “голямата тройка” или “канонична форма на класа”. На пратика те са задължителни при класове, използващи динамичната памет. Това не е просто добра идея, това е закон.

Масиви от обекти

- Създаването на масив от обекти става по два начина:
  - чрез дефиниция
  - чрез функциите за динамично управление на паметта.
- **При първия начин** масивът от обекти се създава при срещането на дефиницията (във функция или блок) и се разрушава при завършване на изпълнението на функцията или при излизане от блока. Дефиницията може да бъде поставена където и да е в тялото на функцията или блока, като пред и след нея може да има изпълними оператори.
- **Примери:** Ще използваме класа rat

а)

```
{    ...
     rat x[10];
}
```

Масиви от обекти …

- В този случай конструкторът rat() е извикан 10 пъти. Конструиран е масивът от обекти x:

```
x[0] x[1]  ...  x[9]
0/1   0/1   ...  0/1
```

- При завършване изпълнението на блока деструкторът ~rat() ще бъде извикан също 10 пъти за да разруши последователно x[9], x[8], ..., x[0].

б)

```
{
   rat x[10] = {rat(1,2), rat(5), 8, rat(1,7)};
}
```

- В този случай конструкторът rat(int = 0, int = 1) е извикан 10 пъти. Конструиран е масивът от обекти x:

```
x[0]x[1]   x[3]   x[4]   x[5]   x[6]   ...    x[9]
1/2 5/1    8/1    1/7    0/1    0/1    ...    0/1
```

Масиви от обекти …

- При завършване изпълнението на блока деструкторът ~rat() ще бъде извикан също 10 пъти за да разруши последователно x[9], x[8], ... x[0].
- **При втория начин**, създаването и разрушаването на масив от обекти се управлява от програмиста. Отново създаването става чрез new, а разрушаването – с delete, като операторите new се включват в конструкторите, а операторите delete – в деструкторът на класа, от който са обектите на масива.
- **Примери:**

а)

```
{
   rat *px = new rat[10];
   delete[] px;
}
```

Масиви от обекти …

В резултат, в хийпа се заделя блок от 80 байта (ако е възможно) и адресът на този блок се записва в px. Тъй като има дефиниран конструктор по подразбиране, конструкторът се извиква и px[i] (i=0, 1, ..., 9) се инициализират с рационалното число 0/1. При масивите, реализирани в динамичната памет, инициализация в явен вид не може да се зададе. Разрушаването на px става чрез delete[] px;. Преди да прекъсне връзката между px и динамичната памет, операторът delete[] извиква деструктора за всеки от обектите на масива.

б)

```
{
    rat *px = new rat[10];
    delete px;
}
```

Масиви от обекти …

- Операторът delete px; извиква деструктора само на обекта px[0] и прекъсва връзката на px с динамичната памет. Компилаторът съобщава за грешка. Причината е, че px е масив от обекти в динамичната памет, а деструкторът на класа rat е извикан само за px[0]. Ще отбележим отново, че ако px беше масив в динамичната памет, но не от обекти на клас, операторът delete px; щеше да работи нормално.

в)

```
{
    int size;
    cin >> size;
    rat* px = new rat[size];
    delete[size] px;
}
```

- В този случай деструкторът на класа rat се извиква size пъти. Реализацията на Visual C++ 6.0 пренебрегва size от delete, но за някои други реализации това не е така.