Обектно ориентирано програмиране

КЛАСОВЕ

ДЕСТРУКТОРИ. СЪЗДАВАНЕ И РАЗРУШАВАНЕ НА ОБЕКТИ НА КЛАСОВЕ

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() {
   static char s[40];
   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 …

Примери - стек с динамична памет:

- Разширяващ се стек
- Свързан стек

Свързан стек

Ще използваме следната структура:

```
struct StackElement {
    int data;
    StackElement* next;
};
```

Графично ще я представяме като двойна кутия:

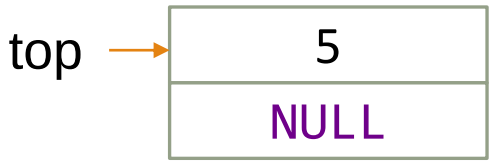
Свързан стек

Връх на стека:

```
StackElement* top;
```

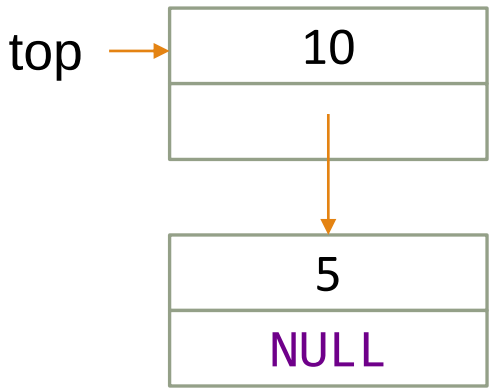
top се инициализира с NULL.

Вмъкваме 5 (push(5))

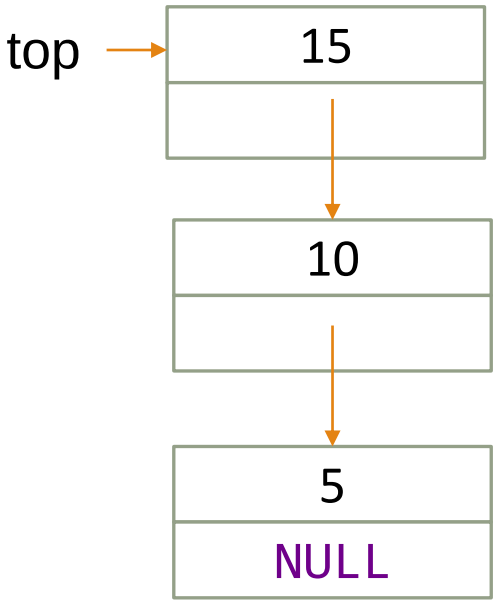


Свързан стек

Вмъкваме 10 (push(10))



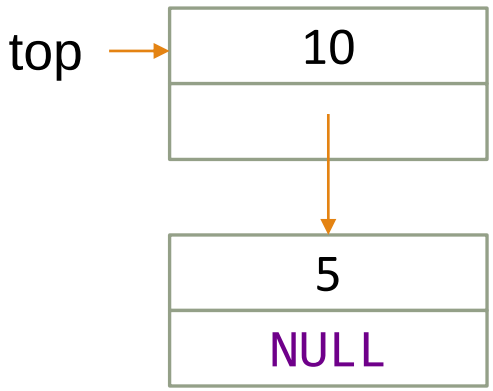
Вмъкваме 15 (push(15))



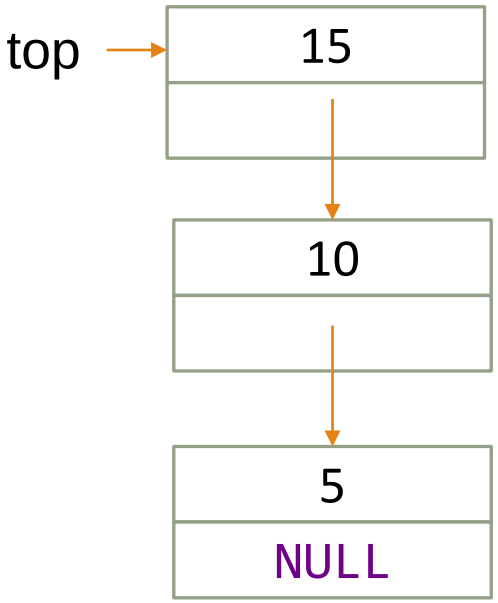

```
StackElement* p = new StackElement;
p->data = x;
p->next = top;
top = p;
```

Свързан стек

Вмъкваме 10 (push(10))



Вмъкваме 15 (push(15))




```
StackElement* p = new StackElement;
p->data = x;
p->next = top;
top = p;
```

Свързан стек

Махане на елемент от върха на стека:

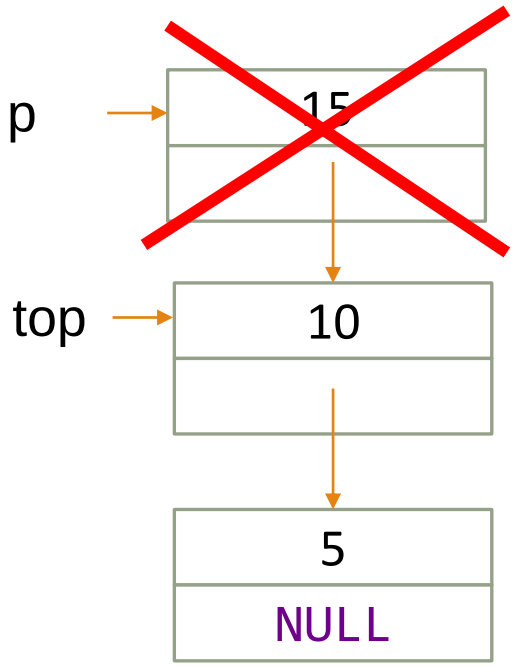
```
StackElement* p = top;

top = top->next;

int x = p->data;

delete p;

return x;
```



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 копиране.

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

```
Rational::~Rational()
{
    cout << "destruct number : "
         << numer << " / "
         << denom << endl;
}
```

Създаване и разрушаване на обекти на класове …

```
void main()
{
   Rational p(1, 8);    // създава обект p и го инициализира с 1/8
   Rational q = Rational(2, 9); // създава обект q и го инициализира с 2/9
   for (int i = 1; i <= 5; i++)
   {
      Rational 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 Rational е извикан толкова пъти, колкото пъти са извършвани дейности по създаване на обекти на класа Rational. Първите 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:

```
Rational *p = new Rational(3, 7);
// търси в хийпа 8B, свързва адреса
// им с p, извиква конструктора
// Rational(3,7) 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 Rational конструкторът е с прототип:

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

Инициализацията

```
Rational p = 7;
```

ще създаде обекта p и ще извика конструктора Rational(7), с който ще инициализира p. Ако в класа Rational 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 копиране, той се използва. В противен случай се използва подразбиращия се конструктор за копиране.

Инициализиране на обекти на класове …

Конструктор за копиране явно не е дефиниран

Тогава се извиква подразбиращия се системен конструктор за копиране.

Пример:

```
Rational p(1, 9);
Rational 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() {
   static char s[80];
   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 копиране, 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 копиране явно е дефиниран

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

<име_на_клас>(<име_на_клас> const&)

{<тяло>}

Ще дефинираме подходящ конструктор за копиране 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 копиране 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 & p);
```

в public-частта на класа.

Забелязваме, че операторната функция за присвояване извършва аналогични действия на тези на конструктора за копиране. 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 копиране. 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 копиране, 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 Rational

а)

```
{
   Rational 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 ~Rational() ще бъде извикан също 10 пъти за да разруши последователно x[9], x[8], ..., x[0].

б)

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

В този случай конструкторът Rational(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 ~Rational() 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)

```
{
    Rational *px = new Rational[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)

```
{
    Rational *px = new Rational[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;
    Rational* px = new Rational[size];
    delete[size] px;
}
```

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