ОБЕКТНО-ОРИЕНТИРАНО ПРОГРАМИРАНЕ

Магдалина Тодорова

спец. Информационни системи,
I курс, I поток

ФМИ, СУ „Св. Климент Охридски“
2018/2019

Тема № 9

Единично наследяване.

Конструктори, деструктор и операторна функция за присвояване на производен клас

1. Основни бележки

Обикновените конструктори, конструкторът за присвояване, операторната функция за присвояване и деструкторът са методи, за които не важат правилата за достъп при наследяване.

Тези методи на основния клас (с някой изключения) не се наследяват от производния клас.

2. Обикновени конструктори и деструктор

Конструктори

Конструкторите на производния клас инициализират само **собствените член-данни на класа**.

Наследените член-данни на производния клас се инициализират от **конструктор на основния му клас**.

Това се осъществява като в дефиницията на конст-руктора на производния клас се укаже обръщение към съответен конструктор на основния клас.

2. *Обикновени конструктори и деструктор*

Дефиниране на конструктор на производен клас (единично наследяване)

<дефиниция_на_конструктор_на_производен_клас> ::=
<име_на_производен_клас>::<име_на_производен_клас>
(<параметри>) **<инициализиращ_списък>**
{ <тяло>
}

<инициализиращ_списък> ::= <празно> |
: <име_на_основен_клас>(<параметри$_i$>)
{ , <член-данна>(<параметри$_i$>) }

2. *Обикновени конструктори и деструктор*

<параметри> ::= <празно> |

{ <параметър>, } <параметър>

<параметър> ::= <тип> <име_на_параметър>

<име_на_параметър> ::= <идентификатор>

<параметри$_i$> е конструкция, която има синтаксиса на <фактически_ параметри> от дефиницията на обръщение към функция

<тяло> е редица от оператори и дефиниции/декларации

2. *Обикновени конструктори и деструктор*

Забележки:

1) При единичното наследяване инициализиращият списък на конструктора на производния клас може да съдържа ***не повече*** от едно обръщение към конструктор на основен клас.

2) ***Ако инициализиращият списък не съдържа обръщение към конструктор на основния клас, чрез което да укаже как да се инициализира наследената част, в базовия клас трябва да е дефиниран конструктор по подразбиране.***

2. *Обикновени конструктори и деструктор*

Освен обръщение към конструктор на базовия клас, инициализиращият списък на конструктора на производния клас *може* да съдържа инициализация на собствени за производния клас член-данни.

Важно. Обръщението към конструктора на основния клас се записва в дефиницията на конструктора на производния клас, а не в неговата декларация в тялото на производния клас.

2. *Обикновени конструктори и деструктор*

<параметри$_i$> са изрази, съответстващи по брой, тип и смисъл на формалните параметри на съответния конструктор на базовия клас, т.е. обръщението

<име_на_основен_клас>(<параметри$_i$>)

трябва да е оформено съгласно дефиницията на съответен конструктор на основния клас.

Имената на параметрите от дефиницията на конструктора на производния клас могат да се използват за фактически параметри в обръщението към конструктор на основния клас.

Пример. Да разгледаме следните изкуствени класове

```
class base
{ public:
      base()                  // конструктор по подразбиране
      { a1 = 0;
        a2 = 0;
      }
      base(int x)             // конструктор с един параметър
      { a1 = x;
      }
      base(int x, int y)    // конструктор с 2 параметъра
      { a1 = x;
        a2 = y;
      }
```

```
    void a3() const
    { cout << "a1: " << a1 << endl
           << "a2: " << a2 << endl;
    }
    protected: int a2;
    private: int a1;
};
// дефиниция на производен на base клас der
class der : public base
{ public:
    der(int x, int y, int z, int t) : base(x, y)
    { d1 = z;
      d2 = t;
    }
```

```
    void d3() const
    { cout << "d1: " << d1 << endl
           << "d2: " << d2 << endl
           << "a2: " << a2 << endl;
      cout << "a3():" << endl;
      a3();
     }
  protected: int d2;
  private: int d1;
};
```

В резултат от изпълнението на фрагмента

```
der x(1, 2, 3, 4);
x.d3();
```

се получава:

```
d1: 3
d2: 4
a2: 2
a3():
a1: 1
a2: 2
```

2. *Обикновени конструктори и деструктор*

Ако конструкторът на класа *der* е дефиниран по следния начин

```
der(int z, int t) : base()
{ d1 = z;
  d2 = t;
}
```

наследените от базовия клас *base* компоненти ще се инициализират от подразбиращия се конструктор за класа *base*.

2. *Обикновени конструктори и деструктор*

В резултат от изпълнението на фрагмента:

```
der x(3, 4);
x.d3();
```

се получава

```
d1: 3
d2: 4
a2: 0
a3():
a1: 0
a2: 0
```

2. *Обикновени конструктори и деструктор*

Последната дефиниция на конструктора на класа *der* е еквивалентна на дефиницията

```
der(int z, int t)
{ d1 = z;
  d2 = t;
}
```

2. *Обикновени конструктори и деструктор*

Ако конструкторът на класа *der* има вида

```
der(int x, int y, int z, int t) : base(x)
{ d1 = z;
  d2 = t;
}
```

наследената компонента *a1* от базовия клас *base* ще се инициализира с *x*, а компонентата *a2* ще остане неинициализирана.

2. *Обикновени конструктори и деструктор*

Резултатът от изпълнението на фрагмента:

```
der x(1, 2, 3, 4);
x.d3();
```

e

```
d1: 3
d2: 4
a2: -858993460
a3():
a1: 1
a2: -858993460
```

2. *Обикновени конструктори и деструктор*

В инициализиращия списък може да участва ***не повече от едно обръщение към конструктор на един и същ базов клас***, т.е. дефиниция от вида

```
der(int x, int y, int z, int t) : base(), base(x, y)
{ d1 = z;
  d2 = t;
}
```

Грешка!!

е недопустима.

2. *Обикновени конструктори и деструктор*

Дефинирането на обект на производен клас е съпроводено от следните действия:

- ✓заделяне на памет за наследените и за собствените член-данни на обекта;
- ✓изпълнение на съответен конструктор на производния клас.

2. *Обикновени конструктори и деструктор*

Изпълнението на конструктор на производен клас преминава през следните стъпки:

- ✓замяна на формалните с фактическите параметри в обръщението към конструктор на основния клас;
- ✓ изпълнение на действията от инициализиращия списък;
- ✓изпълнение на операторите и дефинициите (декларациите) в тялото на конструктора на производния клас.

2. *Обикновени конструктори и деструктор*

Ако производният клас има собствени член-данни, които са обекти на класове и в инициализиращия списък на конструктора не е указано как те да се инициализират, техните конструктори по подразбиране се извикват след изпълнението на обръщението към конструктора на основния клас от инициализиращия списък и преди изпълнението на операторите в тялото на конструктора на производния клас.

Редът на изпълнението им съвпада с реда на член-данните обекти в производния клас.

2. *Обикновени конструктори и деструктор*

Пример. Член-данните *d1* и *d2* на *der* са обекти на класа *base*.

```
#include <iostream>
using namespace std;
class base
{ public:
      base()
      { cout << "constructor base() \n";
        a1 = a2 = 0;
      }
```

2. *Обикновени конструктори и деструктор*

```
base(int x, int y)
{ cout << "constructor base(" << x << ", " << y << ")\n";
   a1 = x;
   a2 = y;
}
```

2. *Обикновени конструктори и деструктор*

```
    void a3() const
    { cout << "a1: " << a1 << endl
           << "a2: " << a2 << endl;
    }
  protected: int a2;
  private: int a1;
};

class der : public base
{ public:
    der(int x, int y) : base(x, y)
    { cout << "constructor der\n";
    }
```

Извикване на конструктора по подразбиране на класа base за инициализиране на обектите d2 и d1

2. *Обикновени конструктори и деструктор*

```
    void d3() const
    { d1.a3();
      d2.a3();
      cout << "a2: " << a2 << endl;
      cout << "a3():" << endl;
      a3();
    }
  protected: base d2;
  private: base d1;
};
int main()
{ der x(1, 2); x.d3();
  return 0;
}
```

2. *Обикновени конструктори и деструктор*

Резултатът от изпълнението й е следният:

constrictor base(1, 2)
constrictor base()
constrictor base()
constrictor der
a1: 0
a2: 0
a1: 0
a2: 0
a2: 2
a3():
a1: 1
a2: 2

2. *Обикновени конструктори и деструктор*

Нека заменим дефиницията на конструктора на класа *der* със следната дефиниция

```
der(int x, int y) : base(x, y)
{ cout << "constructor der\n";
   d1 = base(15, 25);
   d2 = base(35, 45);
}
// …
der x(1, 2);
```

Резултат:

```
constructor base(1, 2)
constructor base()
constructor base()
constructor der
constructor base(15, 25)
constructor base(35, 45)
a1: 15
a2: 25
a1: 35
a2: 45
a2: 2
a3():
a1: 1
a2: 2
```

2. *Обикновени конструктори и деструктор*

Двукратното инициализиране на обектите *d1* и *d2* – член-данни на класа *der* може да се избегне като се използва следната дефиниция на конструктора на класа *der:*

```
der(int x, int y) : base(x, y),   1
                    d1(15, 25),   3
                    d2(35, 45)    2
{ cout << "constructor der\n";
}
```

***Резултат*:**

constructor base(1,2)
constructor base(35,45)
constructor base(15,25)
constructor der
a1: 15
a2: 25
a1: 35
a2: 45
a2: 2
a3():
a1: 1
a2: 2

2. *Обикновени конструктори и деструктор*

Ще разгледаме някои случаи:

• ***В основния клас не е дефиниран конструктор в т.ч. конструктор за присвояване***

В този случай в инициализиращия списък на конструктор(ите) на производния клас не трябва да се задава инициализация за наследените от основния клас член-данни.

Наследената част на производния клас остава неинициализи-рана.

2. *Обикновени конструктори и деструктор*

Пример.

```
#include <iostream>
using namespace std;
class base
{ public:
      void readbase(int x, int y)
      { a1 = x;
        a2 = y;
      }
      void a3() const
      { cout << "a1: " << a1 << endl
             << "a2: " << a2 << endl;
      }
  protected: int a2;
  private: int a1;
};
```

2. *Обикновени конструктори и деструктор*

```
class der : public base
{ public:
      der(int x, int y)
      { cout << "constructor der\n";
        d1 = x;
        d2 = y;
      }
      void d3() const
      { cout << "d1: " << d1 << endl;
        cout << "d2: " << d2 << endl;
        cout << "a2: " << a2 << endl;
        cout << "a3():" << endl;
        a3();
      }
  protected: int d2;
  private: int d1;
};
```

2. *Обикновени конструктори и деструктор*

```
int main()
{ der x(1, 2);
    x.d3();
    return 0;
}
```

Резултат от изпълнението:

```
constructor der
d1: 1
d2: 2
a2: -858993460
a3():
a1: -858993460
a2: -858993460
```

2. *Обикновени конструктори и деструктор*

• ***В основния клас е дефиниран само един конструктор с параметри, който не е подразбиращият се***

Възможни са:

а) ***в производния клас е дефиниран конструктор***

В този случай в инициализиращия списък на конструктора на производния клас задължително трябва да има обръщение към конструктора с параметри на основния клас. Изпълнява се по начина, описан по-горе.

2. *Обикновени конструктори и деструктор*

• ***В основния клас е дефиниран само един конструктор с параметри, който не е подразбиращият се***

б) ***в производния клас не е дефиниран конструктор***

В този случай компилаторът ще сигнализира за грешка. Необходимо е да се създаде конструктор на производния клас, който да извика конструктора на основния клас.

2. *Обикновени конструктори и деструктор*

Пример.

```
#include <iostream>
using namespace std;
class base
{ public:
      base(int x, int y)
      { a1 = x;
        a2 = y;
      }
      void a3() const
      { cout << "a1: " << a1 << endl
             << "a2: " << a2 << endl;
      }
  protected: int a2;
  private: int a1;
};
```

2. *Обикновени конструктори и деструктор*

```
class der : public base
{ public:
      // не е дефиниран конструктор
      void d3() const
      { cout << "d1: " << d1 << endl;
        cout << "d2: " << d2 << endl;
        cout << "a2: " << a2 << endl;
        cout << "a3():" << endl;
        a3();
      }
   protected: int d2;
   private: int d1;
};
```

2. *Обикновени конструктори и деструктор*

```
int main()
{ der x;
   x.d3();
   return 0;
}
```

издава следното съобщение за синтактична грешка

…cpp(33):error C2512: 'der': no appropriate default constructor available

2. *Обикновени конструктори и деструктор*

• ***В основния клас са дефинирани няколко конструктора в т.ч. подразбиращ се конструктор***

Възможни са:

а) ***в производния клас е дефиниран конструктор***

Тогава в инициализиращия списък на конструктора на производния клас може да се посочи, но може и да не се посочи конструктор на основния клас.

Ако не е посочен, компилаторът се обръща към конструктора по подразбиране на основния клас.

2. *Обикновени конструктори и деструктор*

• ***В основния клас са дефинирани няколко конструктора в т.ч. подразбиращ се конструктор***

б) ***в производния клас не е дефиниран конструктор***

В този случай компилаторът автоматично създава конструктор по подразбиране за производния клас. Последният активира и изпълнява конструктора по подразбиране на основния клас.

Собствените член-данни на производния клас остават неопределени.

2. *Обикновени конструктори и деструктор*

Пример.

```
#include <iostream>
using namespace std;
class base
{ public:
      base()
      { a1 = a2 = 0;
      }
      void a3() const
      { cout << "a1: " << a1 << endl
             << "a2: " << a2 << endl;
      }
   private: int a1;
   protected: int a2;
};
```

2. *Обикновени конструктори и деструктор*

```
class der : public base
{ public:
      void d3() const
      { cout << "d1: " << d1 << endl;
        cout << "d2: " << d2 << endl;
        cout << "a2: " << a2 << endl;
        cout << "a3():" << endl;
        a3();
      }
   private: int d1;
   protected: int d2;
};
```

2. *Обикновени конструктори и деструктор*

```
int main()
{ der x;
  x.d3();
  return 0;
}
```

се получава

```
d1: -858993460
d2: -858993460
a2: 0
a3():
a1: 0
a2: 0
```

2. *Обикновени конструктори и деструктор*

Деструктори

Деструкторът на производен клас трябва да разруши само онези **собствени** на производния клас член-данни, които са разположени в динамичната памет.

Деструкторите на производен клас и на неговия основен клас се изпълняват ***автоматично*** в ред, обратен на реда на изпълнението на техните конструктори. Най-напред се изпълнява деструкторът на производния клас, след това се изпълнява деструкторът на основния му клас.

2. *Обикновени конструктори и деструктор*

Пример.

```
#include <iostream>
using namespace std;
class A
{ public:
      A()
      { cout << "Конструктор на клас A\n";
      }
      ~A()
      { cout << "Деструктор на клас A\n";
      }
};
```

2. *Обикновени конструктори и деструктор*

```
class B : public A
{ public:
      B()
      { cout << "Конструктор на клас B\n";
      }
      ~B()
      { cout << "Деструктор на клас B\n";
      }
};
```

2. *Обикновени конструктори и деструктор*

```
class C : public B
{ public:
      C()
      { cout << "Конструктор на клас C\n";
      }
      ~C()
      { cout << "Деструктор на клас C\n";
      }
};
int main()
{ C x;
  return 0;
}
```

2. *Обикновени конструктори и деструктор*

Резултат от изпълнението:

Конструктор на клас A

Конструктор на клас B

Конструктор на клас C

Деструктор на клас C

Деструктор на клас B

Деструктор на клас A

2. *Обикновени конструктори и деструктор*

Задача.

Да се дефинират отново класовете *People*, *Student* и *PStudent*, така че инициализиращите действия да се изпълняват от подходящи конструктори, а разрушителните – от деструктори.

2. *Обикновени конструктори и деструктор*

```
#include <iostream>
#include <cassert>
#include <cstring>
using namespace std;

class People
{ public:
      People(const char* = "", const  char* = "");
      ~People();
      void PrintPeople() const;
   private:
      char* name;
      char* ucn;
};
```

2. *Обикновени конструктори и деструктор*

```
People::People(const char* na, const char* uc)
{ name = new char[strlen(na)+1];
  assert(name != NULL);
  strcpy(name, na);  // strcpy_s(name, strlen(na) + 1, na);
  ucn = new char[strlen(uc)+1];
  assert(ucn != NULL);
  strcpy(ucn, uc);     // strcpy_s(ucn, strlen(uc) + 1, uc);
}

People::~People()
{ cout << "~People()\n";
  delete [] name;
  delete [] ucn;
}
```

2. *Обикновени конструктори и деструктор*

```
void People::PrintPeople() const
{ cout << "Name: " << name << endl;
  cout << "UCN: " << ucn << endl;
}
```

2. *Обикновени конструктори и деструктор*

```
// декларация на класа Student
class Student : public People
{ public:
      Student(const  char* = "", const char* = "",
                                                long = 0, double = 0);
     ~Student()                                 // излишен е
     { cout << "~Student()\n";
     }
     void PrintStudent() const;
  private:
     long fac_numb;
     double gpa;
};
```

2. *Обикновени конструктори и деструктор*

```
Student::Student(const char* na, const char* uc,
                 long f_nu, double gp) : People(na, uc)
{ fac_numb = f_nu;
  gpa = gp;
}
```

2. *Обикновени конструктори и деструктор*

```
// дефиниция на метода PrintStudent
void Student::PrintStudent() const
{  PrintPeople();
   cout << "Fac. number: " << fac_numb << endl;
   cout << "GPA of student: " << gpa << endl;
}
```

2. *Обикновени конструктори и деструктор*

```
// декларация на класа PStudent

class PStudent : public Student
{ public:
      PStudent(const char* = "", const char* = "", long = 0,
                        double = 0, double = 0);
      ~PStudent()                                  // излишен е
      { cout << "~PStudent()\n";
      }
      void PrintPStudent() const;
   private:
      double fee;
};
```

2. *Обикновени конструктори и деструктор*

```
PStudent::PStudent(const char* na, const char* uc, long f_nu,
                   double gp, double fe) : Student(na, uc, f_nu, gp)
{ fee = fe;
}

void PStudent::PrintPStudent() const
{ PrintStudent();
  cout << "Fee: " << fee << endl;
}
```

2. *Обикновени конструктори и деструктор*

```
int main()
{ PStudent PStud("Ivan Ivanov", "9206120000",
                          48444, 5.0, 400);
  PStud.PrintPStudent();
  return 0;
}
```

2. *Обикновени конструктори и деструктор*

Резултат от изпълнението:

Name: Ivan Ivanov
UCN: 9206120000
Fac. number: 48444
GPA of student: 5
Fee: 400
~PStudent()
~Student()
~People()

3. Конструктор за присвояване и операторна функция за присвояване

В общия случай, производният клас **не** наследява от основния си клас конструктора за присвояване и оператора за присвояване.

Изключения:

Конструктор за присвояване

Конструкторът за присвояване на производния клас инициализира собствените член-данни на класа, а конструкторът за присвояване (или друг конструктор) на основния клас инициализира наследените член-данни.

Конструктор за присвояване

Конструкторът за присвояване на производен клас се дефинира по аналогичен начин като обикновените конструктори на производни класове.
Инициализаторът се задава чрез единствения явно указан формален параметър const <име_на_клас>&.

<име_на_клас>::<име_на_клас>(const <име_на_клас>&)
<инициализиращ_списък>
{ <тяло>
}

Конструктор за присвояване

Ако в клас не е дефиниран конструктор за присвояване, ролята на такъв се поема от генерирания системен *конструктор за копиране* с прототип от вида

<име_на_клас>(const <име_на_клас>&);

Конструктор за присвояване

- ***В производния клас НЕ е дефиниран конструктор за присвояване***

Възможни са:

Конструктор за присвояване

а) ***в основния клас е дефиниран конструктор за присвояване***

В този случай компилаторът генерира конструктор за копиране на производния клас, който преди да се изпълни**, активира и изпълнява конструктора за присвояване на основния клас.**

В случая се казва, че конструкторът за присвояване на основния клас се наследява от производния клас.

Конструктор за присвояване

Задача.

Да се допълни класът *People* от предната задача с конструктор за присвояване. Да се създадат и изведат два обекта на класа *Student.* Единият обект да е създаден чрез параметричния конструктор, а другият – чрез конструктора за копиране на класа *Student*.

Конструктор за присвояване

```
#include <iostream>
#include <cstring>
#include <cassert>

using namespace std;
```

Конструктор за присвояване

```
// декларация на класа People

class People
{ public:
      People(const char* = "", const char* = "");
      ~People();
      People(const People&);
      void PrintPeople() const;
   private:
      char* name;
      char* ucn;
};
```

Конструктор за присвояване

```
// дефиниция на конструктора
// за присвояване на класа People

People::People(const People& p)
{ cout << "People(const People&)\n";
   name = new char[strlen(p.name)+1];
   assert(name != NULL);
   strcpy(name, p.name);  // strcpy_s(name, strlen(p.name) + 1, p.name);
   ucn = new char[strlen(p.ucn)+1];
   assert(ucn != NULL);
   strcpy(ucn, p.ucn);        // strcpy_s(ucn, strlen(p.ucn) + 1, p.ucn);
}
```

Конструктор за присвояване

```
// дефиниция на деструктора

People::~People()
{ delete [] name;
  delete [] ucn;
}
```

Конструктор за присвояване

```
class Student : public People
{ public:
    Student(const char* = "", const char* = "", long = 0, double = 0);
    ~Student()
    { cout << "~Student()\n";
    }
    void PrintStudent() const;
  private:
    long fac_numb;
    double gpa;
};
```

Конструктор за присвояване

```
class PStudent : public Student
{ public:
      PStudent(const char* = "", const char* = "", long = 0,
                        double = 0, double = 0);
      ~PStudent()
      { cout << "~PStudent()\n";
      }
      void PrintPStudent() const;
   private:
      double fee;
};
```

Конструктор за присвояване

```
int main()
{ Student s1("Ivan Ivanov ", "9206120000", 48444, 6.0);
  s1.PrintStudent();
  Student s2 = s1;
  s2.PrintStudent();
  return 0;
}
```

```
Name: Ivan Ivanov
UCN: 9206120000
Fac. number: 48444
GPA of student: 6
People(const People&)
Name: Ivan Ivanov
UCN: 9206120000
Fac. number: 48444
GPA of student: 6
~Student()
~People()
~Student()
~People()
```

Конструктор за присвояване

б) ***в основния клас не е дефиниран конструктор за присвояване***

В този случай в основния и в производния му клас се генерират конструктори за копиране.

Конструкторът за копиране на производния клас активира конструктора за копиране на основния клас.

Конструктор за присвояване

```
int main()
{ PStudent s1("Ivan Ivanov", "9206120000", 48444, 5.0, 400);
  s1.PrintPStudent();
  PStudent s2 = s1;
  s2.PrintPStudent();
  return 0;
}
```

```
Name: Ivan Ivanov
UCN: 9206120000
Fac. number: 48444
GPA of student: 5
Fee: 400
People(const People&)
Name: Ivan Ivanov
UCN: 9206120000
Fac. number: 48444
GPA of student: 5
Fee: 400
~PStudent()
~Student()
~People()
~PStudent()
~Student()
~People()
```

- ***В производния клас Е дефиниран конструктор за присвояване***

Дефиницията на конструктора за присвояване на производния клас определя как точно ще се инициализира наследената част.

В неговия инициализиращ списък може да има или да няма обръщение към конструктор (за присвояване или обикновен) на основния му клас.

Препоръчва се в инициализиращия списък на производния клас да има обръщение към конструктора за присвояване на основния клас, ако такъв е дефиниран.

***Забележка*:**

Ако не е указано обръщение към конструктор на основния клас, инициализирането на наследените членове се осъществява от подразбиращия се конструктор на основния клас.

***Ако основният клас няма подразбиращ се конструктор, се съобщава за отсъствието на подходящ конструктор*.**

Конструктор за присвояване

Задача.

Да се допълни класът *Student* от предната задача с конструктор за присвояване.

В случая това не е необходимо, защото генерираният от компилатора конструктор за копиране е напълно достатъчен.
Добавянето на конструктора за присвояване в *Student* е заради експериментални цели.

Конструктор за присвояване

```
class Student : public People
{ public:
    Student(const char* = "", const char* = "", long = 0, double = 0);
    ~Student()
    { cout << "~Student()\n";
    }
    Student(const Student&);
    void PrintStudent() const;
  private:
    long fac_numb;
    double gpa;
};
```

Конструктор за присвояване

```
// дефиниция на конструктора
// за присвояване на класа Student

Student::Student(const Student& st) : People(st)
{ cout << "Student(const Student&)\n";
  fac_numb = st.fac_numb;
  gpa = st.gpa;
}
```

неявно преобразуване на **st** в тип const People&

Конструктор за присвояване

```
int main()
{ Student s1("Ivan Ivanov", "9206120000", 48444, 5.0);
  s1.PrintStudent();
  Student s2 = s1;
  s2.PrintStudent();
  return 0;
}
```

```
Name: Ivan Ivanov
UCN: 9206120000
Fac. number: 48444
GPA of student: 5
People(const People&)
Student(const Student&)
Name: Ivan Ivanov
UCN: 9206120000
Fac. number: 48444
GPA of student: 5
~Student()
~People()
~Student()
~People()
```

Операторна функция за присвояване

Операторната функция за присвояване на производен клас трябва да указва как да се осъществи присвояването както на собствените, така и на наследените си член-данни.

За разлика от конструкторите на производния клас тя прави това в тялото си, т.е. **не поддържа инициализиращ списък**.

```
<производен_клас>& <производен_клас>::operator=
                              (const <производен_клас>& p)
{ if (this != &p)
  {   // дефиниране на присвояването
      // за наследените член-данни
        <основен_клас>::operator=(p);
      // дефиниране на присвояването
      // за собствените член-данни
      Del();      // разрушаване на онези собствени
                  // член-данни на подразбиращия
                  // се обект, които са разположени в ДП
      Copy(p);    // копиране на собствените член-данни
                  // на p в съответните член-данни на
                  // подразбиращия се обект
  }
  return *this;
}
```

неявно преобразуване на **p** в тип *const <основен_клас>&*

Някои случаи:

- ***В производния клас НЕ е дефинирана операторна функция за присвояване***

Компилаторът създава операторна функция за присвояване на производния клас. Тя се обръща и изпълнява операторната функция за присвояване на основния клас (дефинираната или подразбиращата се), чрез която инициализира наследената част, след това инициализира чрез присвояване и собствените член-данни на производния клас.

Затова в този случай се казва, че операторът за присвояване на основния клас се наследява.

• ***В производния клас E дефинирана операторна функция за присвояване***

Тази член-функция ***трябва да се погрижи за присвояването на наследените компоненти. В тялото на нейната дефиниция трябва да има обръщение към дефинирания оператор за присвояване на основния клас***, ако има такъв.

Ако това не е направено явно, стандартът на езика не уточнява как ще стане присвояването на наследените компоненти.

В случая ***операторът за присвояване на основния клас не се наследява***.

Пример. В следващата програма е дефиниран базов клас *base*, който има три производни класа: *der1*, *der2* и *der3*. Показани са различни случаи за дефиниране на оператора за присвояване в производните класове *der1* и *der2*, а в класа *der3* не е дефиниран оператор за присвояване.

```
#include <iostream>
using namespace std;
class base
{ public:
      base(int x = 0)
      { b = x;
      }
```

Операторна функция за присвояване

```
    base& operator=(const base &x)
    { if (this != &x) b = x.b + 1;
      return *this;
    }
  protected:
    int b;
};

class der1 : public base
{ public:
    der1(int x = 1)
    { d = x;
    }
```

Операторна функция за присвояване

```
    der1& operator=(const der1& x)
    { if  (this != &x)
       { b = x.b + 3;
         d = x.d + 2;
       }
       return *this;
    }
    void print() const
    { cout << "der " << d << " base " << b << endl;
    }
  private:
    int d;
};
```

```
class der2 : public base
{ public:
      der2(int x = 2)
      { d = x;
      }
      der2& operator=(const der2& x)
      { if (this != &x)
           d = x.d + 3;
        return *this;
      }
      void print() const
      { cout << "der " << d << " base " << b << endl;
      }
   private:
      int d;
};
```

Операторна функция за присвояване

```
class der3 : public base
{ public:
      der3(int x = 3)
      { d = x;
      }
      void print() const
      { cout << "der " << d << " base " << b << endl;
      }
   private:
      int d;
};
```

Операторна функция за присвояване

```
int main()
{ der1 d11(5), d12;
  der2 d21(5), d22;
  der3 d31(5), d32;
  d12 = d11;
  d22 = d21;
  d32 = d31;
  cout << "d11: "; d11.print();
  cout << "d12: "; d12.print();
  cout << "d21: "; d21.print();
  cout << "d22: "; d22.print();
  cout << "d31: "; d31.print();
  cout << "d32: "; d32.print();
  return 0;
}
```

Операторна функция за присвояване

```
int main()
{ der1 d11(5), d12;
  der2 d21(5), d22;
  der3 d31(5), d32;
  d12 = d11;
  d22 = d21;
  d32 = d31;
  cout << "d11: "; d11.print();
  cout << "d12: "; d12.print();
  cout << "d21: "; d21.print();
  cout << "d22: "; d22.print();
  cout << "d31: "; d31.print();
  cout << "d32: "; d32.print();
  return 0;
}
```

***Резултат*:**

d11: der 5 base 0
d12: der 7 base 3
d21: der 5 **base 0**
d22: der 8 **base 0**
d31: der 5 base 0
d32: der 5 base 1

Задача.

Да се дефинират отново класовете *People*, *Student* и *PStudent* като в класа *Student* член-данната *fac_numb* да е от тип *char** и да се добави член-данна *addr* от тип *char*,* определяща адреса на студент. Двете член-данни да се реализират в динамичната памет. В класа *PStudent* да се добави член-данна *workplace* от тип *char**, определяща месторабoтата на студент от платена форма на обучение и да се реализира в динамичната памет. За всеки от класовете да се дефинира каноничното представяне.

Всеки клас на програмата съдържа две помощни член-функции – за копиране и за изтриване (разрушаване) на собствените им компоненти.

Канонично представяне

```
#include <iostream>
#include <cstring>
#include <cassert>
using namespace std;
class People
{ public:
    // канонично представяне
    People(const char* = "", const char* = "");
    ~People();
    People(const People&);
    People& operator=(const People& p);
    // член-функция за извеждане
    void PrintPeople() const;
```

Канонично представяне

```
  private:
     char* name;                  // име
     char* ucn;                   // ЕГН
     // помощни член-функции
     // за копиране и изтриване
     void copy_People(const char*, const char*);
     void del_People();
};
```

Канонично представяне

```
void People::copy_People(const char* na, const char* uc)
{ name = new char[strlen(na)+1];
  assert(name != NULL);
  strcpy(name, na);   // strcpy_s(name, strlen(na) + 1, na);
  ucn = new char[strlen(uc)+1];
  assert(ucn != NULL);
  strcpy(ucn, uc);      // strcpy_s(ucn, strlen(uc) + 1, uc);
}

void People::del_People()
{ delete [] name;
  delete [] ucn;
}
```

Канонично представяне

```
People::People(const char* na, const char* uc)
{ copy_People(na, uc);
}

People::~People()
{ del_People();
}

People::People(const People& p)
{ copy_People(p.name, p.ucn);
}
```

Канонично представяне

```
People& People::operator=(const People& p)
{ if  (this!=&p)
    { del_People();
      copy_People(p.name, p.ucn);
    }
  return *this;
}

void People::PrintPeople() const
{ cout << "Name: " << name << endl;
  cout << "UCN: " << ucn << endl;
}
```

Канонично представяне

```
class Student : public People
{ public:
    // канонично представяне
    Student(const char* = "", const char* = "", double = 0,
                const char* = "", const char* = "");
    ~Student();
    Student(const Student&);
    Student& operator=(const Student&);
    // член-функция за извеждане
    void PrintStudent() const;
```

Канонично представяне

```
    private:
        double gpa;          // среден успех
        char* fac_numb;  // факултетен номер
        char* addr;          // адрес

        // помощни член-функции
        // за копиране и изтриване
        void copy_Student(double, const char*, const char*);
        void del_Student();
};
```

```
void Student::copy_Student(double gp, const char* f_nu,
                                      const char* add)
{ gpa = gp;
  fac_numb = new char[strlen(f_nu)+1];
  assert(fac_numb != NULL);
  strcpy(fac_numb, f_nu);    // strcpy_s(fac_numb, strlen(f_nu) + 1, f_nu);
  addr = new char[strlen(add)+1];
  assert(addr != NULL);
  strcpy(addr, add);         // strcpy_s(addr, strlen(add) + 1, add);
}
void Student::del_Student()
{ delete [] fac_numb;
  delete [] addr;
}
```

Канонично представяне

```
Student::Student(const char* na, const char* uc, double gp,
                 const char* f_nu, const  char* add) : People(na, uc)
{ copy_Student(gp, f_nu, add);
}

Student::~Student()
{ del_Student();
}

Student::Student(const Student& st) : People(st)
{ copy_Student(st.gpa, st.fac_numb, st.addr);
}
```

Канонично представяне

```
Student& Student::operator=(const Student& st)
{ if  (this != &st)
    { People::operator=(st);
      del_Student();
      copy_Student(st.gpa, st.fac_numb, st.addr);
    }
  return *this;
}
void Student::PrintStudent() const
{ PrintPeople();
  cout << "GPA of student: " << gpa << endl;
  cout << "Fac. nomer: " << fac_numb << endl;
  cout << "Address: " << addr << endl;
}
```

Канонично представяне

```
class PStudent : public Student
{ public:
    // канонично представяне
    PStudent(const char* = "", const char* = "", double = 0,
                const char* = "", const char* = "", double = 0,
                const  char* = "");
    ~PStudent();
    PStudent(const PStudent&);
    PStudent& operator=(const PStudent&);
    // член-функция за извеждане
    void PrintPStudent() const;
```

```
  private:
     double fee;                  // такса
     char* workplace;             // месторабота
     // помощни член-функции
     // за копиране и изтриване
     void copy_PStudent(double, const char*);
     void del_PStudent();
};

void PStudent::copy_PStudent(double fe, const char* wpl)
{ fee = fe;
  workplace = new char[strlen(wpl)+1];
  assert(workplace!= NULL);
  strcpy(workplace, wpl); // strcpy_s(workplace, strlen(wpl) + 1, wpl);
}
```

Канонично представяне

```
void PStudent::del_PStudent()
{ delete [] workplace;
}

PStudent::PStudent(const char* na, const char* uc, double gp,
   const char* f_nu, const char* add, double fe, const char* wpl)
                          : Student(na, uc, gp, f_nu, add)
{ copy_PStudent(fe, wpl);
}

PStudent::~PStudent()
{ del_PStudent();
}
```

Канонично представяне

```
PStudent::PStudent(const PStudent& ps) : Student(ps)
{ copy_PStudent(ps.fee, ps.workplace);
}

PStudent& PStudent::operator=(const PStudent& ps)
{ if (this != &ps)
   { Student::operator=(ps);
     del_PStudent();
     copy_PStudent(ps.fee, ps.workplace);
   }
  return *this;
}
```

Канонично представяне

```
void PStudent::PrintPStudent() const
{ PrintStudent();
  cout << "Fee: " << fee << endl
       << " Workplace:" << workplace<< endl;
}
```

В главната функция:

```
  PStudent s1("Ivanov", "8811226666", 5.5, "777888999",
                "Balgaria 45", 450, "SU-FMI-CS");
  s1.PrintPStudent();
  PStudent s2("Petrova", "9003156677", 4.5, "222333444",
                "J. Boucher 34", 520, "SU-FMI-SE"), s3;
  s2.PrintPStudent();
  s3 = s1;
  s3.PrintPStudent();
```