Обектно ориентирано програмиране

НАСЛЕДЯВАНЕ. ПРОИЗВОДНИ КЛАСОВЕ (ЧАСТ 2)

Предефиниране на компоненти

Базовият и производният клас могат да притежават собствени компоненти с еднакви имена. В този случай производният клас ще притежава компоненти с еднакви имена. Обръщението към такава компонента чрез обект от производния клас извиква декларираната в класа компонента, т.е. *името на собствената компонента е с по-висок приоритет от това на наследената*. За да се изпълни "покритата" наследена компонента се указва пълното й име, т.е.

<име_на_клас>::<компонента>

където <име_на_клас> е името на основния клас.

Предефиниране на компоненти …

```
#include <iostream>
using namespace std;
class Base
{
public:
  void init(int x) {
     bx = x;
   }
   void display() const {
      cout << " class Base: bx= " << bx << endl;
   }
protected:
   int bx;
private:
// …
};
```

Предефиниране на компоненти ...

```
class Der : public Base
{
public:
   void init(int x) {
      bx = x;
      Base::bx = x + 5;
   }
   void display() const {
      cout << " class Der: bx = " << bx;
      cout << " Base::bx = " << Base::bx << endl;
   }
```

Предефиниране на компоненти …

```
protected:
   int bx;
private:
//...
};
void main() {
   Base b;
   Der d;
   b.init(5); d.init(10);
   b.display(); d.display();
   d.Base::init(20);
   d.Base::display();
   d.display();
   b.display();
}
```

Предефиниране на компоненти …

Изход:

```
class Base: bx = 5
class Der: bx = 10 Base::bx = 15
class Base: bx = 20
class Der: bx = 10 Base::bx = 20
class Base: bx = 5
```

Производни класове ...

Конструктори, операторни функции за присвояване и деструктори на производни класове

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

Тези методи (с някои изключения) не се наследяват от производния клас. Ако например конструктор можеше да бъде наследен, той щеше да инициализира само наследената част. Нормално е конструкторът на производен клас да инициализира както наследената, така и собствената част на класа. Същото се отнася и за деструктора. Това е причината, заради която конструкторите и деструкторът на основния клас не се наследяват от производния клас.

Производни класове ...

Възможно е обаче конструкторът на производния клас да активира конструктор на основния клас, който пък да инициализира наследената част. Производният клас не наследява и създадените от програмиста конструктор за копиране и предефинирания оператор за присвояване на обекти =.

Следователно, голямата четворка на основния клас не се наследява от производния клас. Това е следствие на особената роля на четворката.

Дефинирането и използването на голямата четворка за производния клас ще разгледаме на няколко стъпки. За да разграничим конструктора за копиране от останалите конструктори, последните ще наричаме обикновени или само конструктори.

Обикновени конструктори и деструктор

Конструктори

Обикновените конструктори на базовия и на производния клас изпълняват инициализиращи функции. Принципен е въпросът *как и от кого да се реализира инициализирането на наследената част на производния клас.* Най-естествено е това да се направи от конструкторите на производния клас. Но ако това е така, конструкторите на производния клас трябва да имат достъп до наследените, при това най-често, private компоненти на основния клас. Това е в противоречие на принципа за капсулиране на информацията. Затова инициализирането на собствените и наследените членове е разделено между конструкторите на производния и основния клас.

Обикновени конструктори и деструктор …

Конструкторите на производния клас инициализират само собствените член-данни на класа. Наследените член-данни на производния клас се инициализират от конструктор на основния клас. Това се осъществява като в дефиницията на конструктора на производния клас се укаже обръщение към съответен конструктор на основния клас

Дефиниция на конструктор на производен клас

Синтаксис

<дефиниция_на_конструктор_на_производен_клас> ::=

<име_на_производен_клас>::<име_на_производен_клас>(<параметри>)

<инициализиращ_списък>

{

<тяло>

}

Обикновени конструктори и деструктор ...

<инициализиращ_списък> ::= <празно> |

: <име_на_основен_клас>(<параметри$_i$>)

{,<име_на_основен_клас>(<параметри$_i$>)}

<параметри> ::= <празно> | <параметър> |

<параметри>, <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, а <тяло> се определя като тялото на който и да е конструктор.

Обикновени конструктори и деструктор ...

При единично наследяване в инициализиращия списък на производния клас е указано не повече от едно обръщение към конструктор на основен клас. При множествено наследяване в инициализиращия списък може да са указани няколко обръщения към конструктори на основни класове. За разделител се използва символът запетая.

Забележка. Обръщенията към конструкторите на основни класове се обявяват в дефиницията на конструктора на производния клас, а не в неговата декларация в тялото на производния клас.

Ще отбележим също, че <параметри$_i$> са изрази или идентификатори, 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 се оформи според дефиницията на конструктора на основния клас. Имената на параметри от конструктора на производния клас могат да се използват за фактически параметри в обръщението към конструктора на основния клас.

Обикновени конструктори и деструктор …

```
#include <iostream>
using namespace std;
class Base
{
private:
   int a1;
protected:
   int a2;
public:
   Base() { // конструктор по подразбиране
      a1 = 0;
      a2 = 0;
   }
   Base(int x) { // конструктор с един параметър
      a1 = x;
      a2 = 0;
   }
```

Обикновени конструктори и деструктор …

```
    Base(int x, int y)// конструктор с два параметъра
    {
        a1 = x;
        a2 = y;
    }
    void a3()
    {
        cout << "a1: " << a1 << endl
             << "a2: " << a2 << endl;
    }
};
```

Обикновени конструктори и деструктор …

```
// дефиниция на производния клас Der
class Der : public Base
{
private:
   int d1;
protected:
   int d2;
public:
   // конструктор
   Der(int x, int y, int z, int t) : Base(x, y)
   {
      d1 = z;
      d2 = t;
   }
```

Обикновени конструктори и деструктор …

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

Тъй като в инициализиращия списък на конструктора на класа Der участва двуаргументният конструктор на Base, наследените компоненти се инициализират от него.

Обикновени конструктори и деструктор ...

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
```

Обикновени конструктори и деструктор ...

Ако вместо двуаргументния конструктор на основния клас дефиницията на конструктора на производния клас Der използва подразбиращият се конструктор на Base, т.е.

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

наследените компоненти от базовия клас Base ще се инициализират от подразбиращия се за Base конструктор. В резултат от изпълнението на фрагмента:

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

Обикновени конструктори и деструктор …

се получава:

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

Тази дефиниция на конструктора на der е еквивалентна на дефиницията:

```
Der(int x, int y, int z, int t)
{
   d1 = z;
   d2 = t;
}
```

т.е. подразбиращият се конструктор на основния клас може да бъде пропуснат в инициализиращия списък на конструктора на производния клас.

Обикновени конструктори и деструктор ...

Ако вместо двуаргументния конструктор на основния клас дефиницията на конструктора на производния клас der използва едноаргументния конструктор на Base, т.е.

```
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 се инициализира с 0 и в резултат от изпълнението на фрагмента:

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

Обикновени конструктори и деструктор ...

ще се получи:

```
d1: 3
d2: 4
a2: 0
a3():
a1: 1
a2: 0
```

Към примера ще отбележим изрично, че в инициализиращия списък може да участва не повече от едно обръщение към конструктор на един и същ базов клас

Обикновени конструктори и деструктор ...

Семантика

Дефинирането на обект от производен клас предизвиква създаване на “неявен” обект от базовия му клас и добавяне на декларираните в производния клас компоненти. Това означава, че ако и базовият, и производния клас имат конструктори, то първо се извиква конструкторът на базовия, а след това – конструкторът на производния клас.

В случая на множествено наследяване, извикването на конструктор на производен клас води до извикването на указаните в дефиницията му конструктори на неговите основни класове и след завършване на тяхното изпълнение се изпълнява <тяло> на конструктора на производния клас.

Обикновени конструктори и деструктор ...

Процедурата е следната:

- заменят се формалните с фактическите параметри във всяко обръщение към конструктор на основен клас, след което се изпълнява обръщението;
- изпълняват се операторите в тялото на конструктора на производния клас.

Ако производният клас има член-данни, които са обекти, техните конструктори се извикват след изпълнението на обръщенията към конструкторите на основните класове от инициализиращия списък и преди изпълнението на операторите в тялото на конструктора на производния клас. Конструкторите на обектите се извикват по реда на тяхното деклариране в тялото на производния клас.

Обикновени конструктори и деструктор …

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

Обикновени конструктори и деструктор ...

```
    Base(int x, int y) {
        cout << "constructor Base(" << x << ","
             << y << ")\n";
        a1 = x;
        a2 = y;
    }
    void a3() {
        cout << "a1: " << a1 << endl
             << "a2: " << a2 << endl;
    }
};
```

Обикновени конструктори и деструктор …

```
class Der : public Base
{
private:
   Base d1;
protected:
   Base d2;
public:
   Der(int x, int y) : Base(x, y)
   {
      cout << "constructor Der\n";
   }
```

Обикновени конструктори и деструктор …

```
    void d3() {
        d1.a3();
        d2.a3();
        cout << "a2: " << a2 << endl;
        cout << "a3():" << endl;
        a3();
     }
};
void main() {
    Der x(1, 2);
    x.d3();
}
```

Обикновени конструктори и деструктор …

```
Изход:
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
```

Обикновени конструктори и деструктор ...

Ще се отбележим специално на някои случаи:

В основния клас не е дефиниран конструктор

В този случай в инициализиращият списък не се отбелязва нищо. Наследената част на производния клас остава неинициализирана.

Основният клас има само един конструктор с параметри, който не е подразбиращия се

Възможни са:

а) в производния клас е дефиниран конструктор

Тогава трябва да има задължително обръщение към него в инициализиращия списък.

Обикновени конструктори и деструктор ...

б) в производния клас не е дефиниран конструктор

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

Основният клас има няколко конструктора в т. число подразбиращ се конструктор

Възможни са:

а) в производния клас е дефиниран конструктор

Тогава може да не се посочва конструктор за основния клас в инициализиращия списък. Ако не е посочен, компилаторът се обръща към подразбиращия се конструктор на основния клас.

Обикновени конструктори и деструктор ...

б) в производния клас не е дефиниран конструктор

В този случай компилаторът автоматично създава подразбиращ се конструктор за производния клас. Последният активира и изпълнява подразбиращият се конструктор на основния клас. Собствените членове на производния клас са инициализирани неопределено.

Обикновени конструктори и деструктор ...

Деструктори

Деструкторите на един производен клас и на неговите основни класове се изпълняват в ред, обратен на реда на изпълнение на техните конструктори. Най-напред се изпълнява деструкторът на производния клас, след това се изпълняват деструкторите на неговите основни класове.

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

Обикновени конструктори и деструктор …

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

Обикновени конструктори и деструктор …

```
Конструктор на класа A
Конструктор на класа B
Конструктор на класа C
Деструктор на клас C
Деструктор на клас B
Деструктор на клас A
```

При създаването на обекта x се извиква конструкторът на класа C. Тъй като C е производен на класа B и инициализиращият му списък е празен, се извиква конструкторът на класа B, който започва да създава “неявен” обект от клас B. Но класът B е производен на класа A и инициализиращият му списък е празен. Това предизвиква обръщение към подразбиращия се конструктор на класа A. Заради това отначало се изпълнява конструкторът на класа A, след това се довършва създаването на обекта от клас B като се извиква конструкторът му. Най-накрая се извиква конструкторът на класа C за да завърши създаването на обекта x.

Обикновени конструктори и деструктор ...

При завършване изпълнението на тялото на функцията main започва процес на разрушаване на обекта x. Това предизвиква обръщение към деструктора на класа C, след това – към деструктора на класа B, след него – към деструктора на класа A и най-накрая обектът x се разрушава.

Задача. Да се дефинират повторно класовете Person, Student и PStudent от предишната задача, така че инициализиращите действия да се изпълняват от подходящи конструктори. Разрушителните действия да се извършват от деструктори.

Обикновени конструктори и деструктор …

```
#include <iostream>
#include <string>
using namespace std;
// декларация на класа Person
class Person
{
public:
   Person(char * = "", char * = "");
   void printPerson() const;
   ~Person();
private:
   char * name;
   char * egn;
};
```

Обикновени конструктори и деструктор ...

```
// дефиниция на конструктора на Person
Person::Person(char *str, char *num)
{
    name = new char[strlen(str) + 1];
    strcpy(name, str);
    egn = new char[11];
    strcpy(egn, num);
}
// дефиниция на метода printPerson
void Person::printPerson() const
{
    cout << "Ime: " << name << endl;
    cout << "EGN: " << egn << endl;
}
```

Обикновени конструктори и деструктор …

```
// дефиниция на деструктора на Person
Person::~Person()
{
    cout << "~Person(): " << endl;
    delete name;
    delete egn;
}
```

Обикновени конструктори и деструктор …

```
// декларация на класа Student
class Student : Person
{
public:
   Student(char * = "", char * = "", long = 0, double = 0);
   void printStudent() const;
   ~Student() {
      cout << "~Student(): " << endl;
   }
private:
   long facnom;
   double usp;
};
```

Обикновени конструктори и деструктор …

```
//дефиниция на конструктора на класа Student
Student::Student(char *str, char * num, long facn, double u) : Person(str, num)
{
   facnom = facn;
   usp = u;
}
// дефиниция на метода printStudent
void Student::printStudent() const
{
   printPerson();
   cout << "Fac. nomer: " << facnom << endl;
   cout << "Uspeh: " << usp << endl;
}
```

Обикновени конструктори и деструктор ...

```
// декларация на класа PStudent
class PStudent : public Student
{
public:
   PStudent(char * = "", char * = "", long = 0, double = 0, double = 0);
   ~PStudent() {
      cout << "~PStudent() \n";
   }
   void printPStudent() const;
protected:
   double tax;
};
```

Обикновени конструктори и деструктор …

```
// дефиниция на конструктора на класа PStudent
PStudent::PStudent(char *str, char *num, long facn,
double u, double t) : Student(str, num, facn, u)
{
  tax = t;
}
// дефиниция на метода printPStudent
void PStudent::printPStudent() const
{
   printStudent();
   cout << "Tax: " << tax << endl;
}
```

Обикновени конструктори и деструктор …

```
void main()
{
    Person pe;
    pe.printPerson();
    PStudent PStud("Ivan Ivanov", "8206123422", 42444,
                   6.0, 4567);
    PStud.printPStudent();
}
```

Обикновени конструктори и деструктор …

Резултат:

```
Ime:
EGN:
Ime: Ivan Ivanov
EGN: 8206123422
Fac. nomer: 42444
Uspeh: 6
Tax: 4567
~PStudent()
~Student():
~Person():
~Person():
```

Конструктор за копиране и операторна функция за присвояване

Член-данните на обект на производен клас могат да получат стойности и чрез инициализиране чрез присвояване на друг обект или направо чрез присвояване. Това се осъществява чрез конструктора за копиране и операторната функция за присвояване на производния клас.

В общия случай, производният клас **не** наследява от основния клас конструктора за копиране и оператора за присвояване. Има някои изключения, на които ще се спрем по-долу.

Конструктор за копиране

При конструкторите за копиране се спазва същият принцип като при обикновените конструктори на производния и основния клас. Конструкторът за копиране на производния клас инициализира чрез присвояване собствените член-данни на класа, а конструкторът за копиране на основния клас инициализира наследените член-данни.

Конструкторите за копиране на производни класове се дефинират по същия начин като обикновените конструктори на производни класове.

Ще напомним, че ако в клас не е дефиниран конструктор за копиране, ролята на такъв се поема от генерирания системен *конструктор за копиране* <име_на_клас>(const <име_на_клас>&).

Конструктор за копиране …

Ще отбележим някои случаи на използване на конструкторите за копиране на производния и основния клас.

В производния клас не е дефиниран конструктор за копиране

Възможни са:

а) в основния клас е дефиниран конструктор за копиране

В този случай компилаторът генерира служебен конструктор за копиране на производния клас

<име_на_производен_клас>(const <име_на_ производен_клас>&),

който преди да се изпълни, **активира и изпълнява** конструктора за копиране на основния клас. Ще отбележим, че при обикновените конструктори този случай ще предизвика грешка, ако в основния клас няма подразбиращ се конструктор. Затова в случая се казва, че конструкторът за копиране на основния клас се наследява от производния клас.

Конструктор за копиране …

Задача. Да се допълни класът Person от предната задача чрез конструктор за копиране.

```
#include <iostream>
#include <string>
using namespace std;
// декларация на класа Person
class Person {
public:
   Person(char * = "", char * = "");
   Person(const Person&);
   void printPerson() const;
   ~Person();
private:
   char * name;
   char * egn;
};
```

Конструктор за копиране …

```
// дефиниция на конструктора на Person
…
// дефиниция на конструктора за копиране на Person
Person::Person(const Person& p)
{
    name = new char[strlen(p.name) + 1];
    strcpy(name, p.name);
    egn = new char[11];
    strcpy(egn, p.egn);
}
…
```

Конструктор за копиране …

```
// декларация на класа Student
class Student : Person
{
public:
   Student(char * = "", char * = "", long = 0, double = 0);
   void printStudent() const;
   ~Student() {
      cout << "~Student(): " << endl;
   }
private:
   long facnom;
   double usp;
};
…
```

Конструктор за копиране …

```
// декларация на класа PStudent
class PStudent : public Student
{
public:
   Student(char * = "", char * = "", long = 0,
   double = 0, double = 0);
   ~PStudent() {
      cout << "~PStudent() \n";
   }
   void printPStudent() const;
protected:
   double tax;
};
…
```

Конструктор за копиране …

```
void main()
{
    Student s1("Ivan Ivanov", "8206123422", 42444, 6.0);
    s1.printStudent();
    Student s2 = s1;
    s2.printStudent();
}
```

Конструктор за копиране …

В резултат от изпълнението й се получава:

```
Ime: Ivan Ivanov
EGN: 8206123422
Fac. nomer: 42444
Uspeh: 6
Ime: Ivan Ivanov
EGN: 8206123422
Fac. nomer: 42444
Uspeh: 6
~Student():
~Person():
~Student():
~Person():
```

Конструктор за копиране …

б) в основния клас не е дефиниран конструктор за копиране

В този случай се генерират “служебни” конструктори за копиране за двата класа. Конструкторът за копиране на производния клас активира конструктора за копиране на основния клас.

Нека се върнем към решението на предходната задача. Класът PStudent е производен на класа Student и в двата класа не са дефинирани конструктори за копиране. В резултат от изпълнението на функцията:

Конструктор за копиране …

```
void main()
{
   PStudent s1("Ivan Ivanov", "8206123422", 42444,
               6.0, 350);
   s1.printPStudent();
   PStudent s2 = s1;
   s2.printPStudent();
}
```

Конструктор за копиране …

```
Ime: Ivan Ivanov
EGN: 8206123422
Fac. nomer: 42444
Uspeh: 6
Tax: 350
Ime: Ivan Ivanov
EGN: 8206123422
Fac. nomer: 42444
Uspeh: 6
Tax: 350
~PStudent()
~Student():
~Person():
~PStudent()
~Student():
~Person():
```

Конструктор за копиране …

В производния клас е дефиниран конструктор за копиране

В този случай отначало се активира конструкторът за копиране на производния клас. В неговия инициализиращ списък може да има или да няма обръщение към конструктор (за копиране или обикновен) на основния клас. Препоръчва се в инициализиращия списък на производния клас да има обръщение към конструктора за копиране на основния клас, ако такъв е дефиниран. Ако не е указано обръщение към конструктор на основния клас, инициализирането на наследените членове става чрез подразбиращия се конструктор на основния клас. Ако основният клас няма такъв, ще се съобщи за отсъствието на подходящ конструктор.

Така конструкторът за копиране на производния клас чрез дефиницията си определя как точно ще се инициализира наследената част.

Конструктор за копиране …

Задача. Да се допълни и класът Student от предната задача с конструктор за копиране.

В случая това не е необходимо, защото генерирания от компилатора служебен конструктор за копиране напълно ни устройва. Предложеното решение е заради технически съображения.

```
#include <iostream>
#include <string>
using namespace std;
// декларация на класа Person
class Person
{
public:
   Person(char * = "", char * = "");
   Person(const Person&);
   void printPerson() const;
   ~Person();
private:
   char * name;
   char * egn;
};
…
```

Конструктор за копиране …

```
class Student : Person
{
public:
   Student(char * = "", char * = "", long = 0, double = 0);
   Student(const Student& st);
   void printStudent() const;
   ~Student() {
      cout << "~Student(): " << endl;
   }
private:
   long facnom;
   double usp;
};
…
```

Конструктор за копиране …

```
// дефиниция на конструктора за копиране на Student
Student::Student(const Student& st) : Person(st)
{
   facnom = st.facnom;
   usp = st.usp;
}
…
void main() {
   Student s1("Ivan Ivanov", "8206123422", 42444, 6.0);
   s1.printStudent();
   Student s2 = s1;
   s2.printStudent();
}
```

Конструктор за копиране …

Забележете, аргументът на обръщението Person(st), в инициализиращия списък на конструктора за присвояване на класа Student, е от тип const Student&, а не const Person& (По-нататък ще разгледаме преобразуването на типовете).

Друга реализация на конструктора за копиране на класа Student е:

```
Student::Student(const Student& st) : Person(st.name, st.egn)
{
    facnom = st.facnom;
    usp = st.usp;
}
```

ако член-данните name и egn на класа Person са обявени в секция protected. В този случай за инициализиране на наследените член-данни е използван двуаргументният конструктор на Person.

Конструктор за копиране ...

Ако конструкторът за копиране на класа Student има вида:

```
Student::Student(const Student& st)
{
    facnom = st.facnom;
    usp = st.usp;
}
```

Конструктор за копиране …

след изпълнението на фрагмента

```
Student s1("Ivan Ivanov", "8206123422", 42444, 6.0);

Student s2 = s1;

s2.printStudent ();
```

полетата name и egn на s2 се инициализират с празния низ, а facnom и успех с 42444 и 6 съответно.

Причината е, че в инициализиращия списък на конструктора за присвояване на класа Student не е указан начинът за инициализиране на член-данните на основния клас. В този случай инициализацията се осъществява чрез конструктора по подразбиране на основния клас Person.

Операторна функция за присвояване

Операторната функция за присвояване на производен клас трябва да указва как да стане присвояването както на собствените, така и на наследените си член-данни. За разлика от конструкторите на производни класове тя прави това в тялото си (не поддържа инициализиращ списък). Използването ѝ зависи от това дали такава е дефинирана в производния клас. Ще напомним, че ако в клас не е дефинирана операторна функция за присвояване, компилаторът създава operator=(const <име_на_клас>&).

Ще разгледаме следните два случая:

В производния клас не е дефинирана операторна функция за присвояване

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**, чрез която инициализира наследената част, след това инициализира чрез присвояване и собствената част на производния клас. Затова в този случай се казва, че операторът за присвояване на основния клас се наследява, т.е. за наследените член-данни се използва подразбиращият се или предефинираният оператор за присвояване на основния клас.

В производния клас е дефинирана операторна функция за присвояване

Дефинираният в производния клас оператор за присвояване **трябва да се погрижи за наследените компоненти. Налага се в тялото на неговата дефиниция 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 се казва, че операторът за присвояване на основния клас не се наследява.

Операторна функция за присвояване…

Пример: Да разгледаме следната йерархична схема от 4 класа: базов клас base и три негови наследници: der1, der2 и der3, реализирана чрез програмата:

```
#include <iostream>
using namespace std;
class Base {
public:
   Base(int x = 0) { b = x; }
   Base& operator=(const Base &x) {
      if (this != &x)
         b = x.b + 1;
      return *this;
   }
protected:
   int b;
};
```

Операторна функция за присвояване…

```
class Der1 : public Base {
public:
   Der1(int x = 1) { d = x; }
   Der1& operator=(const Der1& x) {
      if (this != &x) {
         d = x.d + 2;
         b = x.b + 3;
      }
      return *this;
   }
   void Print() {
      cout << "Der: " << d << "  Base: " << b << endl;
   }
private:
   int d;
};
```

Операторна функция за присвояване…

```
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() {
      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() {
      cout << "Der: " << d << "  Base: " << b << endl;
   }
private:
   int d;
};
```

Операторна функция за присвояване…

```
void 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();
}
```

Операторна функция за присвояване…

В резултат от изпълнението й се получава:

```
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
```

Операторна функция за присвояване…

Класовете der1, der2 и der3 са дефинирани и използвани по идентичен начин с изключение на предефинирания оператор за присвояване.

В класа der1 операторът за присвояване е предефиниран и се грижи за наследената част.

В класа der2 операторът за присвояване е предефиниран, но не указва как става присвояването на наследената член-променлива. Тъй като операторът за присвояване на базовия клас в този случай не се наследява, стандартът на езика не уточнява стойността на наследената член-променлива на обекта d22. В този случай нейната стойност е тази от инициализацията der2 d22.

В класа der3 операторът за присвояване не е предефиниран. Тогава за собствените на класа компонента се използва подразбиращият се, а за наследената – предефинираният оператор за присвояване на базовия клас се наследява и изпълнява.

Операторна функция за присвояване…

Задача. Да се допълнят класовете Person и Student от предишната задача с операторни функции за присвояване.

```
#include <iostream>
#include <string>
using namespace std;
// декларация на класа Person
class Person
{
public:
   Person(char * = "", char * = "");
   Person(const Person&);
   Person& operator=(const Person& p);
   void printPerson() const;
   ~Person();
private:
   char * name;
   char * egn;
};
```

Операторна функция за присвояване…

```
…
Person& Person::operator=(const Person& p) {
   if (this != &p) {
      delete [] name;
      delete [] egn;
      name = new char[strlen(p.name) + 1];
      strcpy(name, p.name);
      egn = new char[11];
      strcpy(egn, p.egn);
    }
    return *this;
}
…
```

Операторна функция за присвояване…

```
// декларация на класа Student
class Student : Person
{
public:
   Student(char * = "", char * = "", long = 0, double = 0);
   Student(const Student& st);
   void printStudent() const;
   Student& operator=(const Student& st);
   ~Student() {
      cout << "~Student(): " << endl;
   }
private:
   long facnom;
   double usp;
};
```

Операторна функция за присвояване…

```
…
Student& Student::operator=(const Student& st)
{
    if (this != &st) {
        Person::operator=(st);
        facnom = st.facnom;
        usp = st.usp;
    }
    return *this;
}
…
```

Операторна функция за присвояване...

```
// декларация на класа PStudent
class PStudent : public Student
{
public:
   PStudent(char * = "", char * = "", long = 0,
   double = 0, double = 0);
   PStudent& operator=(const PStudent& st);
   ~PStudent() {
      cout << "~PStudent() \n";
   }
   void printPStudent() const;
protected:
  double tax;
};
```

Операторна функция за присвояване…

```
PStudent& PStudent::operator=(const PStudent& st)
{
   if (this != &st) {
      Student::operator=(Student(st));
      tax = st.tax;
   }
   return *this;
}
```

Операторна функция за присвояване…

```
void main()
{
   PStudent s1("Ivan Ivanov", "8206123422", 42444, 6.0, 4444);
   s1.printPStudent();
   PStudent s2("Jonko Dimov", "9012074442", 43344, 5, 3434);
   s2.printPStudent();
   s2 = s1;
   s2.printPStudent();
}
```