Обектно ориентирано програмиране

МНОЖЕСТВЕНО НАСЛЕДЯВАНЕ. ВИРТУАЛНИ КЛАСОВЕ

Множествено наследяване

В случаите, когато производният клас наследява няколко основни класа, се казва, че класът е с множествено наследяване. Този вид наследяване е мощен инструмент на ООП, тъй като чрез него се изграждат графовидни йерархични структури.

Деклариране на производен клас с множествено наследяване

<декларация_на_производен_клас> ::=

class <име_на-производен_клас> :

[<атрибут_за_област>] <име_на_базов_клас$_1$>,

[<атрибут_за_област>] <име_на_базов_клас$_2$>{,

[<атрибут_за_област>] <име_на_базов_клас$_n$>}

{<декларация_на_компоненти>

};

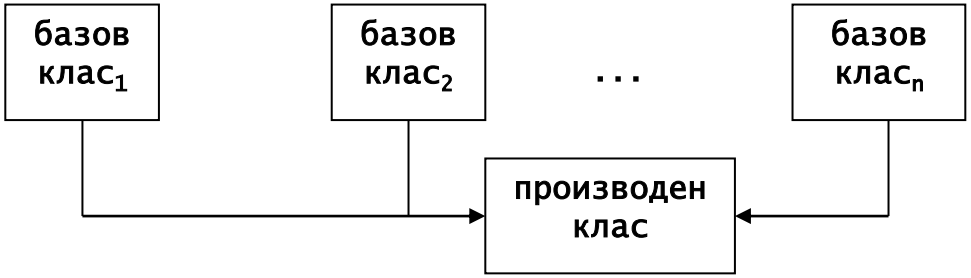
<име_на-производен_клас> ::= <идентификатор>

<атрибут_за_област> ::= public | private | protected

<име_на_базов_класі> ::= <идентификатор>

Множествено наследяване ...

Атрибутите за област се задават за всеки базов клас. Семантиката им беше вече пояснена при разглеждане на производни класове с единично наследяване. Ако за някои клас атрибутът за област е пропуснат, подразбира се private.



Множествено наследяване ...

Производният клас наследява компонентите на всички базови класове като видът на наследяване private, public или protected се определя от атрибута за област на базовия клас. Правилата са същите като при единичното наследяване.

За член-функциите на голямата четворка на производен клас с множествено наследяване са в сила същите правила, като при производен клас с единично наследяване. В общия случай тези член-функции за основните класове не се наследяват от производния им клас. Изключенията отново са при конструкторите за присвояване и операторните функции за присвояване.

Ще напомним дефиницията на конструктора на производен клас с множествено наследяване

Множествено 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$_i$>)

{, <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.
- В <инициализиращ_списък> може да има повече от едно обръщение към конструктор на основен клас.

Множествено наследяване ...

При извикването на този конструктор последователно се извикват:

а) Конструкторите на базовите класове по реда на тяхното задаване не в инициализиращия списък на конструктора, а в декларацията на производния клас.

б) Конструкторите на класовете, чиито обекти са членове на производния клас. Редът на извикване съответства на реда на деклариране на тези членове в тялото на производния клас.

в) Конструкторът на производния клас.

Отново са възможни:

1. ***В основен клас не е дефиниран конструктор***

   В този случай в инициализиращия списък на конструктора на производния клас не се прави обръщение към конструктора на този клас и наследената му част остава неинициализирана.

Множествено наследяване ...

2. ***В основен клас има един конструктор с параметър, от който не следва подразбиращия се***

   Тогава ако в производния клас е дефиниран конструктор, в инициализиращия му списък задължително трябва да има обръщение към конструктора с параметър на този основен клас. Ако в производния клас не е дефиниран конструктор за присвояване, компилаторът ще сигнализира грешка.

3. ***Основен клас има няколко конструктора в т. число подразбиращ се***

   Ако в производния клас е дефиниран конструктор, в инициализиращия му списък може да не се посочва конструктор за този основен клас. Ще се използва подразбиращия се. Ако в производния клас не е дефиниран конструктор, компилаторът автоматично създава за него подразбиращ се конструктор. В този случай обаче всички основни класове на производния клас трябва да имат подразбиращ се конструктор.

Множествено наследяване ...

Дефинирането на деструкторите става по описания вече начин. Всеки деструктор се грижи само за собствените си компоненти. Извикването им става в обратен ред – първо се извиква деструкторът на производния клас, след това, в обратен ред, се извикват деструкторите на класовете на обектите – членове на производния клас и най-накрая на основните му класове, отново в обратен ред на реда на извикване на техните конструктори.

Пример:

```
#include <iostream>

using namespace std;

class base1

{

 public:

   base1(int x = 0)

   {

      cout << "base1:\n";

      b1 = x;

   }
```

Множествено наследяване ...

```
   ~base1()
   { cout << "~base1()\n"; }
 private:
   int b1;
};
class base2
{
 public:
   base2(int x = 0)
   {
      cout << "base2:\n";
      b2 = x;
   }
   ~base2()
   { cout << "~base2()\n"; }
 private:
   int b2;
 };
```

Множествено наследяване ...

```
class base3
{
 public:
   base3(int x = 0)
   {
      cout << "base3:\n";
      b3 = x;
   }
   ~base3()
   { cout << "~base3()\n"; }
 private:
   int b3;
};
class der : public base2, base1, base3
{
 public:
```

Множествено наследяване

...

```
   der(int x = 0) : base3(x), base1(x), base2(x)
   { d = 5; }
 private:
   int d;
};
void main()
{
   der d1(5);
}

 base2
 base1
 base3
 ~base3()
 ~base1()
 ~base2()
```

Множествено наследяване ...

Същият е резултатът от изпълнението на програмата ако от инициализиращия списък на класа der пропуснем някое от обръщенията към основните класове base1, base2 или base3, или даже всичките. В тези случаи се използват конструкторите по подразбиране на основните класове. Резултатът от изпълнението не се променя ако в производния клас der не е дефиниран конструктор. В този случай компилаторът създава за der конструктор по подразбиране, който се обръща към конструкторите по подразбиране на основните класове в реда, указан в декларацията на производния клас.

В тази програма класът der има две собствени и 3 наследени компоненти (конструкторите и деструкторите не се наследяват). Може да си мислим за него като клас от вида:

Множествено наследяване

...

```
class der : public base2, base1, base3
{
public:
   der(int x = 0) : base3(x), base1(x), base2(x)
   { d = 5;}
private:
  int d;
  int b1, b2, b3;
};
```

Дефиницията der d1(5); предизвиква създаване на обект d1 с компоненти d, b3, b1, b2 и извикване на конструктора на класа der с параметър 5. Преди да се изпълни неговото тяло се изпълняват конструкторите на базовете класове base2, base1 и base3 с параметър 5. Това инициализира отделената памет за член-данните на d1 с 5.

Множествено наследяване ...

Дефинирането на конструктора за копиране и операторната функция за присвояване на производен клас с множествено наследяване се извършва по същия начин както при единичното наследяване.

Ще напомним, че ако в производния клас не е дефиниран конструктор за копиране, компилаторът генерира за него “служебен” конструктор за копиране, който преди да се изпълни активира и изпълнява конструкторите за копиране на всички основни класове в реда, указан в декларацията на производния клас.

Ако в производния клас е дефиниран конструктор за копиране, препоръчва се в инициализиращия му списък да има обръщения към конструкторите за копиране на основните класове (ако такива са дефинирани).

Множествено наследяване

...

Операторната функция за присвояване на производен клас с множествено наследяване има вида:

```
<производен_клас>&
<производен_клас>::operator=(const<производен_клас>& p)
{
   if (this != &p)
   {
      <основен_клас1>::operator=(p);
      <основен_клас2>::operator=(p);
      ...
      <основен_класN>::operator=(p);
      // дефиниране на присвояването
      // за собствените за класа компоненти
      Del();  // изтриване от ДП на собствените компоненти на
              // подразбиращия се обект
      Copy(p); // копиране на собствените компоненти на p в
               // подразбиращия се обект
   }
   return *this;
}
```

Множествено наследяване ...

Ако в производния клас не е дефинирана операторна функция за присвояване, компилаторът създава такава. Тя изпълнява операторните функции за присвояване на всички основни класове на производния клас.

Ако в производния клас е дефинирана операторна функция за присвояване, тя трябва да се погрижи за присвояването на наследените компоненти. Ако това не е направено явно, стандартът на езика не уточнява как ще стане присвояването на наследените компоненти.

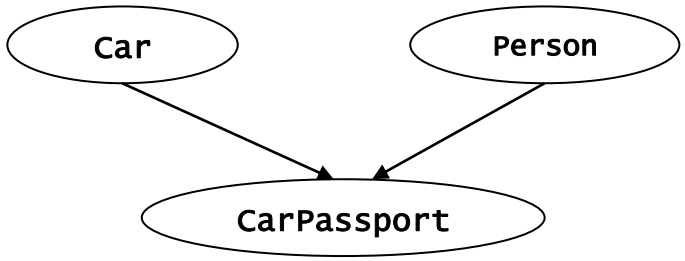
Чрез обект на производния клас могат да се викат всички негови public компоненти – собствени и наследени. Ако в базовите класове са дефинирани компоненти с еднакви имена, необходимо е да се използва пълното име на компонентата, т.е. <име_на_клас>::<компонента>. Ако всички основни класове basei на производния клас der имат член-функция f() и член-данна x с еднакви имена, различаването им в производния клас става чрез пълните им имена:

```
base1::f(), base2::f(), ...

base1::x, base2::x, ...
```

Множествено наследяване ...

Задача. Да се дефинират класове Car и Person, определящи понятията “автомобил” и “човек”. Да се дефинира клас CarPassport, производен на класовете Car и Person и определящ понятието “паспорт на автомобил”. За всеки от класовете да се определи голямата четворка.



Множествено наследяване ...

```
#include <iostream>
#include <string>
using namespace std;
class Car
{
 public:
   Car(char * = "", unsigned = 0, unsigned = 0);
   ~Car();
   Car(const Car&);
   Car& operator=(const Car&);
   void display() const;
 private:
   char *mark;
   unsigned year;
   unsigned reg_numb;
};
```

Множествено наследяване

...

```
Car::Car(char *m, unsigned y, unsigned r_n)
{
   mark = new char[strlen(m)+1];
   strcpy(mark, m);
   year = y;
   reg_numb = r_n;
}
Car::~Car()
{
   cout << "~Car()\n";
   delete mark;
}
Car::Car(const Car& c)
{
   mark = new char[strlen(c.mark)+1];
   strcpy(mark, c.mark);
```

Множествено наследяване ...

```
   year = c.year;
   reg_numb = c.reg_numb;
}
Car& Car::operator=(const Car& c)
{
   if (this != &c)
   {
      delete mark;
      mark = new char[strlen(c.mark)+1];
      strcpy(mark, c.mark);
      year = c.year;
      reg_numb = c.reg_numb;
   }
   return *this;
}
```

Множествено наследяване

...

```
void Car::display() const
{
   cout << "Mark: " << mark << endl;
   cout << "Year: " << year << endl;
   cout << "Reg. Number: " << reg_numb << endl;
}
class Person
{
 public:
   Person(char * = "", char * = "");
   Person(const Person&);
   Person& operator=(const Person& p);
   void display() const;
   ~Person();
 private:
   char * name;
   char * egn;
};
```

Множествено наследяване ...

```
Person::Person(char *str, char *num)
{
   name = new char[strlen(str)+1];
   strcpy(name, str);
   egn = new char[11];
   strcpy(egn, num);
}
Person::Person(const Person& p)
{
   name = new char[strlen(p.name)+1];
   strcpy(name, p.name);
   egn = new char[11];
   strcpy(egn, p.egn);
}
```

Множествено наследяване ...

```
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;
}
void Person::display() const
{
   cout << "Ime: " << name << endl;
   cout << "EGN: " << egn << endl;
}
```

Множествено наследяване ...

```
Person::~Person()
{
   cout << "~Person()\n";
   delete name;
   delete egn;
}
class CarPassport: public Car, public Person
{
 public:
   CarPassport(char* = "", unsigned = 0, unsigned = 0,
               char* = "", char* = "");
   ~CarPassport();
   CarPassport(const CarPassport&);
   CarPassport& operator=(const CarPassport&);
   void display() const;
};
```

Множествено наследяване ...

```
CarPassport::CarPassport(char *mark, unsigned year,
                         unsigned reg_numb, char* name, char *egn) :
                        Car(mark, year, reg_numb), Person(name, egn)
{
}
CarPassport::~CarPassport()
{
   cout << "~CarPassport()\n";
}
CarPassport::CarPassport(const CarPassport& cp) :
             Car(cp), Person(cp)
{
}
CarPassport& CarPassport::operator=(const CarPassport& cp)
{
   if (this != &cp)
```

Множествено наследяване ...

```
    {
       Car::operator =(cp);
       Person::operator =(cp);
    }
    return *this;
}
void CarPassport::display() const
{
    Car::display();
    Person::display();
}
```

Множествено наследяване

...

```
void main()
{
   CarPassport x("FORD FIESTA", 2000, 2295,
                 "Vassil Todorov", "8012174586");
   x.display();
   CarPassport y("LADA", 1900, 8817,
                 "Sonia Todorova", "8203314576");
   y = x;
   y.display();
}
```

В резултат се получава:

```
Mark: FORD FIESTA
Year: 2000
Reg. Number: 2295
Ime: Vassil Todorov
EGN: 8012174586
```

Множествено наследяване

...

```
Mark: FORD FIESTA
Year: 2000
Reg. Number: 2295
Ime: Vassil Todorov
EGN: 8012174586
~CarPassport()
~Person()
~Car()
~CarPassport()
~Person()
~Car()
```

Множествено наследяване ...

Забележка: Няма значение как се изброяват основните класове в списъка на производния клас с множествено наследяване. Но вече фиксирана, тази последователност се използва при:

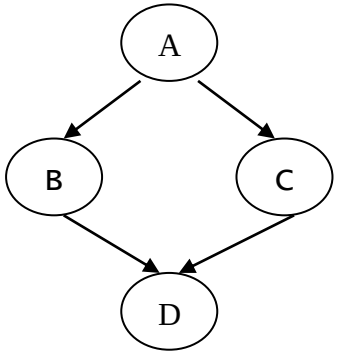
- разполагане на наследените части на производния клас в паметта;
- неявното извикване на конструкторите (деструкторите) на основните класове.

Освен това редът на записване на конструкторите на основния клас в инициализиращия списък на конструктора на производния клас няма значение. Конструкторите ще се активират в реда им в декларацията на производния клас. Ще отбележим още веднъж, че когато се използва множествено наследяване, изискванията към основните класове и съгласуването им с производния клас са както в случая на единичното наследяване.

Виртуални класове

Стандартният механизъм за наследяване дефинира йерархия, която се представя чрез дърво. В този случай, при всяко срещане на основен клас се създава копие на неговите член-данни. Виртуалните основни класове са механизъм за отменяне на стандартния наследствен механизъм.

При реализиране на йерархии на класове с множествено наследяване е възможно един производен клас да наследи многократно даден базов клас. Например, ако класът A има два производни класа B и C, които са базови за класа D



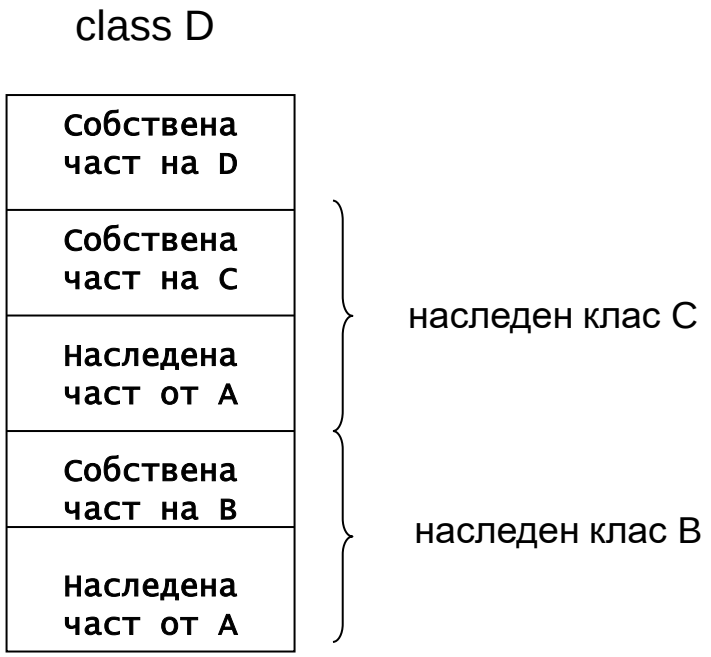
Виртуални класове …

```
class A
{...
};
class B: public A
{...
};
class C: public A
{...
};
class D: public B, public C
{...
};
```

Като производни на класа A, класовете B и C наследяват неговите компоненти. От друга страна класовете B и C са базови за класа D. Следователно класът D ще наследи два пъти компонентите на класа A, веднъж чрез класа B и още веднъж – чрез класа на C. За член-функциите двойното наследяване не е от значение, тъй като за всяка член-функция се съхранява само едно копие. Член-променливите обаче се дублират и обект на класа D ще наследи двукратно всяка член-променлива, дефинирана в класа A. Класът D в паметта ще има вида:

Виртуални класове …

Този пример илюстрира един от недостатъците на многократното наследяване на клас – *неефективността от поддържането на множество копия на наследени компоненти*.



Виртуални класове …

Ще покажем и други недостатъци на многократното наследяване на класове. Да разгледаме още веднъж горната йерархична схема като попълним декларацията на класовете A, B, C и D. Конструктор по подразбиране няма да дефинираме за нито един от тези класове.

```
class A
{
 public:
   A(int a)
   { x = a;}
   int f() const;
   void print() const;
   int x;
};
```

Виртуални класове ...

```
int A::f() const
{
   return x;
}
void A::print() const
{
   cout << "A:: x " << x << endl;
}
class B: public A
{
 public:
   B(int a, int b): A(a)
   { x = b; }
   int f() const;
   void print() const;
   int x;
};
```

Виртуални класове …

```
int B::f() const
{
   return x;
}
void B::print() const
{
   A::print();
   cout << "B:: x " << x << endl;
}
class C: public A
{
 public:
   C(int a, int c): A(a)
   { x = c; }
   int f() const;
   void print() const;
   int x;
};
```

Виртуални класове ...

```
int C::f() const
{
   return x;
}
void C::print() const
{
   A::print();
   cout << "C:: x " << x << endl;
}
class D: public B, public C
{
 public:
   D(int a, int b, int c, int d): B(a, b), C(c, d)
   {}
   void D::func() const;
   void print() const;
};
```

Виртуални класове …

```
void D::print() const
{
   B::print();
   C::print();
}
```

След дефиницията:

```
D d(1, 2, 3, 4);
```

се получава неоднозначност: наследената двукратно член-данна x на класа A има две стойности: 1 и 3. Този проблем можем да разрешим като дефинираме конструктора на класа D по следния начин:

```
D(int a, int b, int c): B(a, b), C(a, c)
  {}
```

но двукратно заделената памет си остава.

Виртуални класове …

Пример: В резултат от компилирането на функцията:

```
void D::func() const

{ A::print(); }
```

се получава грешката: *error C2385: 'D::A' is ambiguous*. Причината е, че единственият аргумент на func е this, който е и аргумент на print() в A::print(). this е указател към обект на класа D, който обект съдържа два обекта от основния клас A. Кой от тях да се свърже с извиканата функция print()? Грижата да се посочи един от двата обекта си остава на програмиста.

Ако е необходим само достъп до “конфликтните” наследени членове на клас A (без да се променят стойностите), осъществяването му става чрез последователно прилагане на операцията за явно преобразуване на типове. При атрибут за област public, обект на производен клас може да се преобразува в обект на основен клас с неявни преобразувания, но поради двата клона в йерархичната схема, обект от клас D не може да се преобразува директно в обект от клас A. Възможни са следните последователни преобразувания:

Виртуални класове …

```
D d(1, 2, 3, 4);

(A)(B) d;

(A)(C) d;
```

Като се използват подобни преобразувания, може да се осигури достъп до наследените от класа A членове на класа D. Ще ги илюстрираме с дефиниция на член-функцията func() на D.

```
void D::func() const
{
cout << "Derived member x in a part A-B-D "
     << ((A)(B)*this).x << endl
     << "Derived member x in a part A-C-D "
     << ((A)(C)*this).x << endl
     << "Derived member-function f() in a part A-B-D "
     << ((A)(B)*this).f() << endl
     << "Derived member-function f() in a part A-C-D "
     << ((A)(C)*this).f() << endl
     << "Derived member-function f() in part C-D"
     << ((C)*this).f() << endl
```

Виртуални класове ...

```
        << "Derived member-function f() in part B-D"
        << ((B)*this).f() << endl;
}
```

Резултатът от изпълнението на фрагмента:

```
D d(1, 2, 3, 4);
d.func();
```

е:

```
Derived member x in a part A-B-D 1
Derived member x in a part A-C-D 3
Derived member-function f() in a part A-B-D 1
Derived member-function f() in a part A-C-D 3
Derived member-function f() in part C-D 4
Derived member-function f() in part B-D 2
```

Виртуални класове ...

Ще напомним, че указателят this сочи обект от клас D, *this е неговото съдържание, т.е. *this е обект от тип D. Чрез явните преобразувания (A)(B)*this този обект се превръща в обект отначало от клас B, а след това в обект от клас A. Чрез оператора . се прави достъп до член на съответния клас. *При тези преобразувания обектите от класове B и A са **временни**. Това е причината, заради която е възможен само достъп, а не промяна на стойности, т.е. при опит за промяна, тя ще е във временна, а не в постоянната памет*.

Ще отбележим също, че обръщението

```
d.print();
```

извиква два пъти метода A::print(), веднъж от обръщението B::print() и друг път от C::print().

Виртуални класове ...

Многократното наследяване на клас води от една страна до затруднен достъп до многократно наследените членове, а от друга до поддържане на множество копия на член-данните на многократно наследения клас, което не е ефективно.

Преодоляването на недостатъците на многократното наследяване на клас се осъществява чрез използването на т.н. ***виртуални основни класове***. Чрез тях се дава възможност да се “поделят” основни класове. Когато един клас е виртуален, независимо от участието му в няколко списъка на основни класове, се създава само едно негово копие. В нашия случай, ако класът A се определи като виртуален за класовете B и C, класът D ще съдържа само един “поделен” основен клас A.

Виртуални класове …

Декларация

Декларацията на основен клас като виртуален се осъществява като в декларацията на производния клас заедно с името и атрибута за област на основния клас се укаже и запазената дума virtual.

Пример: Ще променим декларацията на йерархичната схема като определим класа A като виртуален на класовете B и C:

```
class A
{
public:
   ...
};
...
```

Виртуални класове …

```
class B: virtual public A
{
 public:
    …
};
…
class C: virtual public A
{
 public:
    …
};
…
```

Виртуални класове …

```
class D: public B, public C
{
 public:
  D(int a, int b, int c, int d): A(a), B(a, b), C(c, d)
  {}
void D::func() const;
void print() const;
};
void D::print() const
{
   B::print();
   C::print();
}
```

Виртуални класове ...

Така класът A е обявен за виртуален. Казва се, че **B и C наследяват класа A виртуално**. Виртуалното наследяване на класа A от класовете B и C оказва влияние само на производните на B и C класове. То не променя поведението на самите класове B и C. Забелязваме, че запазената дума virtual е поставена пред атрибута за област на виртуалния клас A. Всъщност, няма значение редът на virtual и атрибута за област.

Дефинирането и използването на виртуални класове има редица особености. Една от тях касае дефиницията и използването на конструкторите на наследените класове.

Правилото за извикване на конструктори с параметри на виртуални класове може да се изкаже така: конструкторите с параметри на виртуални класове трябва да се извикват от конструкторите на всички класове, които са техни наследници, а не само от конструкторите на преките им наследници. С други думи, производният клас е отговорен за инициализирането на класовете, от които произлиза, както и на всички виртуални основни класове. Ако в инициализиращия списък на конструктора на производен клас няма обръщение към конструктор с параметър на виртуалния клас, използва се неговия подразбиращ се конструктор, ако такъв съществува или се съобщава за отсъствието на подходящ конструктор.

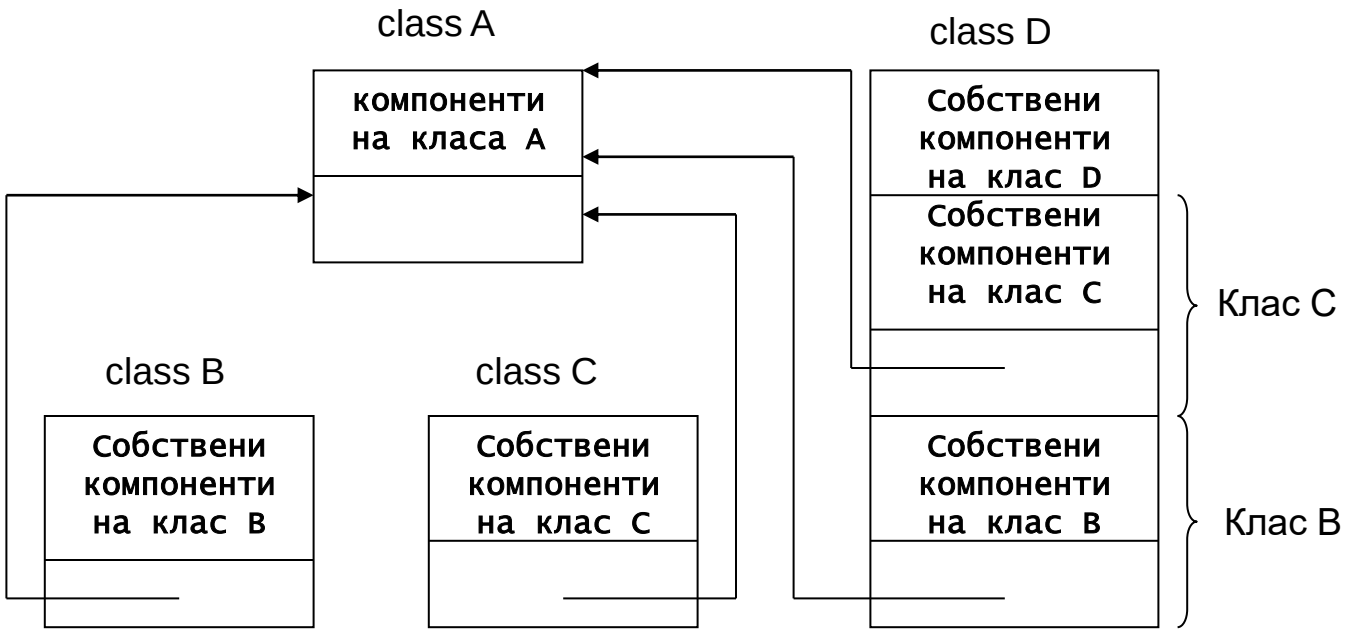
Виртуални класове …

Друга особеност е промяната на реда на инициализиране. Инициализирането на виртуални основни класове предхожда инициализирането на другите основни класове в декларацията на производния клас. Ако производен клас наследява основен и виртуален клас, конструкторът на виртуалния клас се извиква първи. При няколко виртуални класа извикването на конструкторите става по реда им в декларацията на производния клас.

Как използването на виртуални основни класове преодолява недостатъците на многократното наследяване?

Виртуалният основен клас е общ за всички производни от него класове. Ще се върнем към разглежданата по-горе йерархия на класове и схематично ще опишем как тя се реализира след обявяването на клас A за виртуален.

Виртуални класове ...



Виртуални класове …

Фигурата показва как е преодолян проблемът с многократното наследяване на член-данните на клас. Обръщенията A::x, A::f() или A::print() в класа D вече не създават проблем, тъй като класът A се наследява вече само веднъж. Резултатът от изпълнението на фрагмента:

```
D d(1, 2, 3, 4);
d.func();
```

е:

```
Derived member x in a part A-B-D 1
Derived member x in a part A-C-D 1
Derived member-function f() in a part A-B-D 1
Derived member-function f() in a part A-C-D 1
Derived member-function f() in part C-D 4
Derived member-function f() in part B-D 2
```

Виртуални класове ...

Използването на виртуални основни класове не премахва нееднозначността при достъп до някои член-функции. Например, обръщението

```
d.print();
```

ще извика метода A::print() отново два пъти, т.е. резултатът от изпълнението на обръщението:

```
d.print();
```

е:

```
A:: x 1
B:: x 2
A:: x 1
C:: x 4
```

Това може да се избегне чрез подходящо дефиниране на методите print.