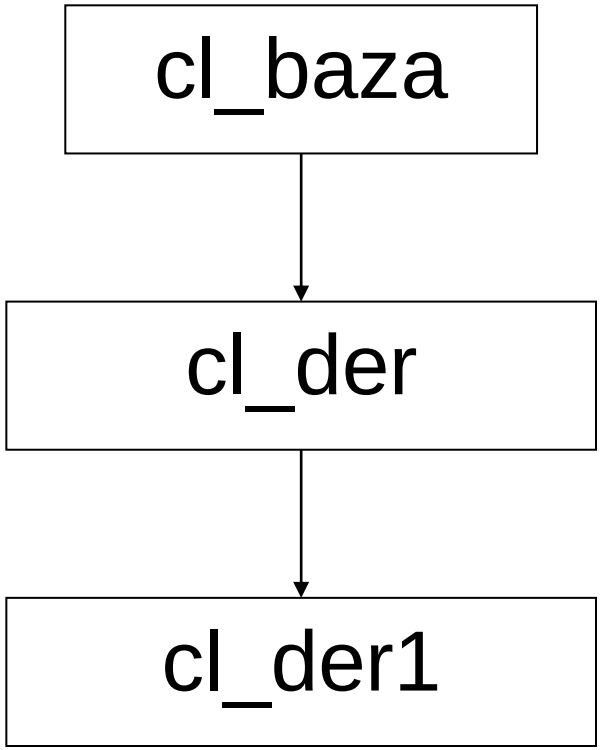
§7. Moştenirea pe mai multe niveluri. Clase virtuale

Atunci când o clasă transmite parametri sau funcționalități altei clase care, la rândul său, se consideră clasă de bază pentru o altă ierarhie de moştenire, vom spune că avem moştenire pe mai multe niveluri.

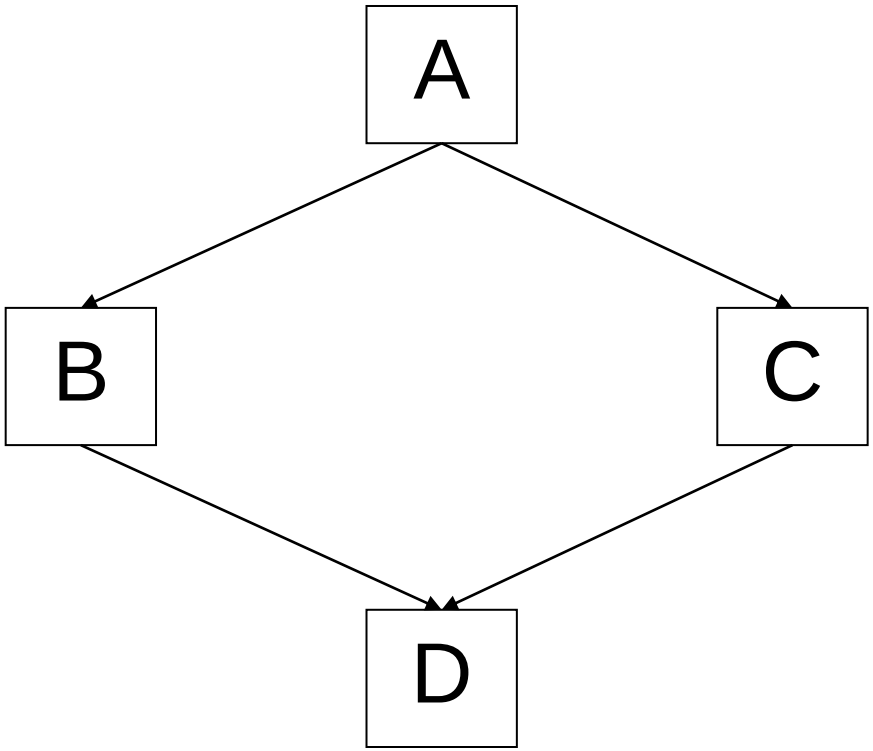
Cea mai simplă diagramă care exemplifică moştenirea pe mai multe niveluri este următoarea:



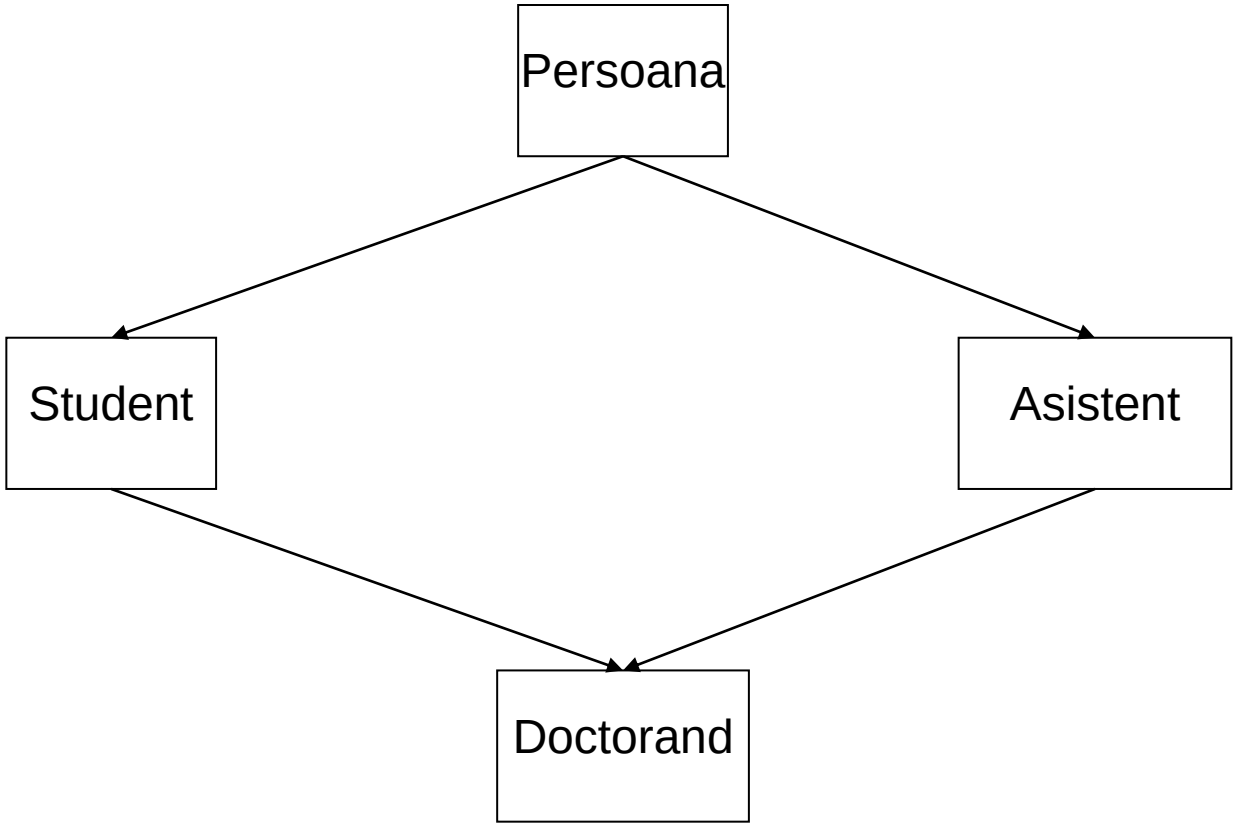
Schema de moştenire pe mai multe niveluri se reduce la o aplicare consecutivă a schemei de moştenire pe un singur nivel. Schemele de moştenire pe mai multe niveluri pot fi reprezentate prin grafuri cu mai multe niveluri.

Diagramele reprezentate prin grafuri care nu conțin cicluri nu prezintă careva dificultăți de realizare.

Dacă însă grafurile au cicluri, apar unele probleme. Cea mai simplă diagramă de acest fel este de următorul tip:



Pentru o astfel de diagramă există şi interpretări care descriu situații reale:



La realizarea unei astfel de scheme, membrii din clasa `Persoana` vor ajunge în clasa `Doctornad` pe două căi: una prin clasa `Asistent` şi cealaltă prin clasa `Student`. Deci clasa `Doctorand` va avea două exemplare pentru fiecare membru venit din clasa `Persoana`.

Generalizând cele spuse anterior, s-ar putea afirma că dacă dintr-un vârf ierarhic superior al diagramei există mai multe drumuri către un alt vârf ierarhic inferior, atunci fiecare membru al vârfului ierarhic superior va ajunge în vârful ierarhic inferior de atâtea ori câte drumuri există între aceste vârfuri.

Uneori, în situația apariției mai multor membri care vin în aceeaşi clasă, apar situații de conflict. S-ar putea evita astfel de situații, prin utilizarea claselor virtuale la moştenire. Pentru a declara o clasă virtuală la moştenire în schema de moştenire se utilizează cuvântul-cheie **virtual** înaintea clasei date. Clasele virtuale sunt protejate de dublări ale membrilor.

În continuare, vor fi exemplificate ideile expuse anterior utilizând diagrama precedentă care descrie relația dintre clasele A, B, C, D.

```
#include<iostream.h>
class A
{
  protected:
     int ia;
  public:
     A(int i){ia=i;}
};
```

```
class B:public A
{
  protected:
     int ib;
  public:
     B(int ia, int i) : A(ia){ ib=i; }
};
//--------------
class C:public A
{
  protected:
     int ic;
  public:
     C(int ia, int i) : A(ia){ ic=i; }
};
```

```
class D:public B, public C
{
      int id;
  public:
      D(int ia, int ib, int ic, int i) :
                          B(ia, ib),C(ia, ic){ id=i; }
      void afisare()
      {
       cout<<”ia=”<<ia<<”ib=”<<ib<<”ic=”<<ic
           <<”id=”<<id<<endl;
      }
};
//==============
void main()
{
      D d(7,17,27,37);
      d.afisare();
}
```

Exemplul anterior are o problemă: funcția membru `afisare()`, apelată prin intermediul obiectului `d`, va genera o eroare din motiv că membrul `ia` nu poate fi afişat, deoarece în clasa `D` există doi membri cu numele `ia`.

Va trebui schimbat modul de afişare după cum urmează:

```
void afisare()
  {
cout<<"ia="<<B::ia<<"ib="<<ib
     <<"ic="<<ic<<"id="<<id
     <<endl;
  }
```

afişând membrul `ia` ce vine prin clasa `B`. După această schimbare, programul va putea fi lansat, dar clasa `D` va continua să aibă doi membri cu numele `ia`. Pentru a scăpa de acest neajuns, clasa `A` va fi declarată clasă virtuală.

Declarând clasa `A` clasă virtuală, va fi făcută o schimbare mică, dar esențială a exemplului. Iată ce se obține:

```
#include<iostream.h>
class A
{
  protected:
    int ia;
  public:
    A(int i){ia=i;}
};
```

```
class B:public virtual A
{
  protected:
     int ib;
  public:
     B(int ia, int i) : A(ia){ ib=i; }
};
//--------------
class C:virtual public A
{
  protected:
     int ic;
  public:
     C(int ia, int i) : A(ia){ ic=i; }
};
```

```
class D:public B, public C
 {
      int id;
   public:
      D(int ia, int ib, int ic, int i) :
                    B(ia, ib),C(ia, ic){ id=i; }
     void afisare()
      {
       cout<<"ia="<<ia<<"ib="<<ib<<"ic="<<ic
           <<"id="<<id<<endl;
      }
};
//===============
void main()
{
  D d(7,17,27,37);
  d.afisare();
}
```

Ordinea de execuție a constructorilor în scheme de moştenire ce conțin clase virtuale este următoarea: sunt evidențiate clasele virtuale şi sunt executați constructorii lor în ordinea în care sunt întâlniți în descrierea clasei derivate. După ce îşi termină lucrul constructorii virtuali, vor lucra constructorii ne-virtuali ai claselor ne-virtuale, tot în ordinea apariției în schema de moştenire.

Membrii transmişi din clasa de bază în clasa derivată au tendința de a-şi micşora gradul de acces din punctul de vedere al clasei derivate. În scheme de moştenire pe mai multe niveluri această micşorare poate fi şi mai accentuată, existând posibilitate de existență a membrilor din clasa de bază, care ajung inaccesibili într-o oarecare clasă derivată de la anumit nivel.

Există posibilitatea de a mări gradul de acces al membrilor transmişi dintr-o clasă de bază într-o clasă derivată.

Pentru a mări gradul de acces al unui membru transmis dintr-o clasă de bază într-o clasă derivată, numele membrului precedat de numele clasei şi operatorul rezoluției sunt plasate în raza de acțiune a modificatorului de acces necesar.

```
modificator_acces:
    . . .
    nume_cl::nume_mem;
```

Trebuie de ținut cont că gradul de acces poate fi mărit readucându-l la acelaşi grad pe care la avut în clasa de bază. Nu sunt permise gradații de acces intermediare.

De exemplu, fie clasa `derivata`, care este derivată din clasa `baza`:

```
class baza
{
 public:
  int ib;
};
//--------
class derivata : private baza
{
  int id;
 public:
  baza::ib;
};
```

Membrul `ib` în clasa derivata va avea gradul de acces `private` şi îl readucem la gradul de acces `public`, pe care îl avea în clasa `baza`. Nu este posibilă atribuirea gradului de acces intermediar `protected`. (~§7)