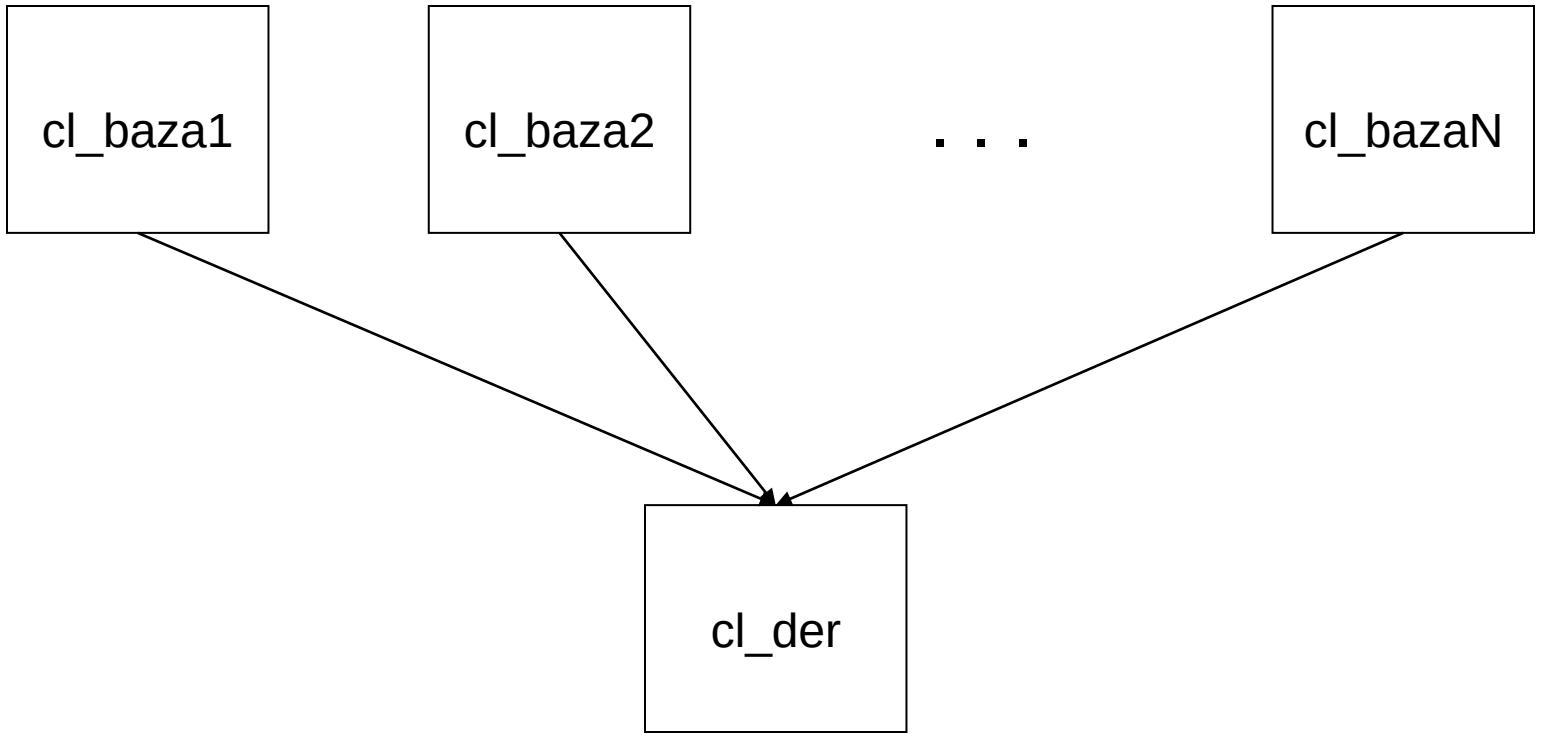
§6. Moştenire multiplă.

Transmiterea de proprietăți şi funcționalități dintr-o serie de clase numite clase de bază într-o clasă numită clasă derivată se numeşte **moştenire multiplă**. Procesul de transmitere se numeşte **derivare**.

Diagrama ce urmează descrie schema generală a moştenirii multiple:



La nivel de descriere în expresii ale limbajului, avem următoarea schemă:

```
class cl_baza1
    {
    .
    <date si functii membre 1>
    .
    };
class cl_baza2
    {
    .
    <date si functii membre 2>
    .
    };
. . .
```

```
class cl_bazaN
     {
     .
     <date si functii membre N>
     .
     };

class cl_der : [mod_m1]cl_baza1, ...,
[mod_mN]cl_bazaN
     {
     .
     <date si functii membre Der>
     .
     };
```

unde `cl_baza1`, `cl_baza2`, ..., `cl_bazaN` sunt clasele de bază, `cl_der` este clasa derivată, `mod_m1`, `mod_m2`, ..., `mod_mN` sunt modificatorii de moştenire care introduc clasele de bază în procesul de derivare. Ca şi în cazul moştenirii singulare modificatori de moştenire pot fi unul dintre cuvintele-cheie: `public`, `protected`, `private`. Dacă o oarecare clasă de bază nu are scris explicit modificator de moştenire, se consideră implicit modificatorul `private`.

Un obiect creat în baza clasei derivate va avea o serie de caracteristici dependente de clasele de bază din care vin, ceea ce este vizibil şi din modul general de descriere a constructorului clasei derivate:

```
cl_der::cl_der(param_der)[:cl_baza1(param_b1)],...[,cl_bazaN(param_bN)]
    {
    .
    //instructiuni
    .
    }
```

Întrucât, la crearea unui obiect în baza unei clase derivate conlucrează mai mulți constructori, va fi evidențiată ordinea de execuție a lor. Mai întâi sunt executați constructorii claselor de bază în ordinea în care apar ei în descrierea clasei derivate. După ce constructorii claselor de bază şi-au terminat lucrul, este executat şi constructorul clasei derivate. Destructorii sunt executați în ordine strict inversă ordinii de execuție a constructorilor.

Pentru a caracteriza interacțiunea dintre constructorii clasei derivate şi constructorii claselor de bază, concretizând astfel descrierea generală a constructorilor clasei derivate vor fi evidențiate următoarele patru situații posibile:

1) Există măcar un constructor în clasa derivată şi constructori măcar în unele clase de bază. În acest caz constructorii claselor de bază pot să lipsească din descrierea constructorului clasei derivate doar când sunt fără parametri sau se reduc la constructori fără parametri.

2) Există măcar un constructor în clasa derivată şi nu există constructori în clasele de bază. În acest caz, fiindcă constructorii impliciți nu fac careva inițializări, ar putea fi deschis accesul spre unii membri ai claselor de bază pentru a putea fi inițializați din constructorul clasei derivate.

3) Nu există constructori în clasa derivată, dar există constructori măcar în unele clase de bază. În acest caz clasele de bază care au constructori trebuie să aibă constructori fără parametri sau constructori care se reduc la constructori fără parametri.

4) Nu există constructori în clasa derivată şi nu există constructori în nici o clasă de bază.

Lista parametrilor fictivi a constructorului clasei derivate adeseori acoperă necesitatea de parametri ai constructorilor claselor de bază, dar aceasta nu este o cerință strictă.

Unii membri ai claselor de bază pot fi supraîncărcați (suprascrişi) în procesul derivării. Existența în clasa derivată a unor membri cu aceleaşi caracteristici ca şi în clasa de bază va fi numită supraîncărcare.

În cazul datelor membre caracteristicile vor însemna:

- numele membrului;
- tipul membrului.

În cazul funcţiilor membre caracteristicile vor însemna prototipul funcţiei membru, adică:

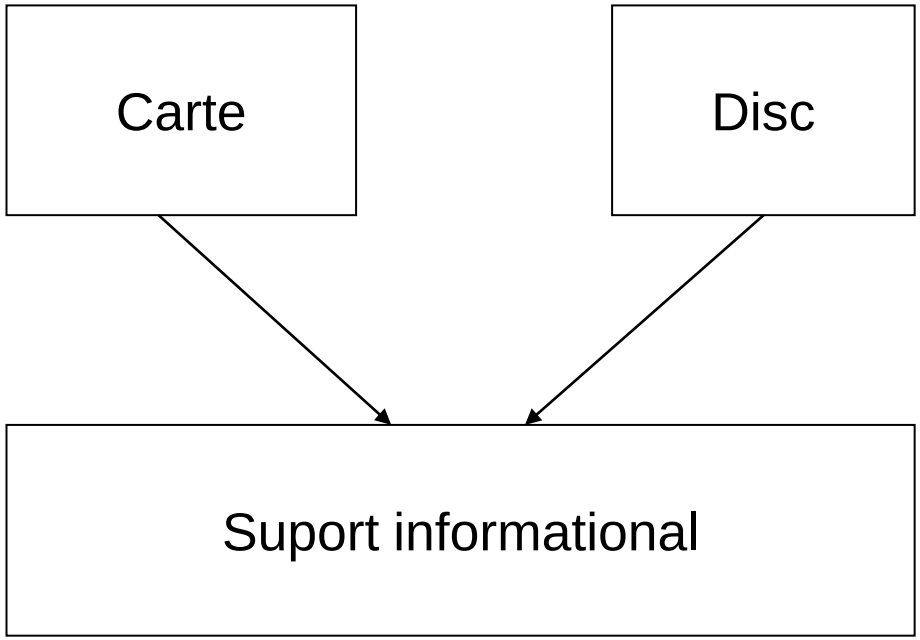
- numele membrului;
- tipul returnat;
- tipul şi numărul parametrilor fictivi ai funcției.

Este posibilă utilizarea membrilor supraîncărcați ai claselor de bază în cadrul funcțiilor membre ale clasei derivate. Pentru a putea accesa un membru supraîncărcat al unei clase de bază, numele lui este prefixat de numele clasei conectat prin operatorul rezoluției:

```
nume_cl_baza::nume_membru
```

O situație similară are loc şi atunci când în, cel puțin, două clase de bază sunt membri cu aceleaşi caracteristici. Pentru a putea accesa careva dintre aceşti membri în cadrul funcțiilor membre ale clasei derivate, numele membrului este, de asemenea, prefixat de numele clasei conectat prin operatorul rezoluției.

Exemplu. În prezent, o serie de cărți sunt dotate cu un disc conținând conținutul cărții în varianta electronică. Exemplul ce urmează se va referi la o astfel de situație. Diagrama de moştenire este următoarea:



Realizare. De alcătuit un program în care sunt implementate clasele `carte`, `disc` şi `supInfo`, care este derivată din clasele `carte` şi `disc`.

```
class carte
{
   char *autor;
   char *titlu;
   int an;
public:
   carte(char *au, char *t, int a);
   carte();
   ~carte();
   void afisare();
};
```

```
class disc
{
    char *tip;
    int capacitate;
 public:
    disc(char *t, int c);
    ~disc();
    void afisare();
};
```

```
class supInfo: public carte, public disc
{
   float pret;
 public:
   supInfo(char *au, char *tt, int a, char *t,
           int c, float p);
   supInfo(char *t, int c, float p);
   void afisare();
};
```

```
carte::carte(char *au, char *t, int a)
 {
  autor=(char *)malloc(strlen(au)+1);
  titlu=(char *)malloc(strlen(t)+1);
  strcpy(autor, au);
  strcpy(titlu, t);
  an=a;
 }
```

```
carte::carte()
 {
   char sir[100];
   cout<<”Autorul: ”;
   cin.getline(sir, 100);
   autor=(char *)malloc(cin.gcount());
   strcpy(autor, sir);
   cout<<”Titlul: ”;
   cin.getline(sir, 100);
   titlu=(char *)malloc(cin.gcount());
   strcpy(titlu, sir);
   cout<<”Anul: ”;
   cin>>an;
 }
```

```
carte::~carte()
  {
    free(autor);
    free(titlu);
  }
//--------------------
void carte::afisare()
  {
    cout<<titlu<<" de "
    <<autor<<" a aparut in "
    <<an<<endl;
  }
```

```
disc::disc(char *t, int c)
 {
   tip=(char *)malloc(strlen(t)+1);
   strcpy(tip, t);
   capacitate=c;
 }
//--------------------
disc::~disc()
 {
   free(tip);
 }
```

```
void disc::afisare()
  {
   cout<<”Are un disc ”<<tipul
       <<” de capacitatea ”
       <<capacitate<<endl;
  }
```

```
supInfo::supInfo(char *au, char *tt, int a,
                 char *t, int c, float p):
                 carte(au,tt,a), disc(t,c)
 {
   pret=p;
 }

//--------------------

supInfo::supInfo(char *t, int c, float p):
                 disc(t,c)
 {
   pret=p;
 }
```

```
void supInfo::afisare()
  {
   carte::afisare();
   disc::afisare();
   cout<<"Si are un pret de "
   <<pret<<" lei"<<endl;
  }
```

Declararea claselor şi implementarea lor va fi memorată în fişierul `supinfo.hpp`. Programul principal în care sunt utilizate obiecte create în baza clasei `supInfo` va fi scris în fişierul `supinfo.cpp`:

```
#include<iostream.h>
#include<string.h>
#include<stdlib.h>
#include ”supinfo.hpp”
void main()
{
  supInfo cpp(”Kris Jansa”, ”Succes cu C++”,
              2000, ”CD-ROM”, 700, 200);
  supInfo pr(”CD-ROM”,700,300);
  cpp.afisare();
  pr.afisare();
}
```

(~§6)