12

CREAREA IERARHIILOR DE CLASE

12.1. MECANISMUL MOŞTENIRII

Moştenirea este o caracteristică a limbajelor de programare orientate obiect, care permite refolosirea codului şi extinderea funcționalității claselor existente (vezi capitolul 9). Mecanismul moştenirii permite crearea unei ierarhii de clase şi trecerea de la clasele generale la cele particulare. (Un concept poate fi implementat printr-o clasă). Această proprietate se manifestă prin faptul că din orice clasă putem deriva alte clase. Procesul implică la început definirea clasei de bază care stabileşte calitățile comune ale tuturor obiectelor ce vor deriva din bază (ierarhic superioară). Prin moştenire, un obiect poate prelua proprietățile obiectelor din clasa de bază.
Moştenirea poate fi:

- Unică (o clasă are doar o superclasă, rezultând o structură arborescentă);
- Multiplă (o clasă are mai multe superclase, rezultând o structură de rețea).

Informația comună apare în clasa de bază, iar informația specifică - în clasa derivată. Clasa derivată reprezintă o specializare a clasei de bază. Orice clasă derivată ***moşteneşte*** datele membru şi metodele clasei de bază. Deci acestea ***nu*** trebuie redeclarate în clasa derivată.

În limbajul C++ încapsularea poate fi forțată prin controlul accesului, deoarece toate datele şi funcțiile membre sunt caracterizate printr-un **nivel de acces.** Nivelul de acces la membrii unei clase poate fi:

- `private`: membrii (date şi metode) la care accesul este private pot fi accesați doar prin metodele clasei (nivel acces implicit);
- `protected`: aceşti membri pot fi accesați prin funcțiile membre ale clasei şi funcțiile membre ale clasei derivate;
- `public:` membrii la care accesul este public pot fi accesați din orice punct al domeniului de existență a clasei respective;
- `friend:` aceşti membri pot fi accesați prin funcțiile membre ale funcției prietene specificate.

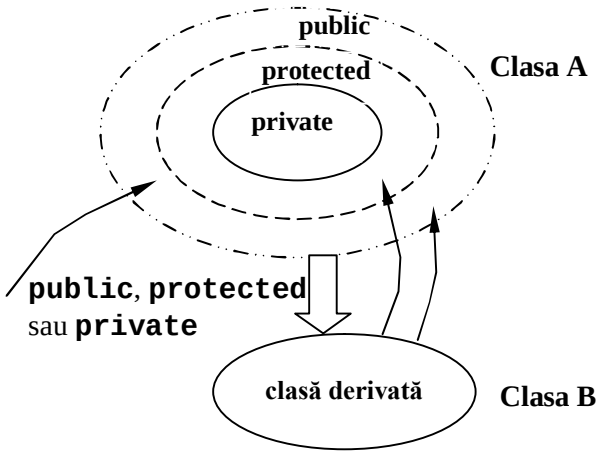

Figura 12.1. Accesul la membrii unei clase. Moştenirea `publică, protejată` sau `privată`

În limbajul C++, nivelul de acces poate preciza şi tipul de moştenire (figura 12.1.):

- Publică, unde în clasa derivată nivelul de acces al membrilor este acelaşi ca în clasa de bază;
- Privată, unde membrii protected şi public din clasa bază devin private în clasa derivată;
- Protejată (la compilatoarele mai noi).

Când o clasă moşteneşte membrii unei alte clase, membrii clasei de bază devin membrii ai clasei derivate. Moştenirea protejată este intermediară celei publice şi celei private. În cazul moştenirii protejate, comparativ cu moştenire privată, singura diferenţă este că membrii publici ai clasei de bază devin protejaţi în timpul derivărilor ulterioare. În funcţie de modificatorii de acces la membrii clasei de bază, la membrii clasei derivate şi de tipul moştenirii, lucrurile se pot rezuma astfel (tabelul 12.1.):

Tabelul 12.1.

Modificator acces la membrii clasei de bază	**Accesul în clasa derivată (noul acces) dobândit prin moştenire *publică***	**Accesul în clasa derivată (noul acces) dobândit prin moştenire *protejată***	**Accesul în clasa derivată (noul acces) dobândit prin moştenire *privată***
private	private	private	private
public	public	protected	private
protected	protected	protected	private

Aşa cum se observă, în toate cazurile, elementele private ale clasei de bază rămân particulare acesteia şi nu sunt accesibile claselor derivate; cele protejate sunt accesibile clasei derivate.

12.2. MODUL DE DECLARARE A CLASELOR DERIVATE

La modul general, la declararea unei clase derivate, se specifică o listă a claselor de bază, precedate de modificatorul de acces care precizează tipul moştenirii.

```
class <nume_cls_deriv>: <modificator_de_acces> <nume_clasă_de_bază>
{       //corpul clasei derivate - elemente specifice clasei derivate
};
```

Exemplu: Declararea clasei derivate `angajat`, cu clasa de bază `persoana` (moştenire simplă):

```
class persoana{
      // corpul clasei de bază
};
class angajat: protected persoana{
      double salariu;
};
```

Exemplu: Declararea clasei derivate `interfaţă`, cu clasele de bază `fereastră` şi `meniu` (moştenire multiplă):

```
class fereastra{
      //membrii clasei
};
class meniu{
      //membrii clasei
};
class interfata: public fereastra, public meniu{
      //membrii clasei
};
```

În ceea ce priveşte folosirea (compilarea şi editarea de legături) clasei derivate în sensul programării, clasa de bază şi cea derivată pot apare în acelaţi fisier sursă, sau declarate în fişiere diferite (figura 12.1.).

12.3. CONSTRUCTORII CLASELOR DERIVATE

Constructorii şi destructorii sunt funcţii membre care **nu** se moştenesc. La instanţierea unui obiect din clasa derivată se apelează mai intâi constructorii claselor de bază, în ordinea în care aceştia apar în lista din declararea clasei derivate. La distrugerea obiectelor, se apelează întâi destructorul clasei derivate, apoi destructorii claselor de bază.

Transmiterea argumentelor unei funcţii constructor din clasa de bază se face folosind ***o formă extinsă a declaraţiei constructorului clasei derivate,*** care transmite argumentele unui sau mai multor constructori din clasa de bază.

În general, clasele utilizează constructori definiţi de programator. În cazul în care aceştia lipsesc, compilatorul generează automat un constructor implicit pentru clasa respectivă. Acelaşi lucru se întâmplă şi în cazul constructorilor de copiere.

La instanţierea unui obiect din clasă derivată, o parte din valorile primite ca parametri folosesc la iniţializarea datelor membru ale claselor de bază, iar restul iniţializează datele membru specifice clasei derivate.

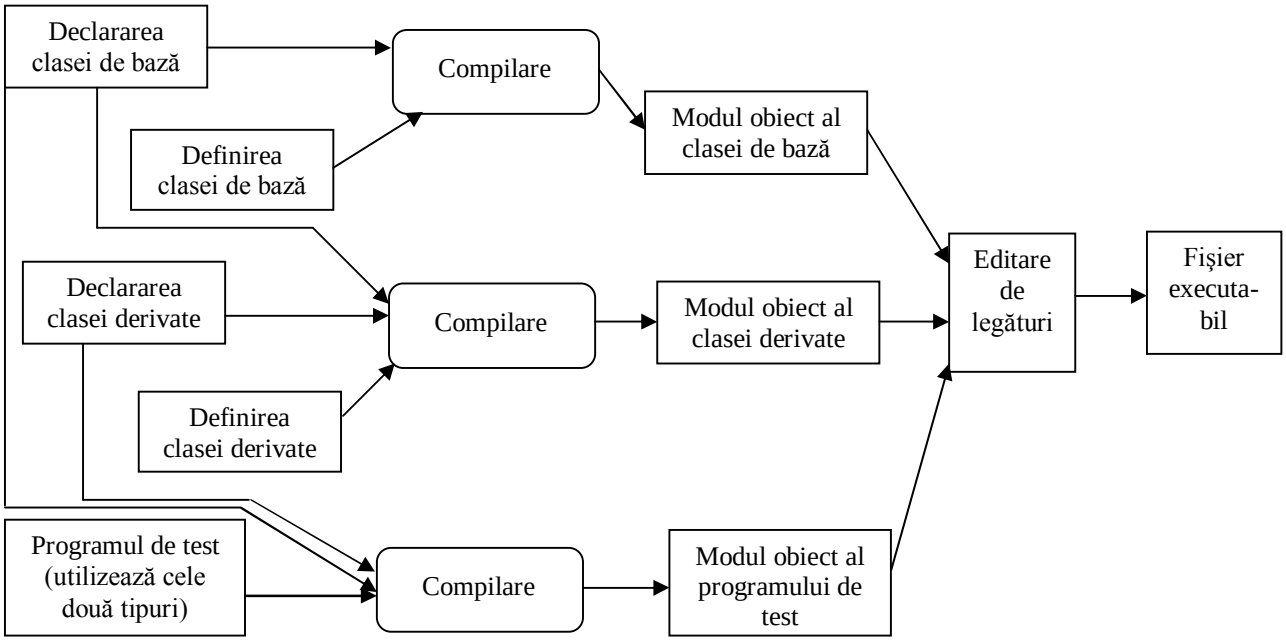

Figura 12.1. Editarea de legături la utilizarea clasei derivate

12.4. MOŞTENIREA SIMPLĂ

Pentru a evidenţia aspectele prezentate, să considerăm următoarele exemple, în care moştenirea este simplă:

Exemplu:

Se construieşte ierarhia de clase din figura 12.2.:

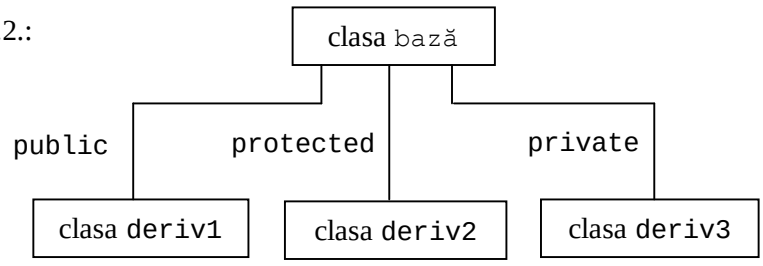

Figura 12.2. Ierarhie de clase

```
#include <iostream.h>
class bază
{
        int a;
protected:
        double w;
        void setează_a(int a1){a=a1;}
        void setează_w(int w1){w=w1;}
public:
        int c;
        baza (int a1, double w1, int c1)
              {a=a1; w=w1; c=c1;cout<<"Constructor cls. bază\n";}
        ~bază()
              {cout<<"Destructor bază\n";}
        void arată()
              {cout<<a<<' '<<w<<' '<<c<<'\n';}
        double calcul()
```

```
            {return a+w+c;}
        friend ostream & operator<<(ostream &, const bază &);
};

class deriv1: public bază
{
      int b;
public:
      deriv1 (int a1, double w1, int c1, int b1):bază(a1, w1, c1)
            {b=b1; cout<<"Constructor deriv1\n";}
      ~deriv1()
            {cout<<"Destructor deriv1\n";}
      double calcul()
            {return w+c+b;}     // membrul a este încapsulat, nu poate fi folosit, fiind private
                  // o alternativă pentru obținerea sumei tuturor datelor membre este:
                  // double calcul(){return bază::calcul()+b;}
      friend ostream &operator<<(ostream &, const deriv1 &);
};

class deriv2: protected bază
{
        int b;
public:
        deriv2(int a1, double w1, int c1, int b1):bază(a1, w1, c1)
            {b=b1; cout<<"Constructor deriv2\n";}
        ~deriv2()
            {cout<<"Destructor deriv2\n";}
        double calcul()
            {return w+c+b;}
        friend ostream &operator<<(ostream &, const deriv2 &);
};

class deriv3: private bază
{
        int b;
public:
        deriv3(int a1, double w1, int c1, int b1):baza(a1, w1, c1)
            {b=b1; cout<<"Constructor deriv3\n";}
        ~deriv3()
            {cout<<"Destructor deriv3\n";}
        double calcul()
            {return w+c+b;}
        friend ostream &operator<<(ostream &, const deriv3 &);
};

ostream &operator<<(ostream &ies, const baza &b)
      {ies<<b.a<<' '<<b.w<<' '<<b.c<<'\n'; return ies;}
ostream &operator<<(ostream &ies, const deriv1& d1)
      {ies<<d1.w<<' '<<d1.c<<' '<<d1.b<<'\n'; // a private
      return ies;}
ostream &operator<<(ostream &ies, const deriv2& d2)
      {ies<<d2.w<<' '<<d2.c<<' '<<d2.b<<'\n'; // a private
      return ies;}
ostream &operator<<(ostream &ies, const deriv3& d3)
      {ies<<d3.w<<' '<<d3.c<<' '<<d3.b<<'\n'; // a private
      return ies;}

void main()
{
      baza x(1, 1.23, 2);           // Constructor cls. baza
      deriv1 y(2, 2.34, 3, 4);      // Constructor cls. baza        Constructor deriv1
```

```
      deriv2 z(3, 3.45, 4, 5);        // Constructor cls. baza        Constructor deriv2
      deriv3 v(4, 5.67, 6, 7);        //Constructor cls. baza        Constructor deriv3
cout<<"x="<<x<<'\n'<<"z="<<z<<'\n'<<"v="<<v<<'\n';
      // x=1   1.23    2 (x.a, x.w, x.c)
      // z=3.45  4  5
      // v=5.67  6  7
cout<<"x.calcul()="<<x.calcul()<<'\n';      // x.calcul()=4.23
cout<<"y.calcul()="<<y.calcul()<<'\n';      // y.calcul()=9.34
cout<<"z.calcul()="<<z.calcul()<<'\n';      // z.calcul()=12.45

cout<<"v.calcul()="<<v.calcul()<<'\n';      // v.calcul()=18.67
cout<<"x.c="<<x.c<<'\n';                    // x.c=2
cout<<"y.c="<<y.c<<'\n';                    // y.c=3
      /*      Destructor deriv3      Destructor baza      ptr. v
              Destructor deriv2      Destructor baza      ptr. z
              Destructor deriv1      Destructor baza      ptr. y
              Destructor baza                             ptr x    */
}
```

Observaţii:

În clasa de bază data membră `a` este `private`, `w` este `protected` şi `c` este `public`. În clasa de bază, cât şi în clasele derivate există constructori care iniţializează datele membru. Membrii `private` dintr-o clasă de bază (clasa `bază`, în cazul nostru) pot fi folosiţi doar în cadrul acesteia (de metodele sale), nu şi în clasele derivate.

Clasa `deriv1`:

Membrii privaţi moşteniţi din clasa `bază` sunt inaccesibili (`a` există, dar este încapsulat). Pentru a putea fi accesaţi, se folosesc metodele clasei `bază` (metoda `calcul`). Deoarece în clasa `deriv1` există o metodă cu acelaşi nume cu al unei metode din clasa de bază (redefinirea unei metode în clasa derivată), se foloseşte operatorul de rezoluţie.

`baza::calcul( )` sau `y.baza::calcul( )`

Clasa `deriv2`:

Deoarece moştenirea este protejată, membrii publici sau protejaţi din clasa `bază` devin protejaţi în clasa `deriv2`. De aceea, dacă în funcţia `main` am încerca folosirea :

`cout<<z.baza::calcul( )` , metoda `calcul` inaccesibilă, ea devenind protejată în clasa *deriv3*.

Clasa `deriv3`:

Deoarece moştenirea este privată, membrii public sau protected din clasa `bază` au devenit privaţi în clasa `deriv3`. Se pot folosi toţi membrii clasei de bază, cu excepţia celor privaţi (a).

La construirea unui obiect dintr-o clasă derivată din clasa `bază`, se apelează mai întîi constructorul din clasa de bază, apoi constructorul clasei derivate Astfel, un obiect `y` din clasa `deriv2` incorporează un obiect deja iniţializat cu ajutorul constructorului din clasa `bază`.

Dacă pentru clasa `deriv1` defineam un constructor de forma:

```
deriv1(int a1, double b1, int c1, int b1){a=a1; b=b1; c=c1; d=d1;}
```

nu era corect, deoarece clasa `bază` nu are constructori fără parametri, deci nu există constructor implicit, iar data `a` este inaccesibilă în `deriv1`. Apelarea constructorului se realizează apelând explicit constructorul din clasa `bază`.

Exerciţiu: Fie clasa punct şi clasa punct colorat, derivată din clasa punct. Metoda `afisare` este redefinită în clasa derivată (`punct_col`).

```
#include <iostream.h>
#include <conio.h>
class punct{
      int x, y;     //date membru private, inaccesibile în clasa punct_col
```

```
public:
      punct (int abs=0, int ord=0)
            {x=abs; y=ord; cout<<"Constr punct "<<x<<","<<y<<'\n';}
      punct (const punct& p)
            {x=p.x; y=p.y; cout<<"Constr copiere punct ";
            cout<<x<<","<<y<<"\n";}
      ~punct()
            {cout<<"Destr punct "<<x<<","<<y<<"\n";}
      void afisare()
            {cout<<"P("<<x<<","<<y<<")\n";}
};

class punct_col:public punct{
      short cul;   //date membru private
public:
      punct_col (int, int, short);
      punct_col (const punct_col & p):punct (p)
            {cul=p.cul; cout<<"Constr copiere punct col "<<cul<<'\n';}
      ~punct_col()
            {cout<<"Destr punct colorat "<<cul<<"\n";}
      void afisare()
            {cout<<"-----------------\n";
            cout<<"Punct colorat:";punct::afisare();
            cout<<"Culoare:"<<cul<<"\n-------------------\n";}
};

punct_col::punct_col(int abs=0, int ord=0, short cl=1):punct(abs, ord)
      {cul=cl; cout<<"Constr punct colorat "<<"culoare="<<cul<<'\n';}

void main()
{clrscr();
punct_col A(10, 15, 3); //Constr punct 10,15     Constr punct colorat culoare=3
punct_col B(2,3);       //Constr punct 2,3       Constr punct colorat culoare=1
punct_col C(12);        //Constr punct 12,0      Constr punct colorat culoare=1
punct_col D;            // Constr punct 0,0      Constr punct colorat culoare=1
D.afisare();
      /*-----------------          apelul metodei afisare a clasei punct_col
      Punct colorat:P(0,0)
      Culoare:1
      ------------------ */
D.punct::afisare();           // P(0,0) apelul metodei afisare a clasei punct
punct_col *pp;
pp=new punct_col(12,25);      //Constr punct 12,25     Constr punct colorat culoare=1
      // obiect dinamic; se apeleaza constructorii
delete pp;                    //Destr punct colorat 1  Destr punct 12,25
      // eliberare memorie
punct P1;                     //Constr punct 0,0
punct P2=P1;                  //Constr copiere punct 0,0
punct_col C1=C;               //Constr copiere punct 12,0      Constr copiere punct col 1
}
      //Destr punct colorat 1  Destr punct 12,0 (pentru C1)
      //Destr punct 0,0        (pentru P1)
      //Destr punct 0,0        (pentru P2)
      //Destr punct colorat 1  Destr punct 0,0  (pentru D)
      //Destr punct colorat 1  Destr punct 12,0 (pentru C)
      //Destr punct colorat 1  Destr punct 2,3  (pentru B)
      //Destr punct colorat 3  Destr punct 10,15 (pentru A)
```

:

Exerciţiu: Se implementează ierahia de clase din figura 12.3.
Clasele `persoana`, `student`, `student_bursier` au ca date membre date de tipul `şir` (implementat în capitolul 11).

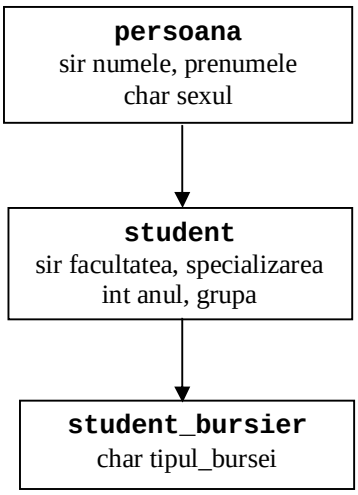

Figura 12.3.

```
#include "sir.cpp"
#include <conio.h>
#include <iostream.h>
class persoană {
protected:
      şir numele,prenumele;
      char sexul;
public:
      persoana ()                    //constructor vid
            {numele="";prenumele="";sexul='m';
            cout<<"Constr PERS vid!\n";}
      persoană(const şir&,const şir&,const char);
                                     //constructor
      persoană (const persoana&);    //constr. copiere
      virtual ~persoană();           //destructor
      const şir& nume()const;
      const şir&prenume() const;
      char sex() const;
      virtual void afişare();
      friend  ostream & operator<<(ostream &, const persoana &);
      friend istream & operator>>(istream &,  persoana &);
};

class student:public persoană {
protected:
      şir facultatea,specializarea;
      int anul,grupa;
public:
student(const şir&,const şir&,const char,const şir&,const şir&,const int,const
int);
student(const persoana&,const şir&,const şir&,const int,const int);
student(const student&);
virtual ~student();
const şir& facult(){return facultatea;}
const şir& spec(){return specializarea;}
int an(){return anul;}
int grup(){return grupa;}
virtual void afişare();
friend  ostream & operator<<(ostream &, const student &);
/* TEMA
friend istream & operator>>(istream &,  student &);*/
};

class student_bursier:public student {
protected:
      char tipul_bursei;
public:
      student_bursier(const student&,char);
      student_bursier(const student_bursier&);
      virtual ~student_bursier();
      char tip_bursa() {return tipul_bursei;}
      double valoare_bursa();
      virtual void afişare();
      //TEMA friend  ostream & operator<<(ostream &, const student_bursier &);
      //TEMA friend istream & operator>>(istream &,  student_bursier &);
};
```

// **METODELE CLASEI PERSOANA**

```
persoană::persoană(const şir& nume,const şir& prenume,const char sex)
      {numele=nume;prenumele=prenume;sexul=sex;
      cout<<"Constr. PERSOANĂ\n";}
persoana::persoana(const persoana& pers)
      { numele=pers.numele;prenumele=pers.prenumele;sexul=pers.sexul;
      cout<<"Constructor copiere PERSOANA\n";}
persoană::~persoană()
      {cout<<"Destructor PERSOANĂ\n";}
const şir& persoană::nume()const
      {return numele;}
const şir& persoană::prenume()const
      {return prenumele;}
char persoană::sex()const
      {return sexul;}
void persoană::afişare()
      { cout<<"Afişare PERSOANĂ:\n";
      cout<<numele<<"   "<<prenumele<<"   ";
      if (toupper (sexul)=='M')     cout<<"SEX barbatesc\n";
      else  cout<<"SEX femeiesc\n";}
ostream & operator<<(ostream &monitor, const persoana &p)
      {monitor<<"\nNume pers:"<<p.numele<<"\nPrenume pers:"<<p.prenumele;
      cout<<"\nSex pers:";
      if (p.sexul=='m')       cout<<"BARBATESC";
      else                    cout<<"FEMEIESC";
      return monitor;}
istream & operator>>(istream & tastat,  persoana &p)
      {tastat>>p.numele>>p.prenumele>>p.sexul;  return tastat;}
```

// **METODELE CLASEI STUDENT**

```
student::student(const şir&nume,const şir&prenume,const char sex,const şir&
facult,const şir& spec,const int an,const int gr):persoana(nume,prenume,sex)
      {numele=nume;prenumele=prenume;
      sexul=sex;facultatea=facult; specializarea=spec; anul=an; grupa=gr;
      cout<<"Construct STUD 1\n"; }
student::student(const persoana &pers,const şir& facult,const şir& spec,const
int an,const int gr):persoana(pers)
      { numele=pers.nume();prenumele=pers.prenume();
      facultatea=facult; specializarea=spec;anul=an;grupa=gr;
      cout<<"Construct STUD 2\n";}
student::student(const student&
stud):persoana(stud.numele,stud.prenumele,stud.sexul)
      { facultatea=stud.facultatea; specializarea=stud.specializarea;
       anul=stud.anul; grupa=stud.grupa;cout<<"Construct copiere STUD!\n"; }
student::~student()
      { cout<<"Destructor student!!\n"; }
void student::afişare()
      { cout<<numele<<"   "<<prenumele<<'\n';cout<<"Sex:"<<sexul<<'\n';
      cout<<"Facultatea: "<<facultatea<<" Specializare: "<<specializarea<<'\n';
      cout<<"Anul: "<<anul<<" Grupa:"<<grupa<<'\n';   }
ostream & operator<<(ostream &monitor, const student &s)
      {monitor<<"\nNume stud:"<<s.numele<<"\nPrenume stud:"<<s.prenumele;
      cout<<"\nSex stud:"<<((s.sexul=='m')?"BARBATESC":"FEMEIESC");
      monitor<<"\nFacultate :"<<s.facultatea<<;
      cout<<"\nSpecializare :"<<s.specializarea;
      monitor<<"\nAnul :"<<s.anul<<"\nGrupa :"<<s.grupa;
      return monitor;}
//TEMA  friend istream & operator>>(istream &,  student &);
```

//**METODE CLASEI STUDENT_BURSIER**

```
/*  TEMA
student_bursier(student&,char);
```

```
  student_bursier(const student_bursier&);*/

student_bursier::student_bursier(const student&stud,char tip_burs):student(stud)
      {tipul_bursei=tip_burs;}
student_bursier::student_bursier(const student_bursier
&stud):student(stud.numele,stud.prenumele,stud.sexul,stud.facultatea,stud.specia
lizarea,stud.anul,stud.grupa)
      {tipul_bursei=stud.tipul_bursei;}
double student_bursier::valoare_bursa()
      { double val;
        switch (tipul_bursei)
              { case 'A':     val=850000; break;
                case 'B':     val=700000; break;
            }
      return val;
      }
student_bursier::~student_bursier()
      {cout<<"Desctructor student bursier\n";}
void student_bursier::afişare()
      { student::afişare();
      cout<<"Tip bursa: "<<tipul_bursei<<" Valoare: "<<valoare_bursa()<<'\n';}

void main()
      {clrscr();persoana x("POP","ION",'m');          //Constructor PERSOANA
      x.afisare();cout<<'\n';                         // POP        ION        m
      cout<<"Apasa tasta...\n";getch();
      persoana x1(x);                                 //Constructor copiere PERSOANA
      cout<<x1<<'\n';          //Nume pers: POP Prenume pers: ION  Sex pers: BARBATESC
      cout<<"Apasa tasta...\n";getch();
      cout<<"Introduceti inf. despre persoana:\n";
      persoana x2;                                    //Constr PERS vid!
      cin>>x2;
      cout<"Inf introduse:\n";
      cout<<x2;
      cout<<"Apasa tasta...\n";getch();
            //x1.afisare(); cout<<'\n';
      student s(x, "N.I.E.", "EA", 1, 2311);
            //Constructor copiere PERSOANA      Construct STUD 2!
      s.afisare();cout<<'\n';
            /* POP   ION    Sex: m  Facultatea: N.I.E. Specializare: EA  Anul: 1 Grupa:2311 */
      cout<<"Apasa tasta...\n";getch();
      student s1(s);                          //Constr. PERSOANA Construct copiere STUD!
      cout<<s1<<'\n';
            /* Nume stud:POP       Prenume stud:ION       Sex stud:BARBATESC Facultate :N.I.E.
            Specializare :EA        Anul :1 Grupa :2311*/
      cout<<"Apasa tasta...\n";getch();
      student s3("STAN", "POPICA", 'm', "MECANICA", "I.M.T.", 1, 320);
      //Constr. PERSOANA   Construct STUD 1!
      cout<<s1<<'\n';    /* Nume stud:POP        Prenume stud:ION      Sex stud:BARBATESC
      Facultate :N.I.E.        Specializare :EA        Anul :1 Grupa :2311 */
      s3=s1;
      cout<<"In urma atribuirii s3="<<s3<<'\n';
      /* In urma atribuirii s3= Nume stud:POPPrenume stud:ION       Sex stud:BARBATESC
      Facultate :N.I.E.        Specializare :EA        Anul :1 Grupa :2311 */
      s3.afisare( );
      }
```

Observaţii:

1. Să se completeze exemplul cu funcțiile date ca temă. Să se completeze programul de test (funcția `main`).

2. Funcția `afişare` este declarată `virtuală` în clasa de bază şi redefinită în clasa derivată. Redefinirea funcției în clasa derivată are prioritate față de definirea funcției din clasa de bază. Astfel, o funcție virtuală declarată în clasa de bază acționează ca un substitut pentru păstrarea datelor care specifică o clasă generală de acțiuni şi declară forma interfeței. Funcția afişare are acelaşi prototip pentru toate clasele în care a fost redefinită (vezi paragraful 12.7.).

12.5. MOŞTENIREA MULTIPLĂ

O clasă poate să moştenească mai multe clase de bază, ceea ce înseamnă că toți membrii claselor de bază vor fi moşteniți de clasa derivată. În această situație apare mecanismul moştenirii multiple. În paragraful 12.2. a fost prezentat modul de declarare a unei clase cu mai multe superclase.

Exercițiu: Se implementează ierahia de clase din figura 12.4.

```
#include <iostream.h>
class bază1 {
protected:
      int x;
public:
      bază1 (int xx)
            {x=xx;cout<<"Constructor cls. bază1\n";
            cout<<x<<'\n';}
      ~baza1()
            {cout<<"Destructor bază1\n"<<x<<'\n'
      void aratax()
            {cout<<"x="<<x<<'\n';}
};

class bază2 {
protected:
      int y;
public:
      bază2 (int yy)
            {y=yy; cout<<"Constructor bază2\n"<<y<<'\n';}
      ~bază2()
            {cout<<"Destructor bază2\n";}
      void aratay(){cout<<"y="<<y<<'\n';}
};

class derivat: public bază1, public bază2 {
public:
      derivat(int xx, int yy):bază1(xx), bază2(yy)
            {cout<<"Constructor derivat\n"; cout<<x<<' '<<y<<'\n';}
      ~derivat()
            {cout<<"Destructor derivat\n"; cout<<x<<' '<<y<<'\n';}
      int arata(){cout<<x<<' '<<y<<'\n';}
      void seteaza(int xx, int yy){x=xx; y=yy;}
};

void main()
{
derivat obiect(7,8);/*Constructor cls. bază1  7    Constructor bază2  8   Constructor derivat  7  8   */
obiect.arata();          // 7 8
obiect.seteaza(1,2);
obiect.aratax();         // x=1
obiect.aratay();         // y=2
obiect.arata();          // 1 2
      /* Destructor derivat 1 2          Destructor bază2 2        Destructor bază1  1 */
}
```

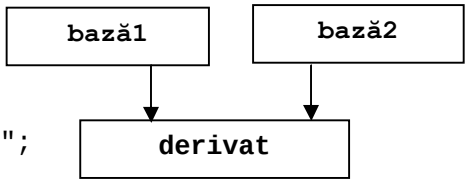

Figura 12.4. Schemă de moştenire multiplă

Aşa cum ilustrează exemplul, la declararea obiectului `obiect` de tipul `derivat` s-au apelat constructorii claselor de bază (`bază1` şi `bază2`), în ordinea în care apar în declararea clasei derivate: mai întâi constructorul clasei `bază1`, în care `x` este dată membru protejată (accesibilă din clasa derivat); apoi constructorul clasei `bază2` , în care `y` este dată membru protejată (accesibilă din clasa derivat); apoi constructorul clasei `derivat` care le încorporează pe acestea într-un singur obiect. Clasa derivat nu are date membre, ci doar metode (figura 12.5.).

După ieşirea din blocul în care a fost declarată variabila `obiect`, se apelează automat destructorii, în ordine inversă apelării constructorilor.

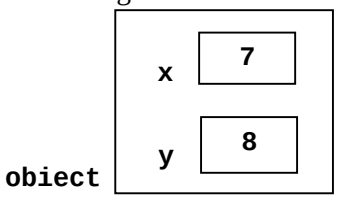

Figura 12.5. Variabila `obiect` de tip `derivat`

12.6. REDEFINIREA MEMBRILOR UNEI CLASE DE BAZĂ ÎN CLASA DERIVATĂ

Aşa cum s-a observat deja din exerciţiul anterior, unii membrii (fie date membru, fie metode) ai unei clase de bază pot fi redefiniţi în clasele derivate din aceasta.

Exemplu în care se redefinesc datele membre ale clasei de bază în clasa derivată

```
class bază{
protected:
      double x, y;
public:
      bază(double xx=0, double yy=0)      {x=xx; y=yy;}
};

class deriv:public bază{
protected:
      double x, y;
public:
      deriv(double dx=0, double dy=0, double bx=0, double by=0): baza (bx, by)
            {x=dx;       // x - membru redefinit în clasa derivată
            y=dy;        // y - membru redefinit în clasa derivată
            }
      void arată() const;
};
void deriv::arată() const
      {cout<<"x din clasă de bază:"<<bază::x;
      cout<<"\ty din clasa de bază:"<<bază::y<<'\n';
      cout<<"x din clasa derivată:"<<x;cout<<"\ty din clasa derivată:"<<y<<'\n';
      }
```

În metoda `arată` a clasei `deriv`, pentru a face distincţie între datele membru ale clasei `deriv` şi cele ale clasei `bază`, se foloseşte operatorul de rezoluţie.

Dacă ne întoarcem la exemplul în care implementam ierarhia de clase `persoana`, `student`, `student_bursier`, remarcăm faptul că metoda `afisare` din clasa `persoana` supraîncărcată în clasele derivate `student` şi `student_bursier`. Redefinirea unei metode a unei clase de bază într-o clasă derivată se numeşte **polimorfism.**

Fie schema de moştenire prezentată în figura 12.6.
La declararea unui obiect din clasa `D`, membrii clasei `A` (int `a`) sunt moşteniţi de obiectul din clasa `D` în dublu exemplar (figura 12.7.). Pentru a evita această situaţie, **clasa** `A` va fi declarată **virtuală**, pentru clasele derivate `B` şi `C`. Metoda `arată` este redefinită în clasele `B`,`C`, `D` (polimorfism) (vezi exerciţiul următor şi figura 12.8.).

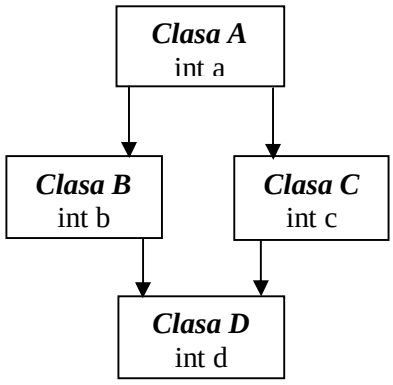

Figura 12.6.

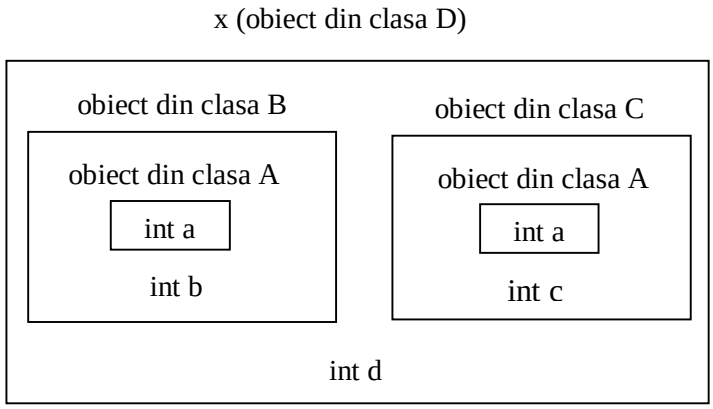

Figura 12.7.

Exercițiu:

```
#include <iostream.h>
class A{
      int a;
public:
      A(int aa)
            {a=aa; cout<<"Constructor A"<<a<<'\n';}
      ~A()
            {cout<<”Destructor A”;}
      void arată()
            {cout<<"A.a="<<a<<'\n';}
};

class B: virtual public A{
      int b;
public:
      B(int bb, int aa=0):A(aa)
            {b=bb;cout<<"Constructor B"<<b<<'\n';}
      ~B()
            {cout<<”Destructor B”;}
      void arata()
            {cout<<"B.b="<<b<<'\n';}
};

class C: virtual public A{
      int c;
public:
      C (int cc, int aa=0):A(aa)
            {c=cc; cout<<"Constructor C"<<c<<'\n';}
      ~C()
            {cout<<”Destructor C”;}
      void arata()
        {cout<<"C.c="<<c<<'\n';}
};

class D: public B, public C{
      int d;
public:
      D(int aa, int bb, int cc, int dd):A(aa), B(bb), C(cc)
            {d=dd;cout<<"Constructor D"<<d<<'\n';}
      ~D()
            {cout<<”Destructor D”;}
```

```
      void arată()
            {cout<<"D.d="<<d<<'\n';}
};
void main()
{
D x(1,2,3,4); /* Constructor A1 Constructor B2 Constructor C3 Constructor D4 */
x.arată();  // apelul metodei arată din clasa D, pentru obiectul x      D.d=4
x.B::arată();       // apelul metodei arată din clasa B       B.b=2
x.C::arată();       // apelul metodei arată din clasa C        C.c=3
x.A::arată();       // apelul metodei arată din clasa A       A.a=1
}      /* Destructor D          Destructor C          Destructor B          Destructor A    */
```

x - obiect din clasa D

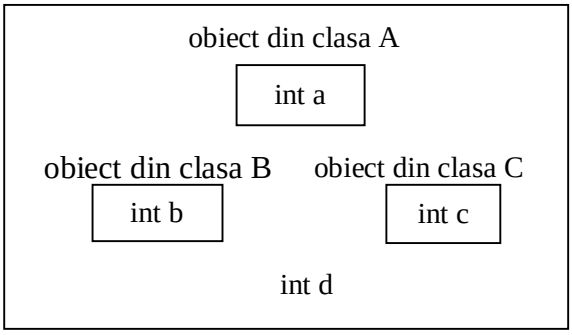

Figura 12.8. Clasa A, **clasă virtuală**

Exerciţiu: Fie următorul program de test pentru ierarhia de clase din figura 12.6., în care `A` este ***clasă virtuală***.

```
void main()
{
A u(10);            // Constructor A  10                          - ptr. obiectul u, de tp A
B v1(9, 7);         // Constructor A  9     Constructor B  7      - ptr. obiectul v1, de tip B
C v2(8,12);         // Constructor A  8     Constructor C  12      - ptr. obiectul v2, de tip C
D w(31, 9, 14, 35);// Constructor A  31    Constructor B  9      Constructor C  14
                    // ConstructorD  35                           - ptr. obiectul w, de tip D
// Se apelează metoda arată, pentru obiectul curent
u.arată();          // A.a=10
v1.arată();         // B.b=7
v2.arată();         // C.c=12
w.arată();          // D.d=35
}
/* Destructor D     Destructor C   Destructor B   Destructor A   - ptr. obiectul  w
Destructor C        Destructor A                                 - ptr. obiectul v2
Destructor B        Destructor A                                 - ptr. obiectul v1
Destructor A                                                     - ptr. obectul  u */
```

Aşa cum se observă din exemplu, metoda **`arată`** din ***clasa de bază*** **`A`** a fost ***redefinită*** în ***clasele derivate*** **`B, C, D`**. În plus, metoda are ***aceeaşi semnătură*** în toate clasele. Dacă nu ar fi fost redefinită în clasele derivate, metoda `arată` (publică) din clasa de bază A ar fi fost moştenită de clasele derivate; redefinirea a fost necesară pentru a putea vizualiza şi datele membre proprii claselor derivate. În cazul de față, identificarea metodei apelate se realizează chiar în etapa compilării, datorită legăturii cu obiectul pentru care a fost apelată. De exemplu, la apelul **`w.arată()`** se aplică metoda din clasa `D` (obiectul `w` este de tip `D`).

Concluzionând, identificarea unei metode din clasa de bază redefinite în clasele derivate, se face prin una din modalitățile:

- Diferențele de semnătură ale metodei redefinite;

- Prin legătura cu obiectul asupra căruia se aplică metoda (vezi apelurile metodei `arată` pentru obiectele `u`, `v1`, `v2`, `w`);
- Prezența operatorului de rezoluție (de exemplu, dacă pentru obiectul w se doreşte apelarea metodei arată din clasa B, apelul va acea forma: `w.A::arată();`).

Un pointer către o clasă de bază poate primi ca valoare adresa unui obiect dintr-o clasă derivată (figura 12.9.). În această situație, se apelează ***metoda din clasa pointerilor***, şi ***nu din clasa obiectului spre care pointează pointerul***.
Pentru exemplificare, vom considera ierarhia de clase (A, B, C, D) ulterioară, şi programul de test:

```
void main()
{
A u(10), *PA;       // Constructor A  10
B v1(9, 7), *PB;    // Constructor A  9      Constructor B  7          - ptr. v1
C v2(8,12), *PC;    // Constructor A  8      Constructor C 12          - ptr. v2
D w(31, 9, 14, 35), *PD;        // Constructor A  31     Constructor B  9
                                // Constructor C  14     Constructor D  35
PA=&u; PA->arată();             // Se selectează metoda arată din clasa A A.a=10
PA=&v1; PA->arată();            // Se selectează metoda arată din clasa A A.a=9
PB=&v1; PB->arată();            // Se selectează metoda arată din clasa B B.b=7
PA=&w; PB=&w; PD=&w;
u.arată();              // Apelul metodei arată ptr. obiectul  curent, clasa A    A.a=31
PA->arată();            // Se selectează metoda arată din clasa A          A.a=31
PB->arată();            // Se selectează metoda arată din clasa B          B.b=9
PD->arată();            // Se selectează metoda arată din clasa D          D.d=35
}
```

Aşa cum se observă din exemplu, `pa`, `pb`, `pc` şi `pd` sunt pointeri de tipurile `A`, `B`, `C`, respectiv `D`:

```
A *pa;B *pb;C *pc;D *pd;
```

În urma atribuirii

```
pa=&v1;
```

pointerul `pa` (de tip `A`) va conține adresa obiectului `v1` (de tip `B`). Apelul metodei `arată` redefinite în clasa derivată `B`

```
pa->arată();
```

va determina ***selecția metodei din clasa pointerului*** (`A`), şi nu a metodei din clasa obiectului a cărui adresa o conține pointerul.

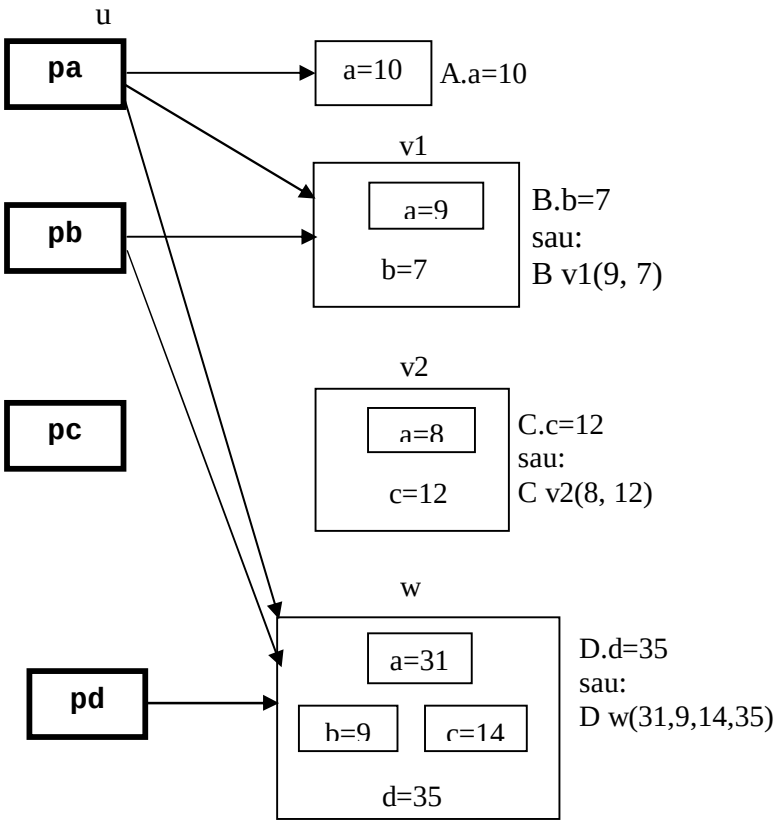

Figura 12.9. Un pointer către o clasă de bază inițializat cu adresa unui obiect dintr-o clasă derivată

În toate cazurile prezentate anterior, identificarea metodei redefinite se realizează în faza de compilare. Este vorba de o ***legare inițială, "early binding"***, în care toate informațiile necesare selectării metodei sunt prezentate din timp şi pot fi utilizate din faza de compilare.

12.7. METODE VIRTUALE

Aşa cum s-a subliniat, un pointer la o clasă de bază poate primi ca valoare adresa unui obiect dintr-o clasă derivată. Deci, având un tablou de pointeri la obiecte de tip A, putem lucra cu tablouri de obiecte *eterogene*, cu elemente de tipuri diferite (B, C sau D). În unele situații, informațiile privind tipul obiectului la care pointează un element al tabloului sunt disponibile abia în momentul execuției programului. O rezolvare a identificării metodei în momentul execuției programului o constituie ***funcțiile virtuale***.

Identificarea unei metode supradefinite, în momentul execuției, se numeşte ***legare ulterioară, "late binding"***.

Dacă dorim ca selectarea metodei `arată`, din exemplul anterior, să se realizeze în momentul execuției, metoda va fi declarată **metodă virtuală** .

Exemplu:

```
class A {
public:
      virtual void arată();
            //în loc de void arată()
            //....
};
class B : virtual public A {
public:
      virtual void arată();
            //în loc de void arată()
            //....
};
class C  : virtual public A {
public:
      virtual void arată();
            //în loc de void arată()
            //....
};
class B : public B, public C{
public:
      virtual void arată();
            //în loc de void arată()
            //....
};
```

În urma acestei modificări, rezultele execuției programului anterior ar fi fost:

```
void main()
{
A u(10), *PA;      // Constructor A  10
B v1(9, 7), *PB;   // Constructor A  9      Constructor B  7              - ptr. v1
C v2(8,12), *PC;   // Constructor A  8      Constructor C 12              - ptr. v2
D w(31, 9, 14, 35), *PD;       // Constructor A  31     Constructor B  9
                               // Constructor C  14     Constructor D  35
PA=&u; PA->arată();            // Se selectează metoda arată din clasa A       A.a=10
PA=&v1; PA->arată();           // Se selectează metoda arată din clasa B       B.b=7
PB=&v1; PB->arată();           // Se selectează metoda arată din clasa B       B.b=7
PA=&w; PB=&w; PD=&w;
```

```
u.arată();          // Apelul metodei arată ptr. obiectul  curent, clasa A   A.a=10
PA->arată();        // Se selectează metoda arată din clasa D                 D.d=35
PB->arată();        // Se selectează metoda arată din clasa D                 D.d=35
PD->arată();        // Se selectează metoda arată din clasa D                 D.d=35
}
```

Observaţie:

1. Deoarece metoda `arată` este virtuală, s-a selectat metoda pentru clasa obiectului spre care pointează pointerul.
2. Dacă *în clasa de bază* se declară o ***metodă virtuală,*** *în clasele derivate* metodele cu aceeaşi semnatură vor fi considerate ***implicit virtuale*** (chiar dacă ele nu sunt declarate, explicit, virtuale).

În cazul unei funcţii declarate virtuală în clasa de bază şi redefinite în clasa derivată, redefinirea metodei în clasa derivată are prioritate faţă de definirea ei din clasa de bază. Astfel, o funcţie virtuală declarată în clasa de bază actionează *ca un substitut* pentru păstrarea datelor care specifică o clasă generală de acţiuni şi declară forma interfeţei. La prima vedere, redefinirea unei funcţii virtuale într-o clasă derivată pare similară cu supraîncărcarea unei funcţiei obişnuite. Totuşi, nu este aşa, deoarece prototipul unei metode virtuale redefinite trebuie să coincidă cu cel specificat în clasa de bază. În cazul supraîncărcării unei funcţii normale, caracteristicile prototipurilor trebuie să difere (prin tipul returnat, numărul şi/sau tipul parametrilor).

Exerciţiu: Fie ierahia de clase din figura 12.10. Metoda virtuală `virt_f` , din clasa `bază`, este redefinită în clasele derivate.

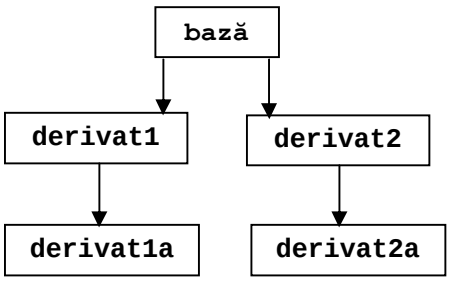

Figura 12.10. Ierarhie de clase

```
#include <iostream.h>
class baza{
public:
        bază()
            {cout<<"Constructor bază\n";}
        ~bază()
            {cout<<"Destructor bază\n";}
        virtual void virt_f()
            {cout<<"Metoda virt_f() din bază\n";}
};
class derivat1: public baza{
public:
        derivat1():baza()
            {cout<<"Constructor derivat1\n";}
        ~derivat1()
            {cout<<"Destructor derivat1\n";}
        virtual void virt_f()
            {cout<<"Metoda virt_f() din derivat1\n";}
};
class derivat2: public baza{
public:
        derivat2():baza()
            {cout<<"Constructor derivat2\n";}
        ~derivat2()
            {cout<<"Destructor derivat2\n";}
        virtual void virt_f()
            {cout<<"Metoda virt_f() din derivat2\n";}
};
class derivat1a: public derivat1{
public:
        derivat1a():derivat1()
            {cout<<"Constructor derivat1a\n";}
        ~derivat1a()
            {cout<<"Destructor derivat1a\n";}
        virtual void virt_f()
            {cout<<"Metoda virt_f() din derivat1a\n";}
};
```

```
class derivat2a: public derivat2{
public:
        derivat2a():derivat2()
            {cout<<"Constructor derivat2a\n";}
        ~derivat2a()
            {cout<<"Destructor derivat2a\n";}
        virtual void virt_f()
            {cout<<"Metoda virt_f() din derivat2a\n";}
};
void main()
{     baza *p;          //Constructor bază
      baza b;           //Constructor bază
      derivat1 d1;       // Constructor bază     Constructor derivat1
      derivat2 d2;       // Constructor bază     Constructor derivat2
      derivat1a d1a;    // Constructor bază     Constructor derivat1    Constructor derivat1a
      derivat2a d2a;    // Constructor bază     Constructor derivat2    Constructor derivat2a
      p=&b; p->virt_f();       // Metoda virt_f() din bază
      p=&d1;p->virt_f();       // Metoda virt_f() din derivat1
      p=&d2;p->virt_f();       // Metoda virt_f() din derivat2
      p=&d1a;p->virt_f();      // Metoda virt_f() din derivat1a
      p=&d2a;p->virt_f();      // Metoda virt_f() din derivat2a
}
      // Destructor derivat2a  Destructor derivat2   Destructor bază    (pentru d2a)
      // Destructor derivat1a  Destructor derivat1   Destructor bază    (pentru d1a)
      // Destructor derivat2   Destructor bază                          (pentru d2)
      // Destructor derivat1   Destructor bază                          (pentru d1)
      // Destructor bază                                                (pentru b)
```

Exerciţiu: Fie ierarhia de clase din figura 12.11. Se prezintă o modalitate de lucru cu un tablou eterogen, cu 5 elemente, care conţine pointeri atât spre clasa `baza`, cât şi spre clasele `derivat1` şi `derivat2`. Pentru a putea trata în mod uniform cele trei tipuri de obiecte, s-a creat clasa `lista_eterogena`. Aceasta are ca dată membru pointerul la tipul `baza` şi metoda `afis` (virtuală, redefinită în clasele derivate).

```
#include <iostream.h>
#include <conio.h>
class baza{
protected:
      int val;
public:
      baza()
            {cout<<"Constructor baza\n";}
      ~baza()
            {cout<<"Destructor baza\n";}
      void set_val(int a)
            {val=a;}
      virtual void afis()
            {cout<<"Element baza="<<val<<"\n";}
};
class derivat1: public baza{
public:
      derivat1():baza()
            {cout<<"Constructor derivat1\n";}
      ~derivat1()
            {cout<<"Destructor derivat1\n";}
      void afis()
            {cout<<"Element derivat1="<<val<<"\n";}
};
class derivat2: public  baza{
public:
```

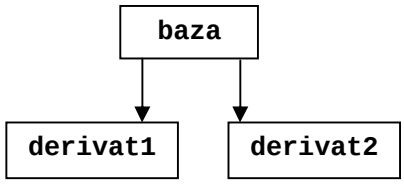

Figura 12.11.

```
      derivat2():baza()
            {cout<<"Constructor derivat2\n";}
      ~derivat2()
            {cout<<"Destructor derivat2\n";}
      void afis()
            {cout<<"Element derivat2="<<val<<"\n";}
};
class lista_eterogena {
      baza *p;
public:
      void set_l(baza *pp)    {p=pp;}
      void afis()             {p->afis();}
};

void main()
{     clrscr();
      baza B[3]; //Constructor baza      Constructor baza      Constructor baza
            // (pentru elementele tabloului B, de tip baza

      derivat1 D1; //Constructor baza    Constructor derivat1  (pentru D1, de tip derivat1)
      derivat2 D2; //Constructor baza    Constructor derivat2  (pentru D2, de tip derivat2)
      lista_eterogena L[5];
      cout<<"Apasa o tasta. . .\n";getch();
      B[0].set_val(10); B[1].set_val(100); B[2].set_val(1000);
      D1.set_val(444); D2.set_val(555);
      L[0].set_l(&B[0]);            //L[0].set_val(B);
      L[1].set_l(&D1);
      L[2].set_l((baza*) &D2);
      L[3].set_l(&B[1]);
            //L[3].set_l(B+1);
            //L[4].set_l(&B[2]);
      L[4].set_l(B+2);
      for (int i=0; i<5; i++)       L[i].afis();
            /*Element baza=10     Element derivat1=444      Element derivat2=555
            Element baza=100      Element baza=1000*/
}
```

În cazul unei ierarhii de clase şi a unei metode virtuale a clasei de bază, toate clasele derivate care moştenesc această metodă şi nu o redefinesc, o moştenesc întocmai. Pentru aceeaşi metodă moştenită şi redefinită în clasele derivate, selecția se realizează în momentul executării programului (legarea târzie).
Funcțiile virtuale ***nu*** pot fi ***metode statice ale clasei din care fac parte***.
Funcțiile virtuale nu pot fi funcții prietene sau constructori, dar pot fi destructori. Destructorii virtuali sunt utili în situațiile în care se doreşte distrugerea uniformă a unor masive de date eterogene.

Metode virtuale pure
În unele situații, o clasă de bază (din care se derivează alte clase) a unei ierarhii, poate fi atât de generală, astfel încât unele metode nu pot fi descrise la acest nivel (atât de abstract), ci doar în clasele derivate. Aceste metode se numesc ***funcții pure*** . ***Metodele virtuale pure*** sunt metode care *se declară*, ***nu se definesc*** la acest nivel de abstractizare. O metodă virtuală pură trebuie să fie prezentă în orice clasă derivată.

Exemple:

```
      class bază{
      public:
            virtual void virt_f()=0; };     //metoda virt_f este o metodă virtuală pură
      class vieţuitoare {
      public:
            virtual void nutriție()=0; };  //metoda nutriție este o metodă virtuală pură
```

O clasă cu cel puțin o metodă virtuală pură se numeşte ***clasă abstractă*** (clasa `vieţuitoare` este abstractă şi, ca urmare, ***nu poate fi instanțiată***).

ÎNTREBĂRI ŞI EXERCIŢII

Chestiuni teoretice

1. Ce este o clasă derivată şi ce caracteristici are?
2. Funcţiile prietene pot fi funcţii virtuale?
3. Destructorii se moştenesc?
4. Ce este o clasă virtuală şi în ce situaţii este utilă?
5. Ce este o metodă virtuală pură şi cum se declară aceasta?
6. Explicaţi ce înseamnă legarea iniţială (early binding).
7. Modul de declarare a unei clase derivate, cu mai multe superclase.
8. Ce este o metodă virtuală ?
9. Funcţiile virtuale pot fi membrii statici ai clasei din care fac parte ?
10. Redefinirea unei funcţii virtuale într-o clasă derivată este similară cu supraincarcarea funcţiei respective? Argumentati răspunsul.
11. Care este utilitatea moştenirii?
12. Explicaţi ce înseamnă legarea ulterioară (late binding).

Chestiuni practice

1. Să se implementeze ierarhia de clase din figura 12.12., cu membrii pe care îi consideraţi necesari.

2. Concepeţi o ierarhie de clase a figurilor geometrice. Ca date membre pot fi considerate poziţia, dimensiunile şi atributele de desenare (culoare, tip linie). Metodele vor permite operaţii de afişare, deplasare, ştergere, modificarea atributelor figurii. Clasa de bază va avea proprietăţile generale ale oricărei figuri: coordonatele pe ecran şi vizibilitate.

3. Din clasa matrice, să se deriveze clasa c_matrice, care reprezintă o matrice de complecşi.

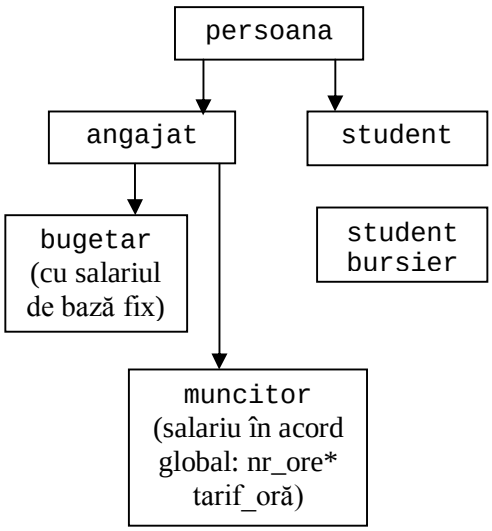

Figura 12.12.