Capitolul 8

Ierarhii de clase

Moştenirea reprezintă elementul distinctiv al paradigmei programării orientate pe obiecte. Deoarece ea permite ca o anumită clasă să poată moşteni datele şi funcţiile membru ale altor clase, rezultă de aici posibilitatea existenţei unei relaţii ierarhice între clase.

Operaţia de moştenire mai este numită operaţie de ***derivare***: o clasă *A* care moşteneşte membrii unei alte clase *B*, se spune că este ***derivată*** din clasa *B*, iar clasa *A* este o ***clasă de bază*** pentru *B*. În limbajul UML relaţia de moştenire se notează astfel:

sageata fiind orientată spre clasa de bază.

De exemplu, pentru secvenţa următoare, unde clasele *P* şi *Q* moştenesc clasa *X*:

```
class X {
  // ...
};

class P: X {
  // ...
};

class Q: X {
  // ...
};
```

diagrama de clase se poate reprezenta ca în figura 8.1.

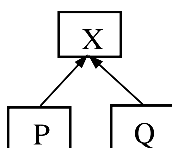

Figura 8.1

Observaţie: O clasă poate moşteni mai multe clase, caz în care este vorba despre o ***moştenire multiplă***, în caz contrar fiind vorba despre o ***moştenire simplă***.

8.1 Derivarea claselor

Sintaxa pentru derivarea unei clase din una sau mai multe clase de bază a fost prezentată în paragraful 3.1. Specificarea moştenirii se face în declaraţia clasei, imediat după numele clasei curente, prin enumerarea claselor moştenite.

În acest fel se pot realiza ierarhii de clase bazate pe relaţia de moştenire. În cazul în care se utilizează doar moştenirea simplă, o ierarhie de clase reprezintă o structura arborescentă, toate clasele din ierarhie fiind derivate dintr-o singură clasa de bază. Aceasta reprezintă o manieră simplă şi sigură de proiectare a unei aplicaţii.

În cazul în care se utilizează şi moştenirea multiplă, structura ierarhică reprezintă un graf orientat, clasele de bază ce generează ierarhia respectivă fiind acele noduri care nu au nici un arc de ieşire.

Observaţie: Uniunile nu pot face parte dintr-o ierarhie de mai multe clase, deoarece ele nu pot fi nici clase de bază şi nici clase derivate.

Exemplul 8.1.

```
struct Punct {
  double x,y;
  void SetCoord (double a, double b) const {
    x = a;
    x = b;
  }
};

struct Cerc: Punct {
  double r;
  void SetRaza (double a) const { r = a; }
};
```

Se observă faptul că într-o clasă derivată se specifică doar membrii suplimentari care se adaugă la clasa de bază.

Un obiect dintr-o clasă derivată este privit de către compilator ca având toţi membrii specificati în clasa derivată, la care se adaugă un ***obiect ascuns*** care este o instanţă a clasei de bază. În acest mod moştenirea reprezintă o relaţie de compunere specială. Pentru exemplul precedent, un obiect instanţă al clasei *Cerc* are o structură prezentată în figura 8.2.

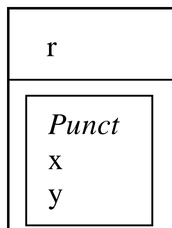

Figura 8.2

În secvenţa următoare:

```
Punct p1;
```

```
p1.SetCoord(1, 0);
Cerc c1;
c1.SetCoord(7, 9);
c1.SetRaza(2);
```

obiectul *c1* conține un obiect ascuns al clasei *Punct*, care însă nu trebuie specificat prin numele său (nu mai este nevoie de un operator de selecție suplimentar). În acest mod, membrii clasei *Punct* pot fi accesați direct, ca şi când ar fi fost declarați în clasa *cerc*.

Moştenirea membrilor unei clase de bază într-o clasă derivată poate modifica dreptul de acces al acestora. Există două moduri în care se poate moşteni o clasă de bază: `public` şi `private`. Cuvântul cheie ce specifică tipul de moştenire al clasei de bază precede numele clasei de bază, iar în cazul în care acesta este absent, moştenirea este implicit de tip `public` pentru `struct` şi `private` pentru `class`.

Accesul la membrii clasei de bază poate fi făcut uneori restrictiv în clasa derivată, dar niciodata mai puțin restrictiv, după cum se poate vedea din tabelul următor:

Tipul de acces al unui membru în clasa de bază	*Tipul de moştenire*	*Tipul de acces al membrului după moştenire*
Public Private Protected	Public	Public Inaccesibil Protected
Public Private Protected	Private	Private Inaccesibil Private

Se observă faptul că membrii `private` ai clasei de bază sunt inaccesibili în clasa derivată, indiferent de tipul de moştenire. Celelalte două tipuri de acces sunt păstrate nemodificate în clase derivate în cazul moştenirii de tip `public` sau modificate la tipul `private` în cazul moştenirii de tip `private`.

Din acelaşi tabel se observă rolul tipului de acces `protected` al unui membru dintr-o clasă de bază: acela de a permite utilizarea lui în clasele derivate din aceasta, dar de a nu permite accesul la el din exteriorul clasei de bază.

Exemplul 8.2.

```
class A {
  int p;
public:
  int q;
protected:
  int r;
  // ...
};

class A1: A {
  int x;
public:
```

```
  void f1();
  // ...
};

class A2: public A {
  int y;
public:
  void f2();
  // ...
};
```

În implementarea funcţiilor *f1* şi *f2* se pot utiliza direct şi membrii *q* şi *r* din clasa *A*, însă în clasa *A1* membrii *q* şi *r* sunt inaccesibili în afară (de exemplu, nu se poate scrie o instrucţiune de forma:

```
A1 a1;
int k = a1.q;
```

pe când în clasa *A2* membrul *q* este `public` (secvenţa urmatoare este corectă):

```
A2 a2;
int k = a2.q;
```

Observaţie. Dacă se doreşte ca anumiţi membri publici din clasa de bază care au devenit privaţi printr-o moştenire de tip privat să fie publici în clasa derivată, numele lor trebuie specificat după cuvântul cheie `public` (doar numele, nu şi tipul de date). De exemplu, dacă se doreşte ca în clasa *A1* membrul *q* să fie public, declaraţia clasei *A1* trebuie scrisă astfel:

```
class A1: A {
  int x;
public:
  A::q;
  void f1();
  // ...
};
```

În acest mod, membrul *q* este vizibil şi în exteriorul clasei *A1*.

8.2 Constructori şi destructori în ierarhii de clase

Constructorii şi destructorii sunt singurii membrii ai unei clase care nu pot fi moşteniţi într-o clasă derivată (în afară de operatorul de atribuire, despre care se va discuta într-un capitul ulterior). Acest lucru este natural, deoarece operaţiile de creare şi distrugere a obiectelor unei clase sunt proprii clasei respective şi nu pot fi transmise mai departe într-o ierarhie de clase.

Deoarece un obiect al unei clase derivate este asimilat cu un obiect compus care are ca obiect membru ascuns o instanţă a clasei de bază, constructorul pentru obiectul clasei derivate trebuie să apeleze mai întâi constructorul obiectului instanţă al clasei de bază. Apelul constructorului obiectului ascuns se poate face explicit într-o listă de iniţializare a constructorului obiectului clasei derivate, sau se poate genera implicit de compilator în cazul în care clasa de bază nu are declaraţi constructori.

Apelul constructorului clasei de bază în lista de inițializare a constructorului clasei derivate se face prin numele clasei de bază, deoarece obiectul ascuns corespunzator clasei de bază nu are nume.

Exemplul 8.3.

```
class Punct {
protected:
  double x, y;
public:
  void SetCoord(double a, double b) {
    x = a;
    y = b;
  }
  Punct(double a = 0, double b = 0) {
    SetCoord (a, b);
  }
  double X() const { return x; }
  double Y() const { return y; }
};

class Cerc: public Punct {
  double r;
public:
  void SetR(double a) { r = 0; }
  Cerc(double a = 0, double b = 0, double c = 1):
     Punct(a,b), r(c) { }
  Cerc(Cerc& c): Punct(c.x, c.y), r(c.r) {}
};
```

Se cunoaşte faptul că un constructor pentru o clasă poate fi generat automat de către compilator, în cazul în care clasa respectivă nu are declarat un constructor. Există o excepție, în cazul în care o clasă este derivată din una sau mai multe clase de bază şi toate clasele de bază au constructori expliciți ce conțin parametri: în acest caz clasa derivată trebuie să conțină un constructor explicit care apelează de asemenea explicit constructorii claselor de bază care au parametri.

Destructorii pentru obiectele din clase derivate apelează în mod implicit destructorii obiectelor instanță ale claselor derivate în ordinea inversă apelului constructorilor. De exemplu, pentru un obiect instanță *O* al unei clase *D* derivată dintr-o clasă *B*, ordinea apelului desstructorilor este următoarea:

- apelul destructorului obiectului *O*;
- apelul destructorilor membrilor suplimentari ai clasei *D*;
- apelul destructorului clasei *B*;
- apelul destructorilor membrilor din clasa *B*;

Observație: În cazul moştenirii multiple, apelul constructorilor obiectelor instanță al claselor de bază se face în ordinea specificării claselor de bază în declarația clasei derivate, iar apelul destructorilor în ordine inversă.

Exemplul 8.4. Se va utiliza o macrodefiniție pentru definiția unor clase:

```
#include <iostream>
using namespace std;

#define CLASS(ID) class ID {\
  public:\
    ID(int){cout<<"Constructor clasa "<<#ID<<endl;}\
    ~ID(){cout<<"Destructor clasa "<<#ID<<endl;}\
  };

CLASS(B1);
CLASS(B2);
CLASS(M1);
CLASS(M2);

class D: public B1, B2 {
  M1 m1;
  M2 m2;
public:
  D(int): m1(10), m2(20), B1(30), B2(40) {
    cout << "Constructor clasa D" << endl;
  }
  ~D() { cout << "Destructor clasa D" << endl; }
};

void main() {
  D d(0);
  // ...
}
```

Ieşirea programului este următoarea:

```
Constructor clasa B1
Constructor clasa B2
Constructor clasa M1
Constructor clasa M2
Constructor clasa D
Destructor clasa D
Destructor clasa M2
Destructor clasa M1
Destructor clasa B2
Destructor clasa B1
```

În cazul în care o clasă derivată nu posedă un *constructor de copiere* propriu, acesta va fi generat în mod automat de către compilator. El apelează constructorii de copiere al claselor de bază, urmat dacă este cazul, de apelul constructorilor de copiere ai obiectelor membre ale clasei (sau ai pseudo-constructorilor în cazul tipurilor predefinite).

Exemplul 8.5.

```
#include <iostream>
```

```
using namespace std;

class Baza {
  int n;
public:
  Baza(int i): n(i) {
    cout << ”Baza(int i)” << endl;
  }
  Baza(const Baza& b): n(b.n) {
    cout << ”Baza(const Baza& b)” << endl;
  }
  Baza(): n(0) { cout << ”Baza()” << endl; }
  void Print() const {
    cout << ”Baza; n=‟ << n << endl;
  }
};

class Membru {
  int n;
public:
  Membru(int i): n(i) {
    cout << ”Membru(int i)” << endl;
  }
  Membru(const Membru& m): n(m.n) {
    cout << ”Membru(const Membru& m)” << endl;
  }
  void Print() const {
    cout << ”Membru; n=‟ << n << endl;
  }
};

class Derivat: public Baza {
  int n;
  Membru m;
public:
  Derivat(int i): Baza(i), n(i), m(i) {
    cout << ”Derivat(int i)” << endl;
  }
  void Print() const {
     cout << ”Derivat; n=‟ << n << endl;
     Baza::Print();
     m.Print();
  }
};

void main() {
  Derivat o1(7);
  cout << ”Apel constructor de copiere: ‟ << endl;
  Derivat o2 = o1;
  cout << ”Valori in o2: ” << endl;
  o2.Print();
```

```
}
```

Ieşirea programului va fi următoarea:

```
Baza(int i)
Membru(int i)
Derivat(int i)
Apel constructor de copiere:
Baza(const Baza& b)
Membru(const Membru& m)
Valori in o2:
Derivat; n=7
Baza; n=7
Membru; n=7
```

Constructorii de copiere pentru clasele *Baza şi Membru* au fost apelaţi implicit (liniile 5 şi 6), precum şi pseudo-constructorul de copiere pentru membrul *n* (in linia 10, valoarea componentei *n* a obiectului *o2* este tot 7, ca şi cea din obiectul *o1*).

În cazul în care se doreşte scrierea unui constructor de copiere, apelul constructorului pentru clasa de bază trebuie să fie explicit.

Exemplu de constructor de copiere eronat pentru clasa *Derivat*:

```
Derivat(const Derivat& d): n(d.n), m(d.m) { }
```

În acest caz ultimele 3 linii de ieşire ale programului sunt:

```
Derivat; n=7
Baza; n=0
Membru; n=7
```

Deoarece nu există un apel explicit pentru un constructor al clasei *Baza*, compilatorul va insera un constructor implicit pentru aceasta.

Un exemplu de constructor de copiere corect pentru clasa *Derivat* poate fi următorul

```
Derivat(const Derivat& d): Baza(d), n(d.n), m(d.m) { }
```

Ieşirea programului va fi în acest caz la fel cu cea din exemplul precedent.

Observaţie: Parametrul constructorului de copiere pentru clasa *Baza* este o referinţă la un obiect al clasei derivate, ceea ce este o operaţie corectă în cazul moştenirii publice (clasa *Derivat* este un subtip al clasei *Baza*). În cazul moştenirii private, probabil este indicat ca un asemenea constructor de copiere să fie apelat implicit de către compilator.

8.3 Moştenirea publică şi privată

Utilizarea tipului de moştenire publică sau privată are o semnificaţie distinctă în proiectarea şi dezvoltarea unor aplicaţii.

A. Moştenirea publică

Moştenirea publică este legată în special de etapa de proiectare a unei aplicaţii, în cazul în care se doreşte ca o anumită clasă să reprezinte o specializare a clasei de bază. De exemplu, clasa *student* este o specializare a clasei *persoana*, în sensul că orice student este o persoană. În literatura acest tip de relaţie este denumit "este-o" sau "este-un".

Acest tip de moştenire presupune moştenirea într-o clasă derivată, atât a interfeţei clasei de bază, cat şi a implementării acesteia, în acest mod un obiect al clasei derivate putând fi utilizat în locul unui obiect al clasei de bază.

Exemplul 8.6. Reluarea exemplului precedent relativ la clasele *Punct* şi *Cerc*. Funcţia *Distanta* determină distanţa între două puncte:

```
double Distanta (Punct& p1, Punct& p2) {
  double d = sqrt((p1.X()-p2.X())*(p1.X()-p2.X())+
                  (p1.Y()-p2.Y())*(p1.Y()-p2.Y()));
  return d;
}
```

Următoarea secvenţă este corectă şi afişează valorile 1 şi 4:

```
Punct p1, p2;
p1.SetCoord(1, 1);
p2.SetCoord(0, 0);
Cerc c1, c2;
c1.SetCoord(7, 7);
c2.SetCoord(3, 3);
double d1 = Distanta(p1, p2);
double d2 = Distanta(c1, c2);
cout << d1 << d2 << endl;
// ...
```

Datorită faptului că o clasă derivată public moşteneşte atât declaraţia cat şi implementarea clasei de bază, într-o clasă derivată se pot ***redefini*** membrii din clasa de bază. Redefinirea unui membru presupune ascunderea în clasa derivată a membrului din clasa de bază cu acelaşi nume.

Exemplul 8.7.

```
class A {
public:
  void f() const {
    cout << "f in clasa A" << endl;
  }
};

class B: public A {
public:
  void f() const {
    cout << "f in clasa B" << endl;
  }
};
```

```
void main (){
  A a;
  a.f();        //f din clasa A
  B b;
  b.f();        //f din clasa B
  b.A::f();     //f din clasa A
  // ...
}
```

În cazul în care se doreşte utilizarea membrului ascuns, acesta trebuie prefixat de numele clasei de bază.

Observaţie: În mod uzual redefinirea unor membrii dintr-o clasă de bază într-o clasă derivată nu este indicată, ea presupunând o eroare de proiectare a ierarhiei de clase. În cazul în care se doreşte acest lucru, vor trebui utilizate ***funcţii virtuale***.

Exemplul 8.8. Deşi majoritatea păsărilor zboară, există şi păsări care nu zboară; de exemplu pinguinul. Exemplul următor reprezintă o proiectare greşită a ierarhiei de clase :

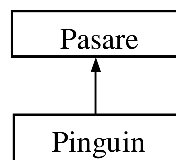

Figura 8.3

```
class pasare {
public:
  void zboara () const {
    cout << "pasarea zboara" << endl;
  }
  // ...
};

class pinguin: public pasare {
  // ...
};

// ...
pasare vultur;
pinguin pinguin;
vultur.zboara();     //corect
pinguin.zboara();    //eroare!!
```

O variantă de modificare a ierarhiei constă în redefinirea funcţiei *zboara* în clasa *pinguin*:

```
class pinguin: public pasare {
public:
  void zboara () const {
    cout << "pasarea nu zboara" << endl;
```

```
  }
  // ...
};

// ...
pinguin pinguin;
pinguin.zboara();   //corect
```

O proiectare corectă a ierarhiei de clase trebuie însă să distingă între cele două categorii de păsări.

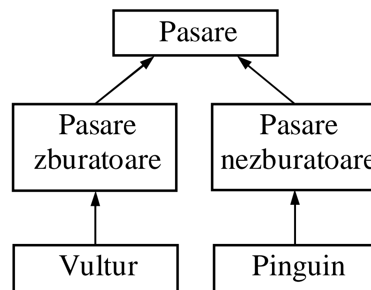

Figura 8.4

```
class PasareZburatoare: public pasare {
public:
  void zboara () const {
    cout << ”pasarea zboara” << endl;
  }
  // ...
};

class PasareNezburatoare: public pasare {
public:
  void zboara () const {
    cout << ”pasarea nu zboara” << endl;
  }
// ...
};

class Vultur: public PasareZburatoare {
  // ...
};

class pinguin: public PasareNezburatoare {
  //...
};

// ...
Vultur vultur;
Pinguin pinguin;
Vultur.zboara();
```

B. Moştenirea `private`

Moştenirea private este specifică etapei de dezvoltare a unei aplicaţii, când se dispune de anumite ierarhii de clase deja proiectate.

În cazul moştenirii `private`, toţi membrii clasei de bază devin membri `private` în clasa derivată, ceea ce îi face să nu poată fi utilizaţi în exteriorul clasei derivate (cu excepţia celor `private` din clasa de bază, care devin inacesibili în clasa derivată). Cu alte cuvinte, în acest caz o clasă derivată moşteneşte doar implementarea clasei de bază, nu şi interfaţa acesteia. Din acest motiv, acest tip de relaţie se mai numeste în literatura de specialitate "`este_o_implementare_a_lui`".

În cazul moştenirii private, un obiect din clasa derivată nu mai este convertit de către compilator într-un obiect al clasei de bază (clasa derivată nu mai reprezintă un subtip al clasei de bază).

Exemplul 8.9. Reluarea exemplului cu clasele *Punct* şi *Cerc*:

```
class Punct {
protected:
  double x, y;
public:
  void SetCoord (double a, double b) {
    x = a;
    x = b;
  }
  Punct (double a = 0, double b = 0) {
    SetCoord(a, b);
  }
  double X() const { return x; }
  double Y() const { return y; }
};

class Cerc: Punct {
public:
  double r;
  void SetRaza (double a = 1) { r = a; }
  Cerc (double a = 0, double b = 0, double c = 1):
         Punct(a,b),r(c) { }
};

double Distanta(Punct p1, Punct p2) {
  double d = sqrt((p1.X()-p2.X())*(p1.X()-p2.X())+
                 (p1.Y()-p2.Y())*(p1.Y()-p2.Y()));
  return d;
}

// ...
punct p1(0, 0), p2(1, 1);
cerc c1(3, 3, 1), c2(7, 7, 2);
```

```
double d1 = Distanta(p1, p2); //corect
double d2 = Distanta(c1, c2); //eroare!!!
```

Observaţie. Acelaşi tip de relaţie "este_o_implementare_a_lui" o reprezintă şi compunerea obiectelor, deoarece pentru un membru obiect într-o clasă compusă se moşteneşte doar implementarea, nu şi interfaţa acesteia. În mod uzual este indicată utilizarea compunerii obiectelor în acest caz, ori de cate ori este posibil. Utilizarea moştenirii private este necesară doar în cazurile în care clasa de bază conţine membrii de tip `protected`, care altfel nu pot fi utilizaţi în clasa derivată.

Exemplul 8.10. Să presupunem că se doreşte scrierea unei clase *ListaOrdonata*, care să memoreze elementele unei liste în ordine crescătoare. Metoda folosită va fi cea de inserare a fiecărui element în listă într-o poziţie corespunzătoare, astfel încât lista să rămână ordonată şi după inserare (metoda sortării prin inserare). Având la dispoziţie clasa *list*, definită în capitolul 5, există două variante: derivarea clasei *ListaOrdonata* din clasa *list*, sau definirea clasei *ListaOrdonata* ca o clasă compusă ce conţine un obiect de tipul *list*. Derivarea publică nu este indicată, deoarece în acest mod se va moşteni şi interfaţa clasei *list* şi deci vor fi vizibile toate funcţiile de adăugare a unui element în listă, ceea ce face ca o listă ordonată să poată devini neordonată prin apelul funcţiilor *AddFirst* sau *AddLast*.

```
class ListaOrdonata: list {
public:
  ListaOrdonata();
  ~ListaOrdonata();
  void InsertOrdonat(int);
  void Print();
};
```

C. Conversia de tip între clasele de bază şi clasele derivate în cazul moştenirii publice

O proprietate importantă a relaţiei de derivare o constituie faptul că o clasă derivată `public` este tratată ca un ***subtip*** al clasei de bază. În acest mod obiectele instanţă ale clasei derivate sunt compatibile cu obiectele clasei de bază şi ***pot apare într-o instrucţiune de atribuire***. Pentru exemplul precedent al claselor *Cerc* şi *Punct*, următoarea instrucţiune de atribuire este corectă:

```
Punct p2;
Cerc c1;
p2 = c1;
```

Operatorul de atribuire va fi discutat mai târziu, dar pentru exemplul anterior se consideră că el copiază bit cu bit datele membru ale obiectului din dreapta, în datele obiectului din stânga.

Observaţii:
1. Operaţia de atribuire copiază doar membrii din clasa de bază;
2. Atribuirea poate funcţiona doar într-o singură direcţie: *nu se poate atribui* un obiect al unei clase de bază unui obiect al unei clase derivate. De exemplu, instrucţiunea următoare este greşită:

Regula compatibilităţii la atribuire se extinde şi la pointeri şi referinţe de obiecte. De exemplu, următoarea secvenţă de program este corectă:

```
class B {
  // ...
};

class A1: public B {
  // ...
};

class A2: public B {
  // ...
};

// ...
A1 a1;
A2 a2;
B *pb;
pb = &a1;
pb = &a2;
```

În exemplul precedent există o conversie implită de tip de la un pointer al unei clase derivate la un pointer al clasei de bază. O asemenea conversie, de la un pointer (sau o referinţă) la o clasă derivată spre un pointer (sau o referinţă) la clasa de bază este numit ***upcasting***. Denumirea vine de la conversia implicită de tip (`cast`) şi de la sensul de conversie din ierarhia de clase (`up`), de la o clasă din partea de jos a unei ierarhii spre o clasă din partea de sus.

Operaţia de ***upcasting*** este destul de frecvent utilizată în aplicaţiile ce folosesc ierarhii de clase, permiţând o tratare uniformă a obiectelor dintr-o asemenea ierarhie prin intermediul pointerilor la clasa de bază.

De exemplu, în cazul moştenirii publice între clasele *Cerc* şi *Punct*, utilizarea funcţiei *Distanta* se bazează pe upcasting:

```
double Distanta (Punct& p1, Punct& p2) {
  // ...
}

Cerc c1(7, 3), c2(2, 2);
// upcasting de la Cerc& la Punct&
double d = Distanta(c1, c2);
```

8.4 Moştenirea multiplă

Nu toate limbajele care suportă paradigma OOP acceptă ***moştenirea multiplă***. De exemplu, limbajul Smalltalk, care a fost limbajul de referinţă până la apariţia limbajului C++ nu acceptă acest tip de moştenire. Smalltalk este numit un limbaj *pur orientat pe obiecte*, care are o

ierarhie predefinită de clase sub forma unui arbore cu o singură rădăcină numită `Object`. Orice clasă definită de un programator trebuie să fie derivată din `Object` sau dintr-o clasă de bază derivată din `Object`.

Limbajul C++ este numit un limbaj *impur* orientat pe obiecte, sau un limbaj *hibrid*, dar are avantajul că permite crearea unor clase şi ierarhii de clase proprii, independente de o anumită ierarhie predefinită.

Moştenirea multiplă apare în cazul în care o anumită clasă derivată moşteneşte (public sau privat) *mai multe* clase de bază. Un exemplu de moştenire multiplă există chiar în clasele predefinite ale limbajului pentru operațiile de intrare/ieşire. Diagrama acestora este prezentată în figura 8.5.

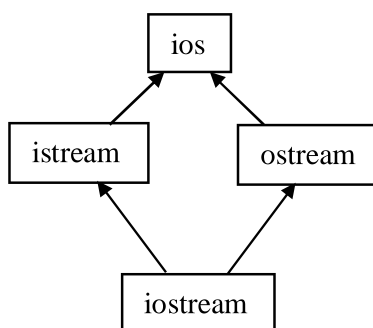

Figura 8.5

Există două probleme importante care pot apare în cazul utilizării moştenirii multiple: ***duplicarea obiectelor ascunse*** şi ***existența unor membri cu acelaşi nume*** în clase de bază diferite.

Duplicarea obiectelor ascunse poate apare în cazul ierarhiilor care au o formă ca în figura precedentă (numite şi ierarhii de tip diamant, datorită formei lor). Datorită faptului că un obiect al unei clase derivate posedă un sub-obiect ascuns al clasei de bază, pentru o ierarhie de forma:

```
class B {
  // ...
};

class M1: public B {
  int a;
  // ...
};

class M2: public B {
  int b;
  // ...
};

class MI: public M1, public M2 {
  int m;
```

```
  // ...
};
```

un obiect instață al clasei *MI* are o structură asemănătoare cu cea prezentată în figura 8.6.

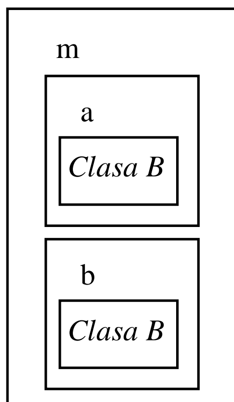

Figura 8.6

Se observă faptul că există două sub-obiecte ascunse identice, corespunzătoare clasei *B*. Acest fapt produce un spațiu suplimentar de memorie, care poate la rândul său să inducă ambiguități într-o aplicație care nu este judicios proiectată.

Principala problemă a moştenirii multiple este însă generată de cazul în care ***mai multe clase de bază posedă un membru cu acelaşi nume***: cum vor putea fi utilizați aceşti membrii în clasa derivată?

Exemplul 8.11.

```
class B1 {
public:
  int a;
  // ...
};

class B2 {
public:
  double a;
  // ...
};

class D: public B1, public B2 {
  // ...
};

void Prelucrare () {
  D d;
  d.a = 5;     //eroare!!
  // ...
}
```

În exemplul precedent, instrucțiunea `d.a=5;` generează o eroare datorită confuziei ce apare la selecția membrului *a*.

În acest caz există două variante uzuale de evitare a confuziei:

a) utilizarea explicită a unui anumit membru cu ajutorul operatorului de rezoluție;
b) redefinirea în clasa de bază a membrului cu acelaşi nume.

În primul caz necesitatea de a scrie un cod corect cade exclusiv în sarcina celui ce utilizează o ierarhie de clase deja proiectată. De exemplu, instrucțiunea care generează eroarea ar putea fi scrisă:

```
d.B1::a = 5;
```

sau

```
d.B2::a = 5;
```

Cazul al doilea reprezintă o rezolvare mai elegantă a problemei apărute, Operația fiind de data aceasta o sarcină a celui ce proiectează ierarhia de clase .

Exemplul 8.12. Exemplul următor reprezintă o ierarhie de clase ce defineşte o clasă *CercText* care permite afisarea unui text în interiorul unui cerc (nu s-au mai specificat implementarile funcțiilor membru care nu sunt `inline`):

```
class Punct {
protected:
  int x, y;
public:
  Punct(int a, int b): x(a), y(b) { }
  int X() const { return x; }
  int Y() const { return y; }
};

// punct grafic
class GPunct: public Punct {
protected:
  int vizibil;
public:
  GPunct(int a, int b): Punct(a, b), vizibil(1) { }
  void Show();        // afiseaza un punct grafic pe ecran
  void Ascunde() { vizibil = 0; }
  int EsteVizibil() const { return vizibil; }
  void Translatare(int a, int b) { x = a; y = b; }
};

class Cerc: public GPunct {
protected:
  int r;
public:
  Cerc(int a, int b, int c): GPunct(a, b), r(c) { }
  void Show();        // deseneaza un cerc
};

// afiseaza un mesaj pe ecran incadrat intr-un dreprunghi
```

```
class Mesaj: public Punct {
public:
  char *msg;          // mesajul
  int l, L;           // dimensiunea dreptunghiului
  Mesaj(int a, int b, int c, int d, char *m):
    Punct(a, b), l(c), L(d), msg(m) { }
  void Show();        // afiseaza mesajul
};

class CercText: Cerc, Mesaj {
public:
  CercText(int a, int b, int r, char *m):
    Cerc(a, b, r), Mesaj(a, b, r, r, m) { }
  void Show()         // afiseaza cercul cu mesaj interior
  {
    Cerc::Show();
    Mesaj::Show();
  }
};

void main() {
  // ...
  CercText c1(250, 100, 25, ”cerc C1”);
  c1.Show();
  // ...
}
```

Se observă faptul ca funcţia *Show* a fost moştenită în clasa *CercText* şi ascunde funcţiile cu acelaşi nume din clasele *Cerc* şi *Mesaj*, pe care le utilizează cu ajutorul operatorului de rezoluţie.

În ierarhia de clase anterioară, moştenirea multiplă nu este neapărat necesară. O alternativă mai aproape de semnificaţia reală, poate considera clasa *CercText* ca o clasă compusă ce conţine în interior două obiecte private, instanţe ale claselor *Cerc* şi *Mesaj*:

```
class CercText {
  Cerc cerc;
  Mesaj mesaj;
public:
  CercText(int a, int b, int r, char *m):
    cerc(a, b, r), mesaj(a, b, r, r, m) { }
  void Show()         // afiseaza cercul cu mesaj interior
  {
    cerc.Show();
    mesaj.Show();
  }
};
```

Necesitatea utilizării moştenirii multiple poate apare atunci când o clasă derivată necesită controlul asupra claselor de bază (prin utilizarea mecanismului funcţiilor virtuale, de exemplu).

Un alt exemplu se referă la necesitatea utilizării mai multor ierarhii de clase la care nu se pot modifica sursele (sunt disponibile doar ca biblioteci şi fişiere header). Un exemplu asupra căruia se va reveni ulterior în capitolul 10, consideră utilizarea unui container fără folosirea mecanismului `template`. Să presupunem că se dispune de o clasă *Container*, care memorează pointeri la o listă de obiecte abstracte dintr-o clasă *Object*. Pentru a putea utiliza clasa *Container* într-o aplicaţie, astfel încât să memoreze obiecte aparţinând unei alte clase, *Figura*, la care nu se poate avea acces, se poate crea o clasă nouă, *FiguraDerivata*, obţinută prin derivarea din clasele *Object* şi *Figura*, după cum se observă din figura 8.7.

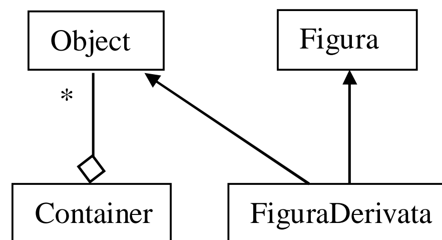

Figura 8.7

În acest mod, prin utilizarea mecanismului de upcasting, containerul poate memora referinţe şi la obiecte ale clasei *FiguraDerivata*.