

Capitolul 8

Ierarhii de clase

Moștenirea reprezintă elementul distinctiv al paradigmei programării orientate pe obiecte. Deoarece ea permite ca o anumită clasă să poată moșteni datele și funcțiile membru ale altor clase, rezultă de aici posibilitatea existenței unei relații ierarhice între clase.

Operația de moștenire mai este numită operație de **derivare**: o clasă *A* care moștenește 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 relația de moștenire se notează astfel:



sageata fiind orientată spre clasa de bază.

De exemplu, pentru secvența următoare, unde clasele *P* și *Q* moștenesc clasa *X*:

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

diagrama de clase se poate reprezenta ca în figura 8.1.

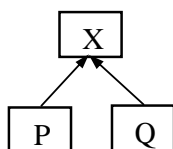


Figura 8.1

Observație: O clasă poate moșteni mai multe clase, caz în care este vorba despre o **moștenire multiplă**, în caz contrar fiind vorba despre o **moștenire 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 moștenirii se face în declarația clasei, imediat după numele clasei curente, prin enumerarea claselor moștenite.

În acest fel se pot realiza ierarhii de clase bazate pe relația de moștenire. În cazul în care se utilizează doar moștenirea simplă, o ierarhie de clase reprezintă o structura arborescentă, toate clasele din ierarhie fiind derivate dintr-o singură clasă de bază. Aceasta reprezintă o manieră simplă și sigură de proiectare a unei aplicații.

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

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

Exemplul 8.1.

```
struct Punct {
    double x,y;
    void SetCoord (double a, double b) const {
        x = a;
        y = 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 toți membrii specificați în clasa derivată, la care se adaugă un *obiect ascuns* care este o instanță a clasei de bază. În acest mod moștenirea reprezintă o relație de compunere specială. Pentru exemplul precedent, un obiect instanță al clasei *Cerc* are o structură prezentată în figura 8.2.

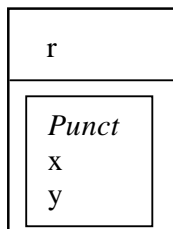


Figura 8.2

În secvența 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 și când ar fi fost declarați în clasa *cerc*.

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

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

<i>Tipul de acces al unui membru în clasa de bază</i>	<i>Tipul de moștenire</i>	<i>Tipul de acces al membrului după moștenire</i>
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 moștenire. Celelalte două tipuri de acces sunt păstrate nemodificate în clase derivate în cazul moștenirii de tip `public` sau modificate la tipul `private` în cazul moștenirii de tip `private`.

Din același 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 funcțiilor $f1$ și $f2$ se pot utiliza direct și membrii q și r din clasa A , însă în clasa $A1$ membrii q și r sunt inaccesibili în afară (de exemplu, nu se poate scrie o instrucțiune de forma:

```

A1 a1;
int k = a1.q;

```

pe când în clasa $A2$ membrul q este `public` (secvența urmatoare este corectă):

```

A2 a2;
int k = a2.q;

```

Observație. Dacă se dorește ca anumiți membri publici din clasa de bază care au devenit privați printr-o moștenire de tip privat să fie publici în clasa derivată, numele lor trebuie specificat după cuvântul cheie `public` (doar numele, nu și tipul de date). De exemplu, dacă se dorește ca în clasa $A1$ membrul q să fie `public`, declarația clasei $A1$ trebuie scrisă astfel:

```

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

```

În acest mod, membrul q este vizibil și în exteriorul clasei $A1$.

8.2 Constructori și destructori în ierarhii de clase

Constructorii și destructorii sunt singurii membri ai unei clase care nu pot fi moșteniți într-o clasă derivată (în afară de operatorul de atribuire, despre care se va discuta într-un capitol ulterior). Acest lucru este natural, deoarece operațiile de creare și distrugere a obiectelor unei clase sunt proprii clasei respective și 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 instanță a clasei de bază, constructorul pentru obiectul clasei derivate trebuie să apeleze mai întâi constructorul obiectului instanță al clasei de bază. Apelul constructorului obiectului ascuns se poate face explicit într-o listă de inițializare a constructorului obiectului clasei derivate, sau se poate genera implicit de compilator în cazul în care clasa de bază nu are declarați constructori.

Apelul constructorului clasei de bază în lista de inițializare a constructorului clasei derivate se face prin numele clasei de bază, deoarece obiectul ascuns corespunzător 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 = a; }
    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 cunoaște 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 și toate clasele de bază au constructori explițiți 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 destructorilor 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 moștenirii 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);
    // ...
}

```

Ieșirea 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();
}

```

```
}
```

Ieșirea 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* și *Membru* au fost apelați implicit (liniile 5 și 6), precum și pseudo-constructorul de copiere pentru membrul *n* (în linia 10, valoarea componentei *n* a obiectului *o2* este tot 7, ca și cea din obiectul *o1*).

În cazul în care se dorește 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 ieșire 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) { }
```

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

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

8.3 Moștenirea publică și privată

Utilizarea tipului de moștenire publică sau privată are o semnificație distinctă în proiectarea și dezvoltarea unor aplicații.

A. Moștenirea publică

Moștenirea publică este legată în special de etapa de proiectare a unei aplicații, în cazul în care se dorește 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 relație este denumit “este-o” sau “este-un”.

Acest tip de moștenire presupune moștenirea într-o clasă derivată, atât a interfeței clasei de bază, cât și 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* și *Cerc*. Funcția *Distanța* determină distanța între două puncte:

```
double Distanța (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 secvență este corectă și afișează valorile 1 și 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 = Distanța(p1, p2);
double d2 = Distanța(c1, c2);
cout << d1 << d2 << endl;
// ...
```

Datorită faptului că o clasă derivată public moștenește atât declarația cât și 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 același 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 dorește utilizarea membrului ascuns, acesta trebuie prefixat de numele clasei de bază.

Observație: 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 dorește acest lucru, vor trebui utilizate *funcții virtuale*.

Exemplul 8.8. Deși majoritatea păsărilor zboară, există și păsări care nu zboară; de exemplu pinguinul. Exemplul următor reprezintă o proiectare greșită a ierarhiei de clase :

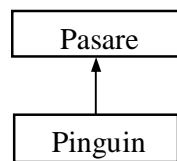


Figura 8.3

```

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 funcției *zboara* în clasa *pinguin*:

```

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

```

```

    }
    // ...
};

// ...
penguin penguin;
penguin.zboara(); //corect

```

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

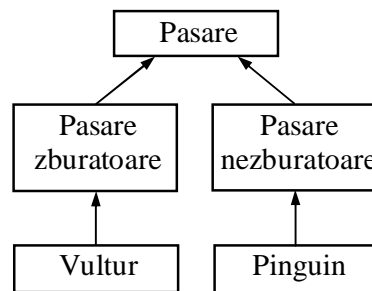


Figura 8.4

```

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 penguin: public PasareNezburatoare {
    //...
};

// ...
Vultur vultur;
Pinguin penguin;
Vultur.zboara();

```

B. Moștenirea private

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

În cazul moștenirii private, toți membrii clasei de bază devin membri private în clasa derivată, ceea ce îi face să nu poată fi utilizați în exteriorul clasei derivate (cu excepția celor private din clasa de bază, care devin inaccesibili în clasa derivată). Cu alte cuvinte, în acest caz o clasă derivată moștenește doar implementarea clasei de bază, nu și interfața acesteia. Din acest motiv, acest tip de relație se mai numește în literatura de specialitate “este_o_implementare_a_lui”.

În cazul moștenirii 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* și *Cerc*:

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

Observație. Același tip de relație “este_o_implementare_a_lui” o reprezintă și compunerea obiectelor, deoarece pentru un membru obiect într-o clasă compusă se moștenește doar implementarea, nu și interfața acesteia. În mod uzual este indicată utilizarea compunerii obiectelor în acest caz, ori de câte ori este posibil. Utilizarea moștenirii private este necesară doar în cazurile în care clasa de bază conține membrii de tip `protected`, care altfel nu pot fi utilizați în clasa derivată.

Exemplul 8.10. Să presupunem că se dorește 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 poziție corespunzătoare, astfel încât lista să rămână ordonată și după inserare (metoda sortării prin inserare). Având la dispoziție clasa *list*, definită în capitolul 5, există două variante: derivarea clasei *ListaOrdonata* din clasa *list*, sau definirea clasei *ListaOrdonata* ca o clasă compusă ce conține un obiect de tipul *list*. Derivarea publică nu este indicată, deoarece în acest mod se va moșteni și interfața clasei *list* și deci vor fi vizibile toate funcțiile de adăugare a unui element în listă, ceea ce face ca o listă ordonată să poată deveni neordonată prin apelul funcțiilor *AddFirst* sau *AddLast*.

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

C. Conversia de tip între clasele de bază și clasele derivate în cazul moștenirii publice

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

Observații:

1. Operația de atribuire copiază doar membrii din clasa de bază;
2. Atribuirea poate funcționa doar într-o singură direcție: *nu se poate atribui* un obiect al unei clase de bază unui obiect al unei clase derivate. De exemplu, instrucțiunea următoare este greșită:

Regula compatibilității la atribuire se extinde și la pointeri și referințe de obiecte. De exemplu, următoarea secvență 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 implicită 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 referință) la o clasă derivată spre un pointer (sau o referință) la clasa de bază este numit **upcasting**. Denumirea vine de la conversia implicită de tip (`cast`) și 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.

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

De exemplu, în cazul moștenirii publice între clasele *Cerc* și *Punct*, utilizarea funcției *Distanța* se bazează pe upcasting:

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

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

8.4 Moștenirea multiplă

Nu toate limbajele care suportă paradigma OOP acceptă **moștenirea multiplă**. De exemplu, limbajul Smalltalk, care a fost limbajul de referință până la apariția limbajului C++ nu acceptă acest tip de moștenire. 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 și ierarhii de clase proprii, independente de o anumită ierarhie predefinită.

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

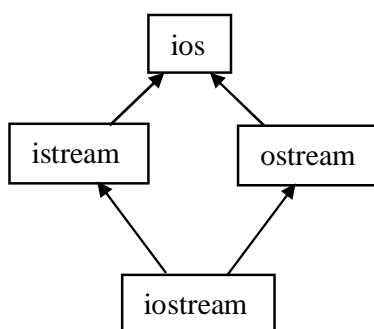


Figura 8.5

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

Duplicarea obiectelor ascunse poate apare în cazul ierarhiilor care au o formă ca în figura precedentă (numite și 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 M1: public M1, public M2 {
    int m;
```

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

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

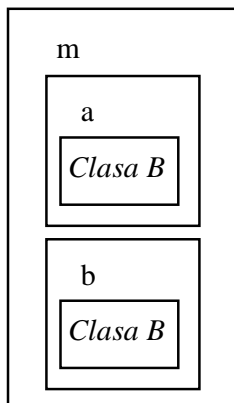


Figura 8.6

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 moștenirii multiple este însă generată de cazul în care ***mai multe clase de bază posedă un membru cu același nume***: cum vor putea fi utilizați acești 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 același 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 definește o clasă *CercText* care permite afisarea unui text în interiorul unui cerc (nu s-au mai specificat implementările 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 Translatate(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 dreptunghi
```

```

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 funcția *Show* a fost moștenită în clasa *CercText* și ascunde funcțiile cu același nume din clasele *Cerc* și *Mesaj*, pe care le utilizează cu ajutorul operatorului de rezoluție.

În ierarhia de clase anterioară, moștenirea multiplă nu este neapărat necesară. O alternativă mai aproape de semnificația reală, poate considera clasa *CercText* ca o clasă compusă ce conține în interior două obiecte private, instanțe ale claselor *Cerc* și *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 moștenirii multiple poate apare atunci când o clasă derivată necesită controlul asupra claselor de bază (prin utilizarea mecanismului funcțiilor 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 și fișiere 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 aplicație, astfel încât să memoreze obiecte aparținând unei alte clase, *Figura*, la care nu se poate avea acces, se poate crea o clasă nouă, *FiguraDerivata*, obținută prin derivarea din clasele *Object* și *Figura*, după cum se observă din figura 8.7.

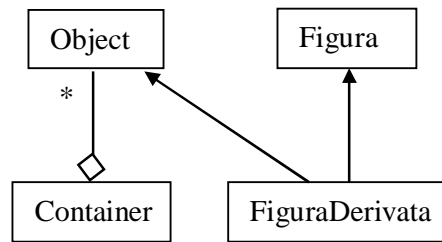


Figura 8.7

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