Capitolul 12

10 Polimorfism şi funcţii virtuale

După cum s-a observat anterior, în cazul derivării claselor există posibilitatea redefinirii unor funcţii membre ale claselor de bază. Operaţia de redefinire a unei funcţii cu acelaşi nume, dar cu parametri diferiţi şi cu efect diferit, poartă numele de ***supraâncarcare***.

În afară de operaţia de supraâncarcare, limbajul C++ pune la dispoziţie încă o metodă de definire a unor funcţii cu acelaşi nume într-o ierarhie de clase: metoda ***funcţiilor virtuale***. Avantajul utilizării funcţiilor virtuale în locul celor supraîncărcate este foarte important: în cazul în care se utilizează un pointer la o clasă de bază pentru a adresa un obiect al unei clase derivate public, compilatorul poate determina exact funcţiile care trebuie apelate cunoscând tipul obiectului curent spre care indică pointerul (chiar dacă el aparţine unei clase de bază).

Diferenţa între cele două metode este semnificativă, dar în esenţă ea se referă la momentul în care se face legarea funcţiilor. În cazul funcţiilor supraâncărcate legarea funcţiilor este statică, în sensul că un compilator poate determina funcţia ce se va utiliza în momentul compilării programului; în cazul funcţiilor virtuale legarea este dinamică, determinarea funcţiei utilizate se face doar la execuţia programului.

Obiectele care aparţin unor clase ce conţin funcţii virtuale se numesc ***polimorfice***. Ele au aceeaşi faţă determinată de interfaţa clasei de bază, dar mai multe forme, determinate de diferitele funcţii virtuale. Un limbaj care permite utilizarea obiectelor polimorfice se spune că suportă noţiunea de ***polimorfism***. Obiectele polimorfice şi noţiunea de ***polimorfism*** reprezintă caracteristica principală a paradigmei programării orientate pe obiecte.

Functiile virtuale sunt legate strict de noţiunea de polimorfism şi moştenire, pe când supraâncarcarea funcţiilor nu este întotdeauna legată de moştenire. Un exemplu îl constituie supraâncarcarea funcţiilor operator.

Polimorfismul reprezintă elementul distinctiv al prográmarii orientate pe obiecte, iar funcţiile virtuale metode de implementare in limbajul C++. Din acest motiv, programatorul care utilizează doar derivarea claselor şi mecanismul moştenirii înseamnă că nu a ajuns la un stadiu suficient de stăpânire al limbajului C++.

10.1 Funcţii supraîncărcate şi funcţii virtuale

Utilizarea moştenirii şi a supraâncărcării funcţiilor membru reprezintă o metodă de generare a obiectelor polimorfice. În cazul în care se utilizează obiecte şi nu adrese de memorie ale unor obiecte (pointeri sau referinţe), apelul funcţiilor supraâncarcate se efectuează în condiţii de siguranţă.

Exemplu:

```
class punct {
public:
  double x, y;
```

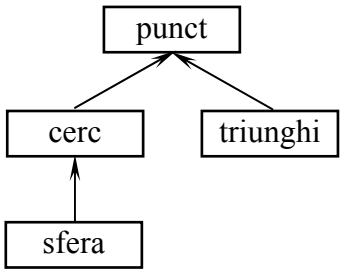
```
  punct (double x0 = 0, double y0 = 0) { x = x0; y = y0; }
  double Aria() const { return 0.0; }
};

class cerc: public punct {
public:
  double r;
  cerc(double x0 = 0, double y0 = 0,
      double r0=1): punct(x0, y0), r(r0) {}
  double Aria() const { return 3.1415*r*r; }
};

void main () {
  punct p(1, 1);
  cerc c(1, 1, 1);
  cout << "Arie punct= " << p.Aria() << endl;
  cout << "Arie cerc= " << c.Aria() << endl;
}
```

Ieşirea programului anterior este corectă, se afişează valorile 0 şi 3.1415 .

O metodă des utilizată în apelul metodelor obiectelor instanţă al unor clase dintr-o ierarhie de clase este cea numită ***upcasting*** şi constă în utilizarea unor pointeri spre obiecte ale claselor de bază, atât pentru obiecte din clasele de bază, cât şi pentru obiecte ale claselor derivate (care sunt considerate subtipuri ale claselor de bază).

Avantajul folosirii metodei `upcasting` se referă la economia de memorie ce se poate realiza, cât mai ales la posibilitatea definirii unor containere eterogene ce conţin obiecte ale unor clase diferite, derivate dintr-o clasă de bază unică.

Exemplu: Să considerăm ierarhia de clase:




```
class punct {
  // ca inainte
};

class cerc: public punct {
  // ca inainte
};

class sfera: public cerc {
public:
  sfera(double x0, double y0, double r0):
      cerc(x0, y0, r0) { }
  double Aria () const { return 4*3.1415*r*r; }
```

```
};

class triunghi: public punct {
public:
  double x1, y1;
  double x2, y2;
  triunghi(double x00, double y00, double x01,
      double y01, double x02, double y02):
           punct(x00, y00), x1(x01), y1(y01),
           x2(x02), y2(y02) { }
  double Aria () const
      { return (x*y1+x1*y2+x2*y-y*x1-y1*x2-y2*x)/2; }
};

void main () {
  punct *v[] = { new cerc(1, 1, 1), new sfera(1, 1, 1),
      new triunghi(0, 0, 1, 0, 0, 1) };
  cout << v[0]->Aria() << endl;
  cout << v[1]->Aria() << endl;
  cout << v[2]->Aria() << endl;
}
```

În exemplul precedent se pot trata uniform obiecte ale ierarhiei respective. În acest caz, al încercării utilizării metodei `upcasting`, apare o problemă: care dintre funcțiile supraâncarcate se va executa în cazul apelului prin intermediul adresei de memorie?

Răspunsul este următorul: în toate cele trei cazuri se va apela funcția `Arie` din clasa `punct`, deşi cele trei obiecte aparțin celorlalte clase.

O asemenea *problemă* apare datorită modului în care compilatorul determină funcția ce trebuie apelată. Acțiunea se numeşte ***legarea apelului funcțiilor*** şi presupune asocierea corpului funcției ce se va executa la prototipul acestora. În cazul în care nu se specifică altfel, legarea funcțiilor pentru apel este ***statică*** şi se realizează în timpul compilării. Întrucât pointerii pot indica spre obiecte diferite în timpul execuției programului, pentru siguranța mecanismului de apel se consideră că obiectele spre care indică aceştia aparțin clasei specificate la declararea pointerilor (clasa *punct* în exemplul anterior).

Soluția acestei probleme este ***legarea dinamică*** a apelului funcțiilor, legare ce se realizează în timpul execuției programului şi nu la compilare. Mecanismul de legare dinamică variaza la diferitele limbaje de programare, dar în general el se referă la inserarea unui anumit tip de informație suplimentară în cadrul obiectelor instanță.

În cazul limbajului C++, forțarea legării dinamice a apelului funcțiilor se realizează cu ajutorul cuvantului cheie `virtual`, specificat în declarația clasei din care face parte funcția pentru care se doreşte o legare dinamică. Din acest motiv, funcțiile pentru care se realizează legarea dinamică a apelului se numesc ***funcții virtuale***.

Din punct de vedere sintactic, cuvantul cheie `virtual` precede declarația unei funcții şi nu se repetă la definirea ei în exteriorul clasei (cu excepția funcțiilor declarate `inline` în interiorul claselor). O funcție declarată virtuală într-o clasă de bază îşi păstrează această proprietate pentru toate clasele derivate, astfel încât cuvantul cheie `virtual` nu mai trebuie specificat la eventuala redefinire a unei asemenea funcții într-o clasă derivata.

Pentru ca programul anterior să funcţioneze corect, trebuie doar ca funcţia *Arie* din cadrul clasei *punct* să fie declarată virtuală:

```
class punct {
public:
  double x, y;
  punct(double x0 = 0, double y0 = 0): x(x0), y(y0) { }
  virtual double Aria () { return 0.0; }
};
```

Astfel programul va afişa ariile corecte ale celor trei obiecte.

Folosirea funcţiilor virtuale este avantajoasă din cel putin două puncte de vedere: asigură corectitudinea metodei `upcasting` de tratare uniformă a obiectelor instanţă şi permite extensibilitatea unei aplicaţii create pe baza unei ierarhii existente.

De exemplu, dacă la ierarhia precedentă s-ar mai adăuga clasa :

```
class piramida:public triunghi {
public:
  double x3,y3;
  piramida(double x00, double y00, double x01, double y01,
      double x02, double y02, double x03, double y03):
      triunghi(x00, y00, x01, y01, x02, y02),
      x3(x03), y3(y03) { }
  double Aria () const; // calculeaza aria fetelor piramidei
};
```

precum şi funcţia:

```
double PretDeCost(punct *p, double pret_unitar) {
  return p->Arie() * pret_unitar;
}
```

apelul funcţiei *PretDeCost* ar fi corect atât pentru adrese ale unor obiecte din clasele anterioare, cât şi din clasa *piramida*:

```
punct *v[]={ new cerc(1, 1, 1), new sfera(1,1,1),
      new triunghi(0, 0, 1, 0, 0, 1),
      new piramida(0, 0, 0, 1, 1, 0) };
cout << PretDeCost(v[0], 10) << endl;
cout << PretDeCost(v[1], 10) << endl;
cout << PretDeCost(v[2], 10) << endl;
cout << PretDeCost(v[3], 10) << endl;
// ...
```

În afara de mecanismul de legare a apelului funcţiilor, între funcţiile supraâncarcate şi funcţiile virtuale există o importantă deosebire impusă de prototipul unic al funcţiilor virtuale: funcţiile supraâncarcate pot avea parametrii diferiţi ca număr şi ca tip de date, pe când funcţiile virtuale **trebuie** să aibă acelaşi număr şi tip de argumente explicite (în afară de argumentul ascuns `this`).

Din acest motiv se poate comite usor greşeala de a scrie o funcţie supraâncarcata în locul uneia virtuale. Redefinirea unei funcţii virtuale într-o clasă derivată se numeşte în mod uzual ***suprascriere***.

Exemplu:

```
class A {
  virtual int f(int k) { return k; }
};

class B: public A {
  // functie supraancarcata
  int f(unsigned int k) { return k+1; }
};

class C: public A {
  //functie virtuala suprascrisa
  int f(int k) { return k+2; }
};

void main () {
  A *p1 = new B;
  A *p2 = new C;
  cout << p1->f(7) << endl;
  cout << p2->f(7) << endl;
  B b;
  cout << b.f(7) << endl;
}
```

Funcţia *f* a clasei *B* este supraâncarcată deoarece parametrul *k* nu corespunde ca tip de data cu funcţia virtuală a clasei de bază, astfel încât se va afişa 7 şi 9 la primele două apeluri, iar la ultimul apel valoarea 8.

Observaţie: În cazul apelului funcţiei *f* prin intermediul pointerului *p1*, compilatorul alege funcţia virtuală din clasa *A*, deoarece clasa *B* nu are nici o funcţie virtuală. În cazul apelului prin intermediul obiectului *b*, se va selecta funcţia supraâncarcată *f* din clasa *B* deoarece în acest caz nu se utilizează tehnica `upcasting`.

10.2 Implementarea legării dinamice a apelului funcţiilor

Pentru determinarea funcţiei adecvate ce va fi apelată în cazul legării dinamice, majoritatea compilatoarelor utilizează un tablou numit *VFTABLE* pentru acele clase care conţin funcţii virtuale. În plus, există un pointer ascuns, numit *VFPTR* ce indică spre tabloul *VFTABLE* asociat fiecărui obiect din clasa respectivă. Deşi reprezintă o metodă foarte utilizată, metoda tabelelor funcţiilor virtuale este în general dependentă de implementare.

Dupa cum s-a precizat, un obiect instanţă al unei clase are o zonă de memorie asociată datelor membru ale clasei. Pentru clasele cu funcţii virtuale există în plus un singur pointer la tabela *VFTABLE*.

Exemplu:

```
class A { // fara funcţii virtuale
  int k;
public:
  void f() const {}
  int g() const { return k; }
};

class B { // cu o functie virtuala
  int k;
public:
  void f() const {}
  virtual int g() const { return k; }
};

class C { // cu două functii virtuale
  int k;
public:
  virtual void f() const {}
  virtual int g() const { return k; }
};

void main() {
  A a;
  B b;
  C c;
  int x = sizeof(void *);
  int y = sizeof(a);
  int z = sizeof(b);
  int u = sizeof(c);
  cout << x << endl
      << y << endl
      << z << endl
      << u << endl;
}
```

Din paragraful precedent se observă următoarea relaţie:

```
u = z = x + y
```

Cu alte cuvinte, în cazul claselor ce conţin funcţii virtuale, sau care sunt derivate din clase ce conţin funcţii virtuale, la secţiunea de date a obiectelor instanţă se adaugă o componentă de tip `void*`.

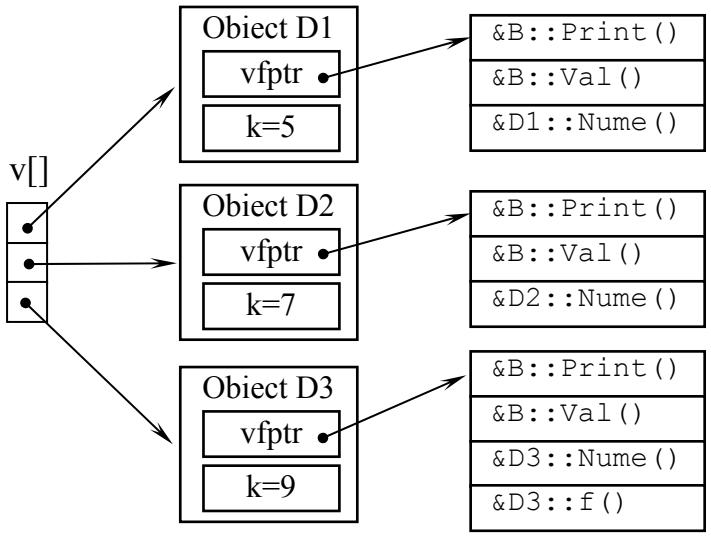
Generarea tabloului VFTABLE pentru aceste clase, precum şi inserarea şi initializarea pointerului VFPTR pentru obiectele acestor clase se realizează în mod automat de către compilator.

Pentru a înţelege mai bine modul de realizare a legării dinamice a apelului funcţiilor să considerăm urmatoarea secvenţă de program:

```
class B {
  int a;
public:
  B(int n=0): a(n) {}
  virtual void Print() const { cout << a << endl; }
```

```
  virtual int Val() const { return a; }
  virtual void Nume() const { cout << ”B” << endl; }
};

class D1: public B {
public:
  D1(int n): B(n) {}
  void Nume() const { cout << ”D1” << endl; }
};

class D2: public B {
public:
  void Nume() const { cout << ”D2” << endl; }
  D2(int n): B(n) {}
};

class D3: public B {
public:
  void Nume() const { cout << ”D3” << endl; }
  D3(int n): B(n) {}
  virtual void f() const {}
};

B *v[] = { new D1(5), new D2(7), new D3(9) };
```

Descrierea grafică a tabloului V este prezentată în figura următoare:



Se observă faptul că pentru fiecare clasă ce conține cel putin o funcție virtuală, sau în cazul în care clasa este derivată dintr-o altă clasă ce conține funcții virtuale, compilatorul creează în mod unic câte o tabelă VFTABLE. În acest tablou se află adresele funcțiilor virtuale proprii clasei şi eventual moştenite, în ordinea în care apar în declarația clasei. În cazul în care o clasă moşteneste o funcție virtuală de la o clasă de bază şi nu o redefinește, adresa trecută în tabela VFTABLE este cea a funcției din clasa în care a fost definită, pe aceeaşi pozitie ca şi în tabela clasei de bază. Pointerul VFPTR din cadrul fiecarui obiect instanță al unei asemenea clase va indica întotdeauna spre prima componentă a tabloului VFTABLE al clasei asociate.

Observație: În mod uzual clasa de bază oferă prin intermediul funcțiilor virtuale o interfață pentru clasele din cadrul unei anumite ierarhii, iar clasele derivate nu adaugă funcții virtuale noi. În cazul în care se întâmplă acest lucru şi se utilizează un pointer la clasa de bază pentru apelul unei funcții virtuale care există doar într-o clasă derivată, apelul funcției respective trebuie prefixat cu un operator `cast`. De exemplu, pentru secvența anterioară, apelul următor este corect:

```
(D3*)v[2]->f()
```

pe când apelul:

```
v[2]->f()
```

este incorect.

Să considerăm acum următoarea secvență:

```
void main() {
  B *p = v[2];
  p->Nume();
  // ...
}
```

Secvența de cod inserată de compilator la apelul `p->Nume()` diferă de cazul în care nu ar exista funcții virtuale. În cazul legării statice al apelului funcțiilor s-ar apela funcția aflată la adresa absolută `&B::Nume()`, ceea ce ar produce afisarea sirului `"B"`.

În cazul legării dinamice (al prezenței funcțiilor virtuale), se va apela funcția aflată la adresa absolută VFPTR+2 (deoarece funcția *Nume* se află pe pozitia a treia în şirul funcțiilor virtuale), adică funcția de la adresa `&D2::Nume()` şi va avea ca efect afisarea sirului `"D2"`. Deoarece adresa obiectului v[2] şi implicit a componentei VFPTR şi a valorii aflate la adresa VFPTR+2 sunt determinate doar în momentul execuției programului, legarea apelului funcției este de tip dinamic în acest caz.

În mod uzual, codul generat de compilator pentru apelul unei funcții virtuale prin intermediul unui pointer *p* arată în modul următor:

```
(*p->vfptr[k])();
```

unde *k* este indicele asociat în mod unic poziției funcției virtuale respective în tabela *VFTABLE*.

Inițializarea lui VFPTR se face în cadrul constructorului fiecărei clase, indiferent dacă este un constructor explicit sau unul implicit, deoarece operația de initializare este realizată în mod automat de către compilator.

Spe deosebire de alte limbaje de programare orientate pe obiecte precum Smalltalk sau Java, în care legarea dinamică a apelului funcțiilor este intrinsecă, în cazul limbajului C++ este posibil ca în anumite cazuri pentru funcțiile virtuale legarea să fie statică.

În cazul în care se foloseşte tehnica `upcasting` (adică se utilizează adrese de memorie şi nu obiecte propriu-zise), pentru funcțiile virtuale legarea este întotdeauna dinamică, deoarece compilatorul nu cunoaşte suficiente informații pentru a genera secvența de apel. În cazul însă în care se apelează o funcție virtuală prin intermediul unui obiect (şi nu al unui pointer sau referințe), compilatorul posedă suficiente informații pentru generarea secvenței de apel chiar

în etapa de compilare. În mod uzual în acest caz legarea va fi statică (pentru cele mai multe compilatoare C++).

De exemplu, pentru ierarhia de clase precedentă să considerăm următoarea secvență:

```
void prelucrare() {
  D1 o1;
  B *p = &o1;
  B &q = o1;
  D2 o2;
  p->Nume();     // legarea dinamica
  q.Nume();      // legare dinamica
  o2.Nume();     // legare statica
  // ...
}
```

În primele două cazuri legarea este dinamică, pentru că se lucrează cu adrese de obiecte, pe când în ultimul caz legarea este statică.

Motivul pentru care limbajul C++ utilizează legarea statică pentru funcțiile virtuale apelate prin intermediul obiectelor o constituie eficiența. Codul generat pentru apelul funcțiilor prin legarea dinamică este semnificativ mai mare decât cel generat în cazul legării statice.

Datorită faptului că pentru funcțiile virtuale trebuie trecută în tabela *VFTABLE* adresa de memorie a codului funcțiilor unde se va face saltul la apelul acestora, funcțiile virtuale nu pot fi `inline`, deoarece în acest caz compilatorul nu execută un salt obișnuit în momentul apelului. Chiar dacă există funcții virtuale declarate `inline`, compilatorul le va trata drept funcții obișnuite.

10.3 Clase abstracte şi funcții virtuale pure

În majoritatea cazurilor în etapa de proiectare a unei ierarhii de clase se doreşte ca clasa de bază a ierarhiei (în special în cazul moştenirii simple) să ofere doar o interfață pentru clasele derivate. Aceasta înseamnă că o astfel de clasă trebuie să fie foarte generală, o clasă care să nu permită crearea de obiecte instanță ale sale, ci să fie folosită pentru operația de `upcasting`. O asemenea clasă se numeşte ***clasă abstractă*** şi se poate realiza prin intermediul ***funcțiilor virtuale pure***.

O funcție virtuală pură este o funcție virtuală la care antetul este urmat de "`= 0`". În mod uzual o asemenea funcție nu are o implementare în clasa de bază, neforțând astfel corpul funcției să aibă o anumită semnificație. În acelaşi timp însă se impune ca în clasele derivate să existe definiții pentru funcția respectivă.

Deşi nu se pot crea obiecte instanță ale unei clase abstracte, se pot utiliza pointerii sau referințe la o asemenea clasă. Aceasta este şi metoda uzuală de folosire, în care prin `upcasting` se referă obiecte ale claselor derivate plecând de la pointeri ai clasei de bază.

Exemplu:

```
#include <string>
```

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

class valoare {
public:
  virtual void Print() const = 0;
};

class intreg: public valoare {
public:
  int n;
  intreg(int k = 0): n(k) {}
  void Print() const { cout << n << endl; }
};

class real: public valoare {
public:
  double x;
  real(double v = 0.0): x(v) {}
  void Print() const { cout << x << endl; }
};

class sir: public valoare {
public:
  char *s;
  sir(char *str) {
    s = new char[strlen(str)];
    strcpy(s, str);
  }
  void Print() const { cout << s << endl; }
};

void main() {
  valoare *v[] = { new intreg(5), new real(7.5),
      new sir("exemplu")};
  for (int i=0; i<3; i++)
    v[i]->Print();
}
```

În cazul în care o funcție este declarată ca fiind virtuală pură, compilatorul rezervă o intrare în tabela VFTABLE a clasei respective, dar nu scrie nici o valoare în acea locație. În acest fel, tabela VFTABLE pentru o clasă abstractă este incompletă şi nu se pot crea obiecte instanță ale clasei.

Limbajul C++ permite ca funcțiile virtuale pure să posede şi anumite definiții într-o clasă de bază. Acestea se numesc ***definiții virtuale pure*** şi nu reprezintă corpul propriu-zis al funcției virtuale. Motivul permiterii definițiilor virtual pure este cel al simplificării scrierii programelor: în cazul în care două sau mai multe implementări ale unei funcții virtuale pure în clase derivate conțin o parte de cod comun, este mai simplu ca partea respectivă de cod să se scrie o singură dată în clasa de bază ca o definiție virtuală pură. Datorită mecanismului de moştenire, definițiile virtuale pure sunt moştenite în clasele derivate şi pot fi utilizate în cadrul implementarilor funcției virtuale pure.

Exemplu: Modificarea exemplului precedent, în care funcție virtuală pură *Print* din clasa *valoare* poseda o definiție virtuală pură:

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

class valoare {
public:
  virtual void Print() const = 0;
};

void valoare::Print() const { cout << "Valoarea este: "; }

class intreg: public valoare {
public:
  int n;
  intreg(int k = 0): n(k) {}
  void Print() const {
    valoare::Print();
    cout << n << endl;
  }
};

class real: public valoare {
public:
  double x;
  real(double v): x(v) {}
  void Print() const { valoare::Print(); cout << x << endl; }
};

class sir: public valoare {
public:
  char *s;
  sir(char *str) {
    s = new char[strlen(str)];
    str copy(s, str);
  }
  void Print() const { valoare::Print(); cout << s << endl; }
};
```

Un alt avantaj al definițiilor virtuale pure constă în faptul că se poate transfera o funcție virtuală obişnuită într-o funcție virtuală pură fără să se modifice partea de program deja scrisa, doar prin simpla inserare a caracterelor "= 0" în prototipul funcției respective din clasa de bază.

10.4 Funcții virtuale, constructori şi destructori virtuali

Dupa cum s-a precizat deja, constructorii nu pot fi moşteniți într-o ierarhie de clase. Din acest motiv nu se justifică eventuala lor declarare ca funcții virtuale. Rolul constructorilor este însă foarte important, mai ales în cazul claselor ce conțin funcții virtuale. În mod uzual, la începutul unui asemenea constructor compilatorul inserează cod pentru inițializarea

membrului ascuns VFPTR a obiectului curent la adresa primei componente a tabelei VFTABLE.

Deşi constructorii nu pot fi virtuali, teoretic este posibil ca în interiorul unui constructor să se apeleze o funcție virtuală. Din punct de vedere practic această variantă nu este indicată, deoarece în acest caz mecanismul legării dinamice nu va mai funcționa şi se va selecta pentru execuție întotdeauna versiunea locală din clasa curentă a funcției virtuale respective.

Există două motive importante pentru acest mod de tratare. În primul rând, în timpul execuției unui constructor, obiectul asociat este doar parțial construit, fără să se poată cunoaşte ce obiecte sunt derivate din obiectul instanță al clasei de bază. Dacă s-ar efectua legarea dinamică a unei funcții virtuale din interiorul constructorului, aceasta ar presupune că se vor utiliza membrii care nu sunt initializați încă (care se află la alt nivel al ierarhiei, posibil mai jos), ceea ce este o sursă de potențiale noi.

Al doilea motiv se referă la suita de apeluri succesive de constructori în cazul unui obiect derivat. Fiecare constructor apelat utilizează pointerul VFPTR spre tabela VFTABLE proprie clasei, astfel încât doar ultimul constructor apelat generează complet tabela VFTABLE asociată. Deoarece la utilizarea legării dinamice a apelului funcțiilor se utilizează tabela VFTABLE pentru determinarea funcției apelate, se va utiliza doar tabela proprie clasei şi nu ultima determinată, care este valabilă doar dupa apelul ultimului constructor (constructorii sunt apelați întotdeauna în ordinea: clasa de bază, prima clasă derivata, a două clasă derivata, s.a.m.d.).

Spre deosebire de constructori, destructorii pot fi virtuali, iar în anumite cazuri este chiar indicat să fie virtuali.

Ordinea de apel a destructorilor este inversă celei pentru constructori: de la clasa derivată de pe nivelul cel mai de jos al ierarhiei de clase, spre clasa de bază a acesteia.

În cazul în care nu se utilizează pointeri şi operatorul `delete`, nu este necesar ca destructorii să fie virtuali. Dacă însă se apelează operatorul `delete` asupra unui pointer la o clasă de bază şi nu există destructori virtuali, compilatorul nu va poseda suficiente informații asupra obiectului spre care indică pointerul şi va apela destructorul din clasa de bază.

Exemplu:

```
class A {
public:
  ~A() { cout << ”Destructorul clasei A” << endl; }
};

class B: public A {
public:
  ~B() { cout << ”Destructorul clasei B” << endl; }
};

void main() {
  A *p = new B;
  delete p;
  {
     B b;
```

```
  }
}
```

Din ieşirea programului anterior se observă faptul că pentru pointerul *p* se apelează destructorul clasei *A*, pe când pentru obiectul *b* se apelează destructorul clasei *B* (în ambele cazuri se apelează constructorul clasei B).

Apelul destructorului din clasa de bază poate cauza diferite erori într-un program dacă obiectul a fost creat ca instanță dintr-o altă clasă, iar pentru anumite date membru nu se eliberează memoria (în exemplul anterior pentru pointerul *p* se apelează constructorul din clasa *B* şi destructorul din clasa *A*).

Soluția la această problemă constă în declararea destructorului din clasa de bază ca virtual. Pentru programul anterior, corect este astfel:

Exemplu:

```
class A {
public:
  virtual ~A() { cout << ”Destructorul clasei A” << endl; }
};

class B: public A {
public:
  ~B() { cout << ”Destructorul clasei B” << endl; }
};
```

În acest caz mecanismul legării dinamice funcționează (destructorii sunt tot funcții) şi în instrucțiunea `delete p;` se apelează corect destructorul pentru clasa *B* (in plus se va apela şi destructorul clasei *A*, care este clasa de bază pentru *B*).

Spre deosebire de constructori, în cazul destructorilor se cunoaşte tipul obiectului indicat de un pointer pentru că obiectul a fost deja construit iar pointerul VFPTR inițializat, astfel încât mecanismul legării dinamice poate avea loc.

În afara de destructori virtuali, limbajul C++ permite şi ***destructori virtuali puri***. În comparație cu celelelte funcții virtuale pure, destructorii virtuali puri au câteva caracteristici specifice:

- pentru aceştia este obligatoriu să se specifice şi corpul lor în clasa de bază;
- in clasele derivate nu este obligatoriu să se inlocuiască corpul acestor destructori din clasa de bază.

De fapt, singura diferență între un destructor virtual şi un destructor virtual pur este doar în cazul în care clasa de bază nu are alte funcții virtuale pure şi destructorul virtual pur o face să fie calsă abstractă, nepermițând astfel instanțe de obiecte pentru clasa respectivă. Acesta este de fapt şi scopul utilizarii destructorilor virtuali puri.

Exemplu: Programul urmator se comportă ca şi cel precedent:

```
class A {
public:
  virtual ~A() = 0;
```

```
};

A::~A() { cout << ”Destructorul clasei A” << endl; }

class B: public A {
public:
  ~B() { cout << ”Destructorul clasei B” << endl; }
};

void main() {
  A *p = new B;
  delete p;
}
```

În concluzie:

- destructorii virtuali se utilizează fie în cazul în care clasa respectivă conține cel putin o funcție virtuală, fie în cazul în care se utilizează pointeri şi mecanismul `upcasting;`
- destructorii virtuali puri se utilizează în cazul în care se doreşte ca o anumită clasă de bază să fie clasă abstractă şi ea nu conține alte funcții virtuale pure.

Ca şi în cazul constructorilor, dar din motive diferite, mecanismul legării dinamice pentru apelul funcțiilor din cadrul destructorilor nu este efectuat de către compilator, astfel încât la apelul unei funcții virtuale în corpul unui destructor se selectează pentru execuție varianta locala a funcției din cadrul clasei.

În cazul în care s-ar efectua legarea dinamică a funcțiilor apelate dintr-un constructor, este posibil ca să se apeleze o funcție a unui obiect deja distrus, datorită ordinii de apel al destructorilor (de la clasa derivată spre cea de bază).

Exemplu:

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

class A {
public:
  virtual ~A() {
    cout<< ”Destructorul clasei A” << endl;
    f();
  }
  virtual void f() { cout << ”functia f în clasa A” << endl; }
};

class B: public A {
public:
  ~B() { cout << ”Destructorul clasei B” << endl; }
  void f() {cout << ”functia f în clasa B” << endl; }
};

void main() {
  A *p = new B;
  delete p;
```

```
}
```

Se observă că se apelează funcția *f* din clasa de bază *A*, desi *f* este virtuală şi redefinită în clasa *B*. Motivul pentru acest comportament al compilatorului este acela că dacă s-ar efectua legarea dinamică, ar trebui apelată o funcție membru al unui obiect care a fost deja distrus. În exemplul precedent se apelează destructorul pentru clasa *B*, apoi destructorul pentru clasa *A*, iar din acest moment nu se mai poate apela funcția *f* din clasa *B* (obiectul a fost deja distrus) şi se apelează funcția locală din clasa *A*.

10.5 Exemplu de ierarhie de clase cu o singura radacină

Una dintre problemele întâlnite în cadrul containerelor este "problema proprietarului": determinarea clasei care este responsabilă pentru distrugerea obiectelor dinamice prin intermediul operatorului `delete`. În mod uzual, pentru pentru a spori flexibilitatea astfel încât containerul să poată conține obiecte de tipuri diferite, obiectele sunt specificate prin pointeri la `void`. Apelul unui operator `delete` asupra unui pointer la `void` nu apelează însă destructorul unei anumite clase, astfel încât containerul trebuie să fie responsabil cu operația de distrugere a obiectelor conținute.

O soluție des utilizată în aplicații o constituie o ierarhie de clase cu o singură radacină.

În exemplul următor se va defini o stivă ca un container ce poate conține obiecte derivate dintr-o clasă generală numită *object*.

```
//fisierul stiva.h
#ifndef STACK_H
#define STACK_H

class object {
public:
  virtual ~object() = 0;
};

inline object::~object() {}

class stack {
  struct Link {
    object *data;
    Link *next;
    Link(object* o, Link* n): data(o), next(n) {}
  } *top;
public:
  stack(): top(0) {}
  ~stack() {
    while(top)
      delete pop();
  }
  void push(object* o) {
    top = new Link(o, top);
  }
  object *pop() {
    if (top == 0) return 0;
```

```
    object *tmp = top->data;
    Link *l = top;
    top = top->next;
    delete l;
    return tmp;
  }
};

#endif
```

Destructorul clasei *object* şi funcţia *pop* au fost definite `inline` pentru simplitate, astfel încât să fie păstrate toate definiţiile în fisierul header `stiva.h`.

Stiva definită anterior este foarte flexibilă, deoarece elementele sale memorează pointeri la *object*. Singura restricţie este ca obiectele propriu-zise păstrate în stiva să fie instanţe ale unor clase derivate din clasa *object*. Dacă se doreşte ca elementele din stivă să aparţină unei anumite clase, metoda aceasta necesita moştenirea multiplă (din clasa *object* şi din clasa respectivă).

De exemplu, clasa *MyString* utilizează altă clasa predefinita `string`, cât şi clasa *object*:

```
#include "stiva.h"
#include <string>
using namespace std;

class MyString: public string, public object {
public:
  MyString(char *s): string(s) {}
};

void main() {
  stack st;
  st.push(new MyString("item 1"));
  st.push(new MyString("item 2"));
  st.push(new MyString("item 3"));
  MyString *s;
  for(int i=0; i<3; i++) {
    s = (MyString*)st.pop();
    cout << *s << endl;
  }
}
```

Deoarece clasa *stack* cunoaşte tipul de obiecte conţinute, se apelează destructorul clasei corespunzătoare (aici clasa *MyString*) la apelul operatorului `delete`.

10.6 Moştenirea multiplă şi clase de bază virtuale

În paragraful 7.4 s-au specificat principalele probleme care pot fi generate de moştenirea multiplă: duplicarea obiectelor ascunse şi existenţa unor funcţii din clase de bază diferite care au acelaşi nume. Dacă ultima problemă are o rezolvare simplă prin redefinirea funcţiilor în clasa derivată, prima problemă poate conduce la situaţii speciale în cazul utilizării metodei `upcasting`.

În cazul moştenirii multiple, un obiect instanţă al unei clase derivate din mai multe clase de bază posedă mai mulţi pointeri `this`, câte unul pentru subobiect al unei clase de bază.

Exemplu.

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

class B1 {
public:
  void Print1() const
    { cout << "B1: this= " << this << endl; }
};

class B2 {
public:
  void Print2() const
    { cout << "B2: this= " << this << endl; }
};

class M: public B1, public B2 {
public:
  void Print() const {
    cout << "M: this= " << this << endl;
    Print1();
    Print2();
  }
};

void main() {
  M m;
  m.Print();
  B1* b1 = &m;
  B2* b2 = &m;
  cout << "B1: pointer= " << b1 << endl;
  cout << "B2: pointer= " << b2 << endl;
}
```

Se observă faptul că valorile lui *b1* şi *b2* sunt diferite şi ele corespund adreselor obiectelor ascunse ale claselor *B1* şi *B2*.

În cazul în care între două clase dintr-o ierarhie (una derivată şi una de bază) există mai multe drumuri, înseamnă că toate obiectele clasei derivate vor conţine mai multe subobiecte ascunse ale clasei de bază: câte unul pentru fiecare drum care leagă clasa drerivată de cea de bază.

Din motivul prezentat mai sus, în cazul în care se utilizează `upcasting`, compilatorul va inidica o eroare la încercarea de a converti adresa obiectului derivat în adresa obiectului de bază.

Exemplu.

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

class B {
public:
  virtual ~B() {}
  virtual char* f() { return "D1"; }
};

class D1 : public B {
public:
  char* f() { return "D1"; }
};

class D2 : public B {
public:
  char* f() { return "D2"; }
};

class M : public D1, public D2 {
public:
  char* f() { return D1::f(); }
};

void main() {
  B* b[3];
  b[0]= new D1;
  b[1] = new D2;
  b[2] = new M;     // !! Eroare: conversie ambigua
  for(int i = 0; i < 3; i++)
    cout << b[i]->f() << endl;
}
```

O soluție la această problemă constă în eliminarea subobiectelor duplicate. Această operație se realizează cu ajutorul unei extensii a mecanismului de derivare, numită ***derivarea virtuală***. Clasa de bază din care se derivează virtual o clasă derivată se numeşte ***clasă de bază virtuală***.

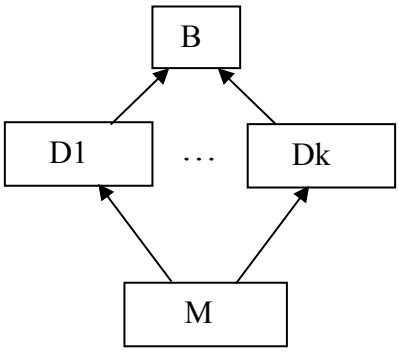
Din punct de vedere sintactic, derivarea virtuală se specifică prin cuvântul cheie `virtual`, care prefixează numele clasei de bază. De exemplu, pentru clasele *D1* şi *D2* precedente, definiția lor corectă este:

```
class D1 : virtual public B {
public:
  char* f() { return "D1"; }
};

class D2 : virtual public B {
```

```
public:
  char* f() { return "D2"; }
};
```

Clasa *B* devine clasă de bază virtuală, iar cu această modificare, programul anterior funcționează corect.

Pentru o clasă derivată virtual dintr-o clasă de bază, doar un subiect al său va fi memorat de clasele derivate din clasa respectivă. Pentru ca acest mecanism să funcționeze, trebuie ca toate clasele *D1*, *D2*, …, *Dk*, derivate dintr-o clasă de bază virtuală *B*, ce pot fi clase de bază pentru o altă clasă derivată *M*, să fie derivate virtual din clasa de bază *B*.



Astfel, deoarece obiectele instanță ale clasei *M* vor conține un singur subobiect al clasei *B*, nu va mai exista nici o confuzie în cazul conversiei `upcasting`.

Datorită faptului că în clasa derivată *M* va exista un singur subobiect al clasei de bază virtuale *B*, constructorii claselor derivate *D1*, *D2*, …, *Dk*, nu vor mai apela constructorul clasei de bază *B* în momentul în care aceştia sunt apelaţi în cadrul constructorului clasei *M*. În acest caz, apelul constructorului clasei de bază virtuale se va face în fiecare clasă derivată din aceasta.

Exemplu. Reluarea exemplului precedent la care s-a mai adăugat o clasă.

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

class B {
public:
  B(int) {}
  virtual char* f() const = 0;
  virtual ~B() {}
};

class D1 : virtual public B {
public:
  D1() : B(1) {}
  char* f() const { return "D1"; }
};

class D2 : virtual public B {
public:
```

```
  D2() : B(2) {}
  char* f() const { return "D2"; }
};

class M : public D1, public D2 {
public:
  M() : B(3) {}
  char* f() const {
    return D1::f();
  }
};

class X : public M {
public:
  // Trebuie intotdeauna initializata clasa de baza virtuala
  X() : B(4) {}
};

void main() {
  B* b[4];
  b[0]= new D1;
  b[1] = new D2;
  b[2] = new M;
  b[3] = new X;
  for(int i = 0; i < 4; i++)
    cout << b[i]->f() << endl;
}
```

Se observă din exemplul precedent faptul că apelul constructorului clasei *B* s-a efectuat în toate clasele derivate din *B*, chiar şi din clasa *X*, care moşteneşte în mod indirect pe *B*.

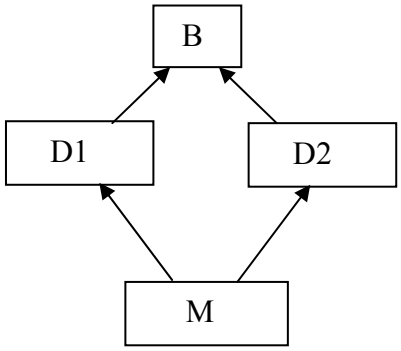
Observație. În cazul în care clasa de bază virtuală *B* posedă un constructor implicit, constructorul implicit al unei clase derivate dintr-o clasă derivată virtual din *B* nu mai trebuie să conțină un apel explicit al constructorului implicit al lui *B*, deoarece el este generat automat de către compilator. Rezultă de aici o simplificare a definirii unei asemenea ierarhii.

Exemplu. Rescrierea ierarhiei precedente, la care s-au utilizat constructori impliciți.

```
class B {
public:
  B(int = 0) {}
  virtual char* f() const = 0;
  virtual ~B() {}
};

class D1 : virtual public B {
public:
  D1() : B(1) {}    // Trebuie apelat constructorul lui B
  char* f() const { return "D1"; }
};
```

```
class D2 : virtual public B {
public:
  D2() : B(2) {}    // Trebuie apelat constructorul lui B
  char* f() const { return "D2"; }
};

class M : public D1, public D2 {
public:
  M() {}  // Constructorul lui B este apelat implicit
  char* f() const {
    return D1::f();
  }
};

class X : public M {
public:
  X() {}  // Constructorul lui B este apelat implicit
};
```

Pentru o mai bună înțelegere a ordinii apelului constructorilor, în continuare se prezintă pentru ierarhia din figura următoare, ordinea de apel a contructorilor. Pentru simplitate s-au notat cu *B1()* şi *B2()* apelul constructorului clasei *B* din cadrul constructorilor claselor *D1*, respectiv *D2*, iar cu *B()* apelul constructorului lui *B* din cadrul constructorului clasei *M*.



`class D1: public B`
`class D2: public B` (există două subobiecte ale lui B)
`1) B1(), D1(), B2(), Dd2()`

`class D1: public B`
`class D2: virtual public B` (există tot două subobiecte ale lui B)
`1) Base(), B1(), D1(), D2()`

`class D1: virtual public B`
`class D2: public B` (există tot două subobiecte ale lui B)
`1) B(), D1(), B2(), D2()`

`class D1: virtual public B`
`class D2: virtual public B` (există un subobiect al lui B)
`1) B(), D1(), Dd2()`

Pentru a gestiona corect obiectele instanță, în cazul derivate din clasele de bază virtuale se utilizează pointeri suplimentari de tipul *VFPTR*.

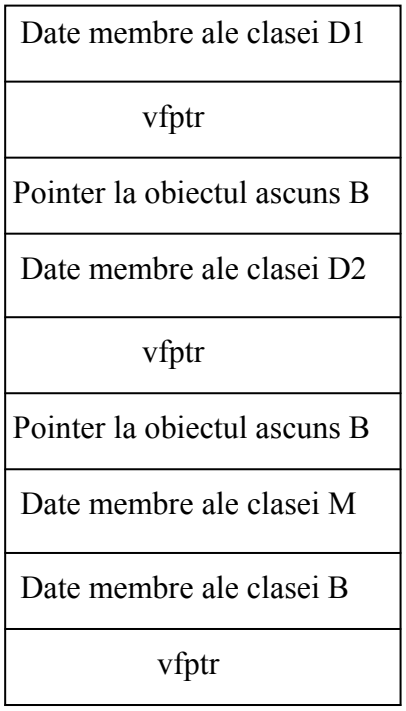
Exemplu. Pentru o anumită versiune a compilatorului Visual C++ al firmei Microsoft, programul următor afişează următoarele valori: `4, 4, 44, 48, 48, 48, 48, 64.`

```
class B {
public:
     int v[10];
     virtual ~B() {}
};

class D1: virtual public B { };

class D2: virtual public B { };

class D3: virtual public B { };

class D4: virtual public B { };

class M: public D1, public D2, public D3, public D4 {
public:
     int n;
};

void main() {
     B b;
     D1 d1;
     D2 d2;
     D3 d3;
     D4 d4;
     M m;
     cout << "sizeof(int)= " << sizeof(int) << endl;
     cout << "sizeof(void*)= " << sizeof(void*) << endl;
     cout << "sizeof(B)= " << sizeof(b) << endl;
     cout << "sizeof(D1)= " << sizeof(d1) << endl;
     cout << "sizeof(D2)= " << sizeof(d2) << endl;
     cout << "sizeof(D3)= " << sizeof(d3) << endl;
     cout << "sizeof(D4)= " << sizeof(d4) << endl;
     cout << "sizeof(M)= " << sizeof(m) << endl;
}
```

Se observă faptul că pentru obiectele clasei *B* există un singur pointer suplimentar VFPTR (operație normală, deoarece există funcții virtuale), pe când pentru obiectele claselor *D1*, *D2*, *D3* şi *D4* există doi pointeri suplimentari, iar pentru obiectele clasei *M* există 5 pointeri suplimentari. Însă în cadrul obiectului *m*, există un singur subobiect ascuns al clasei *B*.

În mod uzual, în afară de pointerul *vfptr* asociat clasei curente, mai există pentru fiecare clasă derivată din clasa de bază un pointer *vfptr* asociat clasei derivate respective, preum şi un pointer asociat obiectului ascuns în clasa derivată respectivă. De exemplu, pentru ierarhia

din figura precedentă, structura unui obiect al clasei *M* este următoarea (în mod uzual ea este dependentă de implementare):



10.7 Operația `downcasting` şi informații RTTI

După cum s-a observat, metoda upcasting este, cu unele excepții în cazul moştenirii multiple, o metodă sigură de conversie de jos în sus între clasele unei ierarhii, utilizând pointeri la obiecte ale clasei de bază. În acest caz se cunoaşte în mod unic clasa de bază şi clasa derivată:

```
class B { ... };
class D1: public B { ... };
class D2: public B { ... };
B* p = new D2;
```

Problema inversă, de transformare a unui pointer sau a unei referințe de la o clasă de bază spre una derivtă se numeşte ***downcasting*** şi nu este întotdeauna sigură:

1. În primul rând, plecând de la o clasă de bază, pot exista mai multe clase derivate unde se poate ajunge, astfel încât trebuie specificat un operator cast explicit;
2. În al doilea rând, este posibil ca drumul de întoarcere pentru o operație `downcasting` să fie altul decât cel asociat operației `upcasting` inițiale.

Exemplu. Pentru ierarhia şi oparația `upcasting` următoare:

```
class B { ... };
class D1: public B { ... };
class D2: public B { ... };
B* pb = new D1;     // drum: D1* -> B*
```

Următoarea operație `downcasting` este o eroare:

```
D2* pd2 = (D2*)pb;  // !!Eroare. Drum: B* -> D2*
```

Limbajul C++ posedă pentru această operație un operator cast explicit, numit `dynamic_cast`. Acesta încearcă să efectueze o schimbare a tipului de date de la un pointer sau o referință la o clasă de bază spre un pointer sau o referință la o clasă derivată, iar în caz în care operația eşuează, returnează zero.

Exemplu.

```
class B {
  virtual ~B() {}
};
class D1: public B { ... };
class D2: public B { ... };
void main() {
  B* pb = new D1;
  D1 *pd1;
  D2 *pd2;
  pd1 = dynamic_cast<D1*>pb;  // O.K.
  pd2 = dynamic_cast<D2*>pb;  // pd2 = 0
}
```

Operatorul `dynamic_cast` se poate utiliza doar pentru ierarhiile ce conțin funcții virtuale, deoarece utilizează tabela VFTABLE pentru a determina dacă tipul destinație al conversiei este corect.

În timpul execuției programului, fiecărui obiect nou creat I se pot asocia anumite informații suplimentare referitoare la tipul clasei din care face parte. Aceste informații sunt memorate sub forma unor obiecte de tipul `type_info` (declarația acestei clase se află în fişierul header `typeinfo.h`). Informațiile de acest tip se numesc informații ***RTTI*** (run-time type information) şi sunt de tip dinamic (se completează în timpul execuției programelor, nu în timpul compilării). Structura clasei poate diferi la anumite compilatoare, însă majoritatea utilizează următoarele date:

```
class type_info {
public:
  virtual ~type_info();
  int operator==(const type_info& rhs) const;
  int operator!=(const type_info& rhs) const;
  const char* name() const;
  const char* raw_name() const;
private:
  void *_m_data;
  char _m_d_name[1];
  type_info(const type_info& rhs);
  type_info& operator=(const type_info& rhs);
};
```

Funcția membru `name` returnează şirul reprezentând numele clasei la care aparține obiectul curent.

În mod uzual, când există mai multe clase derivate dintr-o clasă de bază, trebuie testate toate variantele posibile de conversie `downcasting` pentru a determina varianta corectă (pentru care operatorul `dynamic_cast` returnează un pointer nenul). Aceasta poate constitui şi un avantaj, deoarece se poate determina natura exactă a obiectului respectiv..

Exemplu. Programul următor contorizează diferitele tipuri de obiecte dintr-o ierarhie cu o singură rădăcină, stocate într-un tablou de pointeri la clasa de bază.

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

class B {
// ...
};

class D1: public B {
// ...
};

class D2: public B {
// ...
};

class D3: public B {
// ...
};

B* v[] = { new D1, new D2, new D3, new D3 };

void main() {
  int n1 = 0, n2 = 0, n3 = 0;
  for (int k=0; k<4; k++)
  {
    if (dynamic_cast<D1*>(v[k]))
      n1++;    // Obiect de tipul D1
    if (dynamic_cast<D1*>(v[k]))
      n2++;    // Obiect de tipul D2
    if (dynamic_cast<D1*>(v[k]))
      n3++;    // Obiect de tipul D3
  }
  cout << "n1= " << n1 << endl;
  cout << "n2= " << n2 << endl;
  cout << "n3= " << n3 << endl;
}
```

A doua metodă prin care se pot utiliza în cadrul programelor informațiile RTTI o reprezintă ***operatorul typeid***. În acest caz trebuie inclus fişierul header `typeid.h` în fişierele sursă respective.

Operatorul `typeid` are un argument care poate fi un obiect instanță al unei clase, sau un pointer sau o referință spre o clasă şi returnează un obiect constant de tipul `typeinfo` asociat tipului respectiv. După cum se observă din definiția clasei `typeinfo`, un asemenea obiect poate fi comparat cu alte obiecte, sau se poate extrage din el numele tipului de date asociat.

În mod uzual, operatorul `typeid` este utilizat în cazul claselor ce conțin funcții virtuale.

Există o deosebire în cazul utilizării argumentelor de tip pointer şi referință pentru operatorul `typeid`. Pentru exemplificare, se consideră următoarea ierarhie:

```
class B {
public:
  virtual ~B() {}
};

class D: public B {
// ...
};
```

precum şi declarațiile:

```
B* pb = new D;
B& rb = *pb;
```

Următoarele aserțiuni sunt adevărate:

```
assert ( typeid(rb) == typeid(D) );
assert ( typeid(rb) != typeid(B) );
assert ( typeid(rb) != typeid(D*) );
assert ( typeid(rb) != typeid(B*) );
```

Se observă faptul că tipul unei referințe este clasa derivată (*D* în exemplul precedent) şi nu un pointer la ea (*D** de exemplu).

În cazul pointerilor avem:

```
assert ( typeid(pb) == typeid(B*) );
assert ( typeid(pb) != typeid(D*) );
assert ( typeid(pb) != typeid(D) );
assert ( typeid(pb) != typeid(B) );
```

În acest caz, tipul pointerului *pb* este *B**, adică cel rezultat din declarația lui *pb*. Pe de altă parte, tipul obiectului indicat de *pb* este cel al clasei *D* (ca şi în cazul refernței):

```
assert ( typeid(*pb) == typeid(D) );
```

```
assert ( typeid(*pb) != typeid(B) );
assert ( typeid(*pb) != typeid(D*) );
assert ( typeid(*pb) != typeid(B*) );
```

Operatorul `typeid` poate fi utilizat şi pentru clase nepolimorfice, dar în acest caz el utilizează informații de tip static, nu RTTI.

Exemplu. Următoarele aserțiuni sunt adevărate:

```
class A {
// nu exista functii virtuale
};

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

void main() {
  A* p = new B;
  assert(typeid(*p) == typeid(A));
  assert(typeid(*p) != typeid(B));
}
```

De asemenea, operatorul `typeid` poate fi aplicat ;I asupra unor expresii apar'in\nd unor tipuri predefinite. De exemplu:

```
assert ( typeid(15) == typeid(int) );
assert ( typeid(35.78) == typeid(double) );
int n;
assert ( typeid(n) == typeid(D) );
assert ( typeid(&n) != typeid(int*) );
```

10.8 Supraâncarcarea operatorilor virtuali

Pentru a putea trata corect operatorii în cazul conversiei upcasting, este necesar ca aceştia să fie definiți virtuali în clasa de bază a ierarhiei.

Exemplu. Se consideră cazul tratării uniforme a valorilor întregi, reale şi complexe. Pentru exemplificare se vor supraâncărca doar operatorul de adunare.

```
class valoare {
public:
  virtual ~valoare() {}
  virtual valoare& operator+(valoare&) = 0;
  virtual valoare& operator=(valoare&) = 0;
  virtual void Print() const = 0;
};

class intreg: public valoare {
```

```
  int n;
public:
  intreg(int k = 0): n(k) {}
  int N() const { return n; }
  valoare& operator+(valoare& v) {
    n += (dynamic_cast<intreg&>(v)).n;
    return *this;
  }
  valoare& operator=(valoare& v) {
    n = (dynamic_cast<intreg&>(v)).n;
    return *this;
  }
  void Print() const { cout << n << endl; }
};

class real: public valoare {
  double x;
public:
  complex(double a = 0): x(a) {}
  double X() const { return x; }
  valoare& operator+(valoare& v) {
    x += (dynamic_cast<real&>(v)).x;
    return *this;
  }
  valoare& operator=(valoare& v) {
    x = (dynamic_cast<real&>(v)).x;
    return *this;
  }
  void Print() const { cout << x << endl; }
};

class complex: public valoare {
  double re, im;
public:
  complex(double a = 0, double b = 0): re(a), im(b) {}
  double Re() const { return re; }
  double Im() const { return im; }
  valoare& operator+(valoare& v) {
    re += (dynamic_cast<complex&>(v)).re;
    im += (dynamic_cast<complex&>(v)).im;
    return *this;
  }
  valoare& operator=(valoare& v) {
    re = (dynamic_cast<complex&>(v)).re;
    im = (dynamic_cast<complex&>(v)).im;
    return *this;
  }
  void Print() const { cout << re << im << endl; }
};

void main() {
```

```
  valoare* v1[] = { new intreg(2), new intreg,
          new intreg(9), new intreg(5) };
  valoare* v2[] = { new complex(2, 2), new complex,
          new complex(1, 5), new complex(2, 4) };
  intreg s1;
  for (int k=0; k<4; k++) {
    valoare& a = *v1[k];
    s1 = s1 + a;
  }
  complex s2;
  for (k=0; k<4; k++) {
    valoare& a = *v2[k];
    s2 = s2 + a;
  }
  s1.Print();
  s2.Print();
}
```

Se observă o deficiență a programului precedent: el nu poate trata cazul în care operanzii au tipuri diferite (de exemplu, întreg şi complex). În cadrul fiecărui operator se presupune că ambii operanzi au acelaşi tip de date, pentru că altfel operatorul `dynamic_cast` nu ar funcționa corect.

Pentru a trata corect problema supraîncărcării operatorilor binari, în cazul în care operanzii au tipuri diferite, va trebui introdus un ***al doilea nivel de tratare***, deoarece mecanismul funcțiilor virtuale nu poate trata decât un singur parametru.

Al doilea nivel de tratare se poate implementa în mod uzual cu ajutorul unei funcții suplimentare, care are rolul de a permuta locul operanzilor (oferind astfel şi celui de-al doilea operand posibilitatea de a fi tratat de un operator virtual).

Exemplu. Se va rescrie ierarhia precedentă.

```
class valoare {
public:
  virtual ~valoare() {}
  virtual valoare& operator+(valoare&) = 0;
  virtual valoare& operator=(valoare&) = 0;
  virtual void Print() const = 0;
};

class intreg: public valoare {
  int n;
public:
  intreg(int k = 0): n(k) {}
  int N() const { return n; }
  // Al doilea nivel de tratare
  valoare& operator+(valoare& v) {
    return v.Suma(*this);
  }
  valoare& Suma(intreg* p) {
```

```
    n += p->n;
    return *this;
  }
  valoare& Suma(real* p) {
    return p->Suma(this);
  }
  valoare& Suma(complex* p) {
    return p->Suma(this);
  }
  valoare& operator=(valoare& v) {
    n = (dynamic_cast<intreg&>(v).n;
    return *this;
  }
  void Print() const { cout << n << endl; }
};

class real: public valoare {
  double x;
public:
  intreg(int a = 0): x(a) {}
  double X() const { return x; }
  // Al doilea nivel de tratare
  valoare& operator+(valoare& v) {
    return v.Suma(*this);
  }
  valoare& Suma(intreg* p) {
    x += p->N();
    return *this;
  }
  valoare& Suma(real* p) {
    x += p->x;
    return *this;
  }
  valoare& Suma(complex* p) {
    return p->Suma(this);
  }
  valoare& operator=(valoare& v) {
    x = (dynamic_cast<real&>(v).x;
    return *this;
  }
  void Print() const { cout << x << endl; }
};

class complex: public valoare {
  double re, im;
public:
  intreg(double a = 0, double b = 0): re(a), im(b) {}
  double Re() const { return re; }
  double Im() const { return im; }
  // Al doilea nivel de tratare
  valoare& operator+(valoare& v) {
```

```
    return v.Suma(*this);
  }
  valoare& Suma(intreg* p) {
    re += p->N();
    return *this;
  }
  valoare& Suma(real* p) {
    re += p->Re();
    return *this;
  }
  valoare& Suma(complex* p) {
    re += p->re;
    im += p->im;
    return *this;
  }
  valoare& operator=(valoare& v) {
    re = (dynamic_cast<complex&>(v).re;
    im = (dynamic_cast<complex&>(v).im;
    return *this;
  }
  void Print() const { cout << re << im << endl; }
};

void main() {
  valoare* v[] = { new intreg(2), new real,
          new real(5.5), new complex(1, 3) };
  complex s;
  for (int k=0; k<4; k++) {
    valoare& a = *v[k];
    s1 = s1 + a;
  }
  s.Print();
}
```

Observație. În afară de al doilea nivel de tratare, programul precedent mai realizează o oparație suplimentară: determinarea corectă a tipului de date al rezultatului operației asociate fiecărui operator. De exemplu, produsul dintre un număr real şi unul complex este un număr complex, deci rezultatul va trebui determinat în clasa *complex*, indiferent de ordinea operanzilor. Din acest motiv, o sumă de forma `r + c` (real + complex) utilizează două apeluri de funcții:

```
r.real::operator+(c)
c.complex::Suma(r)
```

pe când o sumă de forma `c + r` (complex + real) utilizează trei apeluri:

```
c.complex::operator+(r)
r.real::Suma(c)
c.complex::Suma(r)
```