9. FUNCȚII VIRTUALE

Importanța funcțiilor virtuale se datorează faptului că ele sunt folosite la implementarea obiectelor polimorfice, la momentul execuției lor. În C++, polimorfismul se realizează în două moduri. În primul mod, polimorfismul se realizează la momentul compilării programului, prin utilizarea *operatorilor* și *funcțiilor redefinite* (*overloading functions*). În al doilea mod, polimorfismul se realizează, la momentul execuției programului (*run-time*), prin utilizarea *funcțiilor virtuale*.
În acest capitol se vor aborda următoarele tematici: pointeri care punctează la funcții virtuale, declararea funcțiilor virtuale, greșeli care se comit la utilizarea funcțiilor virtuale, detalii de implementare a funcțiilor virtuale, polimorfismul.

9.1. Pointeri care punctează la clasele derivate

La baza funcțiilor virtuale și a polimorfismului ce se realizează la momentul execuției unui program stau variabilele de tip pointer care punctează la clasele derivate. Mecanismul este următorul: dacă într-o funcție (de exemplu, main()) sau într-o clasă de bază se declară un pointer care punctează la clasa de bază, acesta poate fi folosit pentru a puncta și la alte clase, derivate din clasa de bază.
Considerăm două clase, denumite BAZA și DERIVATA, unde DERIVATA moștenește clasa de bază, BAZA. În aceste condiții, următoarele instrucțiuni sunt valabile:

```
BAZA  * p;                  // p = pointerul care punctează la clasa de bază, BAZA
BAZA  OB_B;                 // OB_B este un obiect de tip BAZA
DERIVATA  OB_D;             // OB_D este un obiect de tip DERIVATA
// Este clar că pointerul p poate puncta la obiecte de tip BAZA
p = & OB_B;                 // p punctează la obiectul OB_B
// Pointerul p poate, de asemenea, puncta la obiecte de tip DERIVATA
p = & OB_D;                 // p punctează la obiectul OB_D
```

După cum se remarcă în ultimele linii de comentariu, un pointer la clasa de bază poate puncta la un obiect din clasa derivată, fară a genera vreo eroare. Deși acest lucru este valabil, putem avea acces doar la membrii clasei derivate, care au fost moșteniți de la clasa de bază, aceasta deoarece pointerul la clasa de bază nu "cunoaște" decât clasa de bază, neavând informații despre membrii clasei derivate. Inversul situației nu este valabil, adică, o variabilă de tip pointer, care punctează la clasa derivată, nu poate fi utilizat pentru a avea acces la un obiect al clasei de bază. Precizăm de asemenea că nu mai este valabilă nici aritmetica pointerilor ce punctează la clasa de bază.

Programul din exemplul următor arată modul în care poate fi utilizat un pointer al clasei de bază pentru a avea acces la o clasă derivată.

```
// Program P9_1.CPP   Pointeri care punctează la clasa de bază și clasa derivată

# include <iostream.h>

class BAZA {                    // Va fi utilizată pentru crearea clasei derivate, DERIVATA
        int x;                  // Variabilă de tip private
public:
        void init_x(int i) { x = i; }
        int obtine_x() {return x; }
};
class DERIVATA : public BAZA {
// Clasa DERIVATA moștenește toți membrii publici din clasa BAZA
        int  y;                 // Variabilă de tip private
public:
```

```
        void init_y(int i) { y = i; }
        int obtine_y() {return y; }
};
void  main (void)              // Programul principal
{   BAZA  * p;                 // Pointerul p punctează la clasa de bază, BAZA
    BAZA OB_B;                 // Se creează obiectul OB_B de tip BAZA
    DERIVATA OB_D;             // Se creează obiectul OB_D de tip DERIVATA
    p = & OB_B;                // Variabila p se utilizează pentru a avea acces la obiectul OB_B
    p->init_x(10);             // Se obtine acces la OB_B
    cout << "x din OB_B: " << p->obtine_x() << '\n';
    p = & OB_D;                // Variabila p se utilizează pentru a avea acces la obiectul OB_D
    //p->init_y(100);          // NU se poate utiliza pointerul p pentru a inițializa
                               // variabila y, astfel încât accesul se face direct
    OB_D.init_y(100);
    p->init_x(50);
    cout << "x din OB_B: " << p->obtine_x() << '\n';
    cout << "y din OB_D: " << OB_D.obtine_y() << '\n';
}
```

Prin execuția acestui program, pe ecran se vor afișa mesajele:

```
        x din OB_B: 10
        x din OB_B: 50
        y din OB_D: 100
```

Considerăm că nu mai sunt necesare alte comentarii, deoarece cele incluse în program sunt chiar exhaustive.

9.2. Declararea funcțiilor virtuale

O funcție virtuală este o funcție membră a unei clase, care se declară în interiorul unei clase de bază și se redefinește (modifică) într-o clasă derivată. Pentru a crea o funcție virtuală, se utilizează cuvântul cheie **virtual**, plasat înaintea declarației funcției. Când se moștenește o clasă care conține o funcție virtuală, clasa derivată redefinește funcția virtuală a clasei de bază. În esență, funcțiile virtuale implementează filosofia "*o singură interfață, mai multe metode*", care pune în evidență polimorfismul. Funcția virtuală din interiorul clasei de bază definește *forma interfeței* cu acea funcție. Fiecare redefinire a ei în clasa derivată implementează operațiile specifice clasei derivate, adică fiecare redefinire creează câte o *metodă specifică*. Când o funcție virtuală este modificată într-o clasă derivată, nu este necesară utilizarea cuvântului cheie **virtual**. Deși o funcție virtuală poate fi apelată ca orice funcție membră a unei clase, totuși apelul ei prezintă o particularitate când se folosește pentru aceasta un *pointer*. Se știe că un pointer la clasa de bază poate fi folosit pentru a puncta la un obiect al clasei derivate. *Când un pointer la clasa de bază punctează la o funcție virtuală din clasa derivată și aceasta este apelată prin intermediul acestui pointer, compilatorul, ținând cont de tipul obiectului la care punctează acel pointer, determină versiunea funcției trebuie apelată*. Deci, tipul obiectului la care punctează pointerul determină versiunea funcției virtuale ce va fi executată. Prin urmare, dacă două sau mai multe clase sunt derivate dintr-o clasă de bază care conține o funcție virtuală, atunci când un pointer la clasa de bază punctează la mai multe funcții virtuale redefinite în clasele derivate, vor fi executate diferitele versiuni ale acelei funcții. Acesta este de fapt procesul prin care se realizează polimorfismul la momentul execuției programului. Pentru fixarea acestor idei, în continuare, sunt prezentate câteva exemple:

1. Programul din listingul următor arată modul de utilizare a unei funcții virtuale

```
// Program P9_2a.CPP   Utilizarea unei funcții virtuale
# include <iostream.h>
class BAZA {                    // Va fi utilizată pentru crearea claselor derivate D1 și D2
public:
        int i;                  // Variabilă de tip public
        BAZA (int x) { i = x; }
        virtual void f_v ()
        {
                cout << "Utilizeaza versiunea f_v() din clasa BAZA: ";
                cout << i << '\n';
        }
};
class D1 : public BAZA {
// Clasa D1 moștenește toți membrii publici din clasa BAZA
public:
        D1 (int x) : BAZA (x)  { }
        void f_v ()
        {
                cout << "Utilizeaza versiunea f_v() din clasa D1: ";
                cout << i*i << '\n';
        }
};
class D2 : public BAZA {
// Clasa D2 moștenește toți membrii publici din clasa BAZA
public:
        D2 (int x) : BAZA (x)  { }
        void f_v ()
        {
                cout << "Utilizeaza versiunea f_v() din clasa D2: ";
                cout << i*i*i << '\n';
        }
};
// Programul principal
void  main (void)
{  BAZA  * p;                   // Pointerul p punctează la clasa de bază, BAZA
   BAZA OB_B (10);              // Se creează obiectul OB_B de tip BAZA
   D1  OB_D1 (10);              // Se creeaza obiectul OB_D1 de tip D1
   D2  OB_D2 (10);              // Se creeaza obiectul OB_D2 de tip D2
   p = & OB_B;                  // Se utilizează variabila p pentru a avea acces la obiectul OB_B
   p->f_v ();                   // Utilizează funcția f_v() din clasa BAZA
   p = & OB_D1;                 // Se utilizează variabila p pentru a avea acces la obiectul OB_D1
   p->f_v ();                   // Utilizează funcția f_v() din clasa D1
   p = & OB_D2;                 // Se utilizează variabila p pentru a avea acces la obiectul OB_D2
   p->f_v ();                   // Utilizează funcția f_v() din clasa D2
}
```

Acest program va afișa pe ecran următoarele:

```
        Utilizeaza versiunea f_v() din clasa BAZA: 10
```

Utilizeaza versiunea f_v() din clasa D1: 100
Utilizeaza versiunea f_v() din clasa D2: 1000

Redefinirea unei funcții virtuale în interiorul unei clase derivate pare similară cu redefinirea funcțiilor (*overloading functions*). Totuși, cele două procese sunt distincte. În primul rând, o funcție redefinită trebuie să difere prin tip și/sau prin număr de parametri. *O funcție virtuală redefinită trebuie să aibă același tip și număr de parametri și aceeași tip al valorii returnate*. De fapt, dacă la redefinirea unei funcții virtuale se schimbă fie numărul, fie tipul parametrilor, ea devine o funcție de tip oveloaded și natura sa virtuală se pierde. În plus, *funcțiile virtuale trebuie să fie funcții membre ale unei clase*, ceea ce nu este absolut necesar în cazul funcțiilor redefinite. Din cauza acestor diferențe, pentru a descrie redefinirea funcțiilor virtuale se folosește termenul *overriding* (suprapunere).
Menționam că *funcțiile destructor pot fi virtuale, pe când funcțiile constructor, nu*.

Programul de mai sus creează trei clase. Clasa de bază, BAZA, definește funcția virtuală f_v(). Clasa BAZA este moștenită atât de clasa D1, cât și de clasa D2. Fiecare dintre acestea redefinesc funcția f_v(), adaptând-o la specificul lor. În cadrul funcției main(), pointerul p este declarat de tip BAZA, dar este utilizat împreună cu obiecte de tip BAZA, D1 și D2. Mai întâi lui p i se atribuie adresa lui OB_B (un obiect de tip BAZA). Când funcția f_v() este apelată prin intermediul pointerului p, se va utiliza versiunea acesteia din clasa de bază. Apoi, lui p i se atribuie adresa obiectului OB_D1 (de tip D1) și este apelată din nou funcția f_v(). Întrucât tipul obiectului la care punctează pointerul determină care funcție virtuală trebuie folosită, de această dată este executată versiunea f_v() redefinită în clasa D1. În sfârșit, când lui p i se atribuie adresa obiectului OB_D2 și se apelează din nou funcția f_v(), va fi executată versiunea funcției definită în clasa D2.

2. Când se folosește mecanismul moștenirii, funcțiile virtuale respectă o ierarhie. Astfel, când o clasă derivată nu redefinește o funcție virtuală, este folosită implementarea definită în clasa de bază. Acest aspect se poate observa în listingul următor:

```
// Program P9_2b.CPP   Ierarhia funcțiilor virtuale

# include <iostream.h>

class BAZA {                    // Va fi utilizată pentru crearea claselor derivate D1 și D2
public:
        int i;                  // Variabilă de tip public
        BAZA (int x) { i = x; }
        virtual void f_v ()
        {
                cout << "Utilizeaza versiunea f_v() din clasa BAZA: ";
                cout << i << '\n';
        }
};

class D1 : public BAZA {
// Clasa D1 moștenește toți membrii publici din clasa BAZA
public:
        D1 (int x) : BAZA (x)  { }
        void f_v ()
        {
                cout << "Utilizeaza versiunea f_v() din clasa D1: ";
                cout << i*i << '\n';
        }
```

```
};
class D2 : public BAZA {
public:
        D2 (int x) : BAZA (x)  { }
        // Clasa D2  NU  redefinește funcția f_v ()
};
// Programul principal
void  main (void)
{  BAZA  * p;                       // Pointerul p punctează la clasa de bază, BAZA
   BAZA OB_B (10);                  // Se creează obiectul OB_B de tip BAZA
   D1  OB_D1 (10);                  // Se creează obiectul OB_D1 de tip D1
   D2  OB_D2 (10);                  // Se creează obiectul OB_D2 de tip D2
   p = & OB_B;              // Se utilizează variabila p pentru a avea acces la obiectul OB_B
   p->f_v ();                       // Utilizează funcția f_v() din clasa BAZA
   p = & OB_D1;             // Se utilizează variabila p pentru a avea acces la obiectul OB_D1
   p->f_v ();                       // Utilizează funcția f_v() din clasa D1
   p = & OB_D2;             // Se utilizează variabila p pentru a avea acces la obiectul OB_D2
   p->f_v ();                       // Utilizează funcția f_v() din clasa BAZA
}
```

Acest program va afișa pe ecran următoarele:

```
Utilizeaza versiunea f_v() din clasa BAZA: 10
Utilizeaza versiunea f_v() din clasa D1: 100
Utilizeaza versiunea f_v() din clasa BAZA: 10
```

În această versiune, clasa D2 nu redefinește funcția f_v(). Când pointerului p i se atribuie adresa lui OB_D2 și este apelată f_v(), se utilizează versiunea funcției din clasa de bază, deoarece aceasta se află pe un nivel superior în ierarhia claselor. În general, când o clasă derivată nu redefinește o funcție virtuală, se utilizează versiunea ei din clasa de bază.

3. Programul următor arată cum răspunde o funcție virtuală la evenimentele produse la momentul execuției; mai exact, el face o selecție între obiectele OB_D1 și OB_D2 în funcție de valoarea înnapoiată de generatorul de numere aleatoare rand():

```
// Program P9_2c.CPP   Utilizarea funcțiilor virtuale

# include <iostream.h>
# include <stdlib.h>

class BAZA {                    // Va fi utilizată pentru crearea claselor derivate D1 si D2
public:
        int i;                  // Variabilă de tip public
        BAZA (int x) { i = x; }
        virtual void f_v ()
        {
                cout << "Utilizeaza versiunea f_v() din clasa BAZA: ";
                cout << i << '\n';
        }
};
class D1 : public BAZA {
// Clasa D1 moștenește toți membrii publici din clasa BAZA
public:
```

```
        D1 (int x) : BAZA (x)  { }
        void f_v ()
        {
                cout << "Utilizeaza versiunea f_v() din clasa D1: ";
                cout << i*i << '\n';
        }
};
class D2 : public BAZA {
public:
        D2 (int x) : BAZA (x)  { }
        // Clasa D2 nu redefinește funcția f_v()
};
void  main (void)                       // Programul principal
{  BAZA  * p;                           // Pointerul p punctează la clasa de bază, BAZA
   D1  OB_D1 (10);                      // Se creează obiectul OB_D1 de tip D1
   D2  OB_D2 (10);                      // Se creează obiectul OB_D2 de tip D2
   int i, j;
   for ( i=0; i<10; i++) {
        j = rand();
        if (j % 2)   p = & OB_D1;       // Dacă j este impar, se utilizează obiectul OB_D1
        else   p = & OB_D2;             // Daca j este par, se utilizează obiectul OB_D2
        p->f_v ();                      // Se apelează functia f_v() corespunzătoare
   }
}
```

9.3. Funcții virtuale pure

Când o funcție virtuală definită într-o clasă de bază nu întreprinde nimic, trebuie ca funcțiile claselor derivate din clasa de bază să o redefinească. Pentru aceasta, C++ oferă *funcții virtuale pure*. O astfel de funcție nu este definită în clasa de bază, aceasta conținând doar prototipul ei. Sintaxa generală de declarare a unei funcții virtuale pure este:

virtual tip nume_functie (lista_de_parametri) = 0;

Inițializarea cu zero a acestei funcții spune compilatorului că în clasa de bază pentru această funcție nu există nici o definiție (un corp). *O funcție virtuală pură trebuie obligatoriu redefinită în clasele derivate*, altfel apare o eroare la compilare. Dacă o clasă conține cel puțin o funcție virtuală pură, ea se numește *clasă abstractă*. Întrucât o clasă abstractă conține cel puțin o funcție virtuală care nu prezintă corp propriu, ea este incompletă, și deci nu se pot crea obiecte din acea clasă. Prin urmare, clasele abstracte există doar pentru a fi moștenite. Totuși, trebuie avut în vedere faptul că se poate crea un pointer care să puncteze la o clasă abstractă, pointer prin intermediul căruia se implementează polimorfismul. Astfel, dacă o clasă derivată moștenește o funcție virtuală din clasa de bază și ea, la rândul ei, este folosită drept clasă de bază pentru o altă clasă ce va fi derivată din ea, funcția virtuală poate fi redefinită de clasa derivată finală, precum și de prima clasă derivată. Prezentăm, în acest sens, următoarele exemple:

1. Programul de mai jos generează o clasă generică denumită ARIA, care determină suprafața unor figuri plane, utilizând două dimensiuni specifice ale acestor figuri. De asemenea, se declară o funcție virtuală pură, obtine_aria(), care atunci când este redefinită în clasele derivate, înapoiază aria tipului de figură definită de clasa derivată. În acest exemplu, se calculează aria unui dreptunghi și aria unui triunghi dreptunghic.

```
// Program P9_3.CPP   Utilizarea funcțiilor virtuale pure

# include <iostream.h>
# include <stdlib.h>

class ARIA {                        // Va fi utilizată pentru crearea unor clase derivate
        double dim1, dim2;          // Dimensiunile figurii
public:
   void init_aria (double d1, double d2)          //
   { dim1 = d1; dim2 = d2; }
   void obtine_dim (double &d1, double &d2)       //
   { d1 = dim1; d2 = dim2; }
   virtual double obtine_aria() = 0;              // Funcție virtuală pure
};

class DREPTUNGHI : public ARIA {
// Clasa DREPTUNGHI moștenește toți membrii publici din clasa ARIA
public:
   double obtine_aria ()            // Prima redefinire a funcției virtuale pure, obtine_aria()
   {
        double d1, d2;
        obtine_dim (d1, d2);
        return d1*d2;
   }
};

class TRIUNGHI_D : public ARIA {
// Clasa TRIUNGHI_D moștenește toți membrii publici din clasa ARIA
public:
   double obtine_aria ()            // A doua redefinire a funcției virtuale pure, obtine_aria()
   {
        double d1, d2;
        obtine_dim (d1, d2);
        return 0.5*d1*d2;
   }
};

// Programul principal
void  main (void)
{  ARIA  * p;                       // Pointerul p punctează la clasa de bază, ARIA
   DREPTUNGHI  D;                   // Se creează obiectul D de tip DREPTUNGHI
   TRIUNGHI  T;                     // Se creează obiectul T de tip TRIUNGHI_D
   D.init_aria (3.0, 4.0);
   T.init_aria (3.0, 4.0);
   p = &D;
   cout << "Aria dreptunghiului este: " << p->obtine_aria() << '\n';
   p = &T;
   cout << "Aria triunghiului este: " << p->obtine_aria() << '\n';
}
```

Acest program garantează că fiecare clasă derivată va redefini funcția virtuală pură, obtine_aria().

2. Programul următor arată cum se menține natura virtuală a unei funcții, când se aplică mecanismul moștenirii multiple:

```
// Program P9_4.CPP  Utilizarea funcțiilor virtuale
// Funcțiile virtuale își păstrează natura virtuală când sunt moștenite

# include <iostream.h>

class BAZA {                    // Va fi utilizată pentru crearea claselor derivate DERIVATA1
                                // și DERIVATA2
public:
   virtual void f_v ()
   {
        cout << "Utilizeaza versiunea f_v() din clasa BAZA \n";
   }
};
class DERIVATA1 : public BAZA {
// Clasa DERIVATA1 moștenește toți membrii publici din clasa BAZA
public:
   void f_v ()
   {
        cout << "Utilizeaza versiunea f_v() din clasa DERIVATA1 \n";
   }
};
class DERIVATA2 : public DERIVATA1 {
// Clasa DERIVATA2 moștenește toți membrii publici din clasa DERIVATA1
public:
   void f_v ()
   {
        cout << "Utilizeaza versiunea f_v() din clasa DERIVATA2 \n";
    }
};

// Programul principal
void  main (void)
{  BAZA  * p;                   // Pointerul p punctează la clasa de bază, BAZA
   BAZA OB_B;                   // Se creează obiectul OB_B de tip BAZA
   DERIVATA1  OB_D1;            // Se creează obiectul OB_D1 de tip DERIVATA1
   DERIVATA2  OB_D2;            // Se creează obiectul OB_D2 de tip DERIVATA2
   p = &OB_B;
   p -> f_v();                  // Utilizează funcția f_v() din clasa BAZA
   p = &OB_D1;
   p -> f_v();                  // Utilizează funcția f_v() din clasa DERIVATA1
   p = &OB_D2;
   p -> f_v();                  // Utilizează funcția f_v() din clasa DERIVATA2
}
```

În acest program, funcția virtuală este moștenită mai întâi de clasa DERIVATA1, care o redefinește; apoi clasa DERIVATA2 moștenește clasa DERIVATA1 și în ea este redefinită din nou

funcția f_v(). Întrucât funcțiile virtuale sunt ierarhizate, dacă clasa DERIVATA2 n-ar fi redefinit funcția f_v(), atunci când aveam acces la obiectul OB_D2, ar fi trebuit utilizată funcția f_v() din clasa DERIVATA1. Dacă nici una din clasele derivate nu ar fi redefinit funcția f_v() din clasa de bază, toate referințele la ea ar fi fost direcționate la funcția definită în clasa de bază.

9.4. Polimorfismul

Așa cum s-a arătat, polimorfismul este procesul prin care o interfață comună se poate aplica la două situații similare (dar diferite din punct de vedere tehnic), implementându-se filosofia "*o singură interfață, metode multiple*". Polimorfismul este foarte important pentru că el poate simplifica, în mare măsură, sistemele complexe. Se folosește o singură interfață, bine definită, pentru a asigura accesul la acțiuni diferite, care sunt totuși în legătură. Când o interfață comună are acces la mai multe acțiuni, trebuie memorate mai puține lucruri. În mediile de programare orientate pe obiecte există doi termeni foarte importanți: *early binding* (*legare timpurie*) și *late binding* (*legare târzie*).
Primul termen se referă la acele apeluri de funcții care pot fi rezolvate în timpul compilării. Acestea includ: funcții "normale", funcții redefinite, funcții membre nevirtuale și funcții de tip **friend**. Când sunt compilate toate aceste funcții, informațiile referitoare la adresele apelurilor sunt cunoscute la momentul compilării. Principalul avantaj al acestui mecanism îl constituie faptul că *apelurile funcțiilor, legate la momentul compilării, reprezintă cele mai rapide tipuri de apeluri de funcții*. Dezavantajul îl constituie lipsa de flexibilitate a programelor.
Al doilea termen se referă la evenimentele care trebuie să se producă la momentul execuției programului. Într-un astfel de apel, adresa funcției care urmează a fi apelată nu este cunoscută până la execuția programului. În C++, *o funcție virtuală este o funcție care se leagă târziu*. Când un pointer la clasa de bază punctează la o funcție virtuală, programul este cel care va determina, la momentul execuției, la care tip de obiect punctează pointerul, după care va selecta versiunea corespunzătoare a funcției redefinite, care va trebui executată. Principalul avantaj al mecanismului *late binding* îl reprezintă flexibilitatea programului, dezavantajul reprezentându-l întârzierea produsă de apelul funcției, generându-se astfel execuții mai lente.
Programul din exemplul următor ilustrează filosofia "*o singură interfață, mai multe metode*". El implementează o clasă generică de tip "listă simplu înlănțuită" (*singly linked list*), care conține valori întregi. Programul declară mai întâi natura interfeței pentru o listă. Se definesc următoarele funcții: stocheaza(), pentru a memora o valoare, și regaseste() pentru a extrage o valoare din listă. Clasa de bază, LISTA, nu definește metode implicite pentru aceste acțiuni. În schimb, fiecare clasă derivată din clasa LISTA va implementa propria versiune de listă. În program vor fi create două tipuri de liste: o "coadă" (*queue*) și o "stivă" (*stack*). Deși, cele două tipuri de liste operează complet diferit, ele folosesc aceeași interfață.

```
// Program P9_5.CPP   Utilizarea funcțiilor virtuale pure
// Creează o clasă generică de tip lista cu întregi

# include <iostream.h>
# include <stdlib.h>
# include <ctype.h>

class LISTA {                  // Va fi utilizată pentru crearea claselor derivate COADA și STIVA
public:
   LISTA  * cap;               // Pointer care punctează la capul listei
   LISTA  * coada;             // Pointer care punctează la coada listei
   LISTA  * urm_art;           // Pointer care punctează la următorul articol din listă
   int  valoare;               // Valoarea care trebuie memorată
   LISTA () { cap = coada = urm_art = NULL; }
```

```
   virtual void stocheaza (int i) = 0;          // Funcție virtuală pură
   virtual int regaseste () = 0;                // Funcție virtuală pură
};

// Se creează o listă de tip "coadă" (queue)
class COADA : public LISTA {
// Clasa COADA moștenește toți membrii publici din clasa LISTA
public:
   void stocheaza (int i);
   int regaseste ();
};
void COADA :: stocheaza (int i)              // Definiția funcției stocheaza() din clasa COADA
{  LISTA * articol;
   articol = new COADA;
   if ( ! articol )  {
         cout << "Eroare de alocare ! \n";
         exit (1);
   }
   articol->valoare = i;                     // Pune articolul la coada listei
   if ( coada )  coada->urm_art = articol;
   coada = articol;
   articol->urm_art = NULL;
   if ( ! cap ) cap = coada;
}
int COADA :: regaseste ()                    // Definiția a funcției regaseste() din clasa COADA
{  int i;
   LISTA * p;
   if ( ! cap )  {
         cout << "Lista este vida ! \n";
         return 0;
   }
   // Se elimină articolul din capul listei
   i = cap->valoare;
   p = cap;
   cap = cap->urm_art;
   delete p;
   return i;
}

// Se creează o listă de tip "stivă" (stack)
class STIVA : public LISTA {
// Clasa STIVA moștenește toți membrii publici din clasa LISTA
public:
   void stocheaza (int i);
   int regaseste ();
};
void STIVA :: stocheaza (int i)              // Definiția funcției stocheaza() din clasa STIVA
{  LISTA * articol;
```

```
    articol = new STIVA;
    if ( ! articol )  {
          cout << "Eroare de alocare ! \n";
          exit (1);
    }
    articol->valoare = i;                  // Pune articolul în capul listei pentru operații de tip stivă
    if ( coada )  coada->urm_art = articol;
    if ( cap ) articol->urm_art = cap;
    cap = articol;
    if ( ! coada ) coada = cap;
}
int STIVA :: regaseste ()
{  int i;
    LISTA * p;
    if ( ! cap )  {
          cout << "Lista este vida ! \n";
          return 0;
    }
    // Se elimina articolul din capul listei
    i = cap->valoare;
    p = cap;
    cap = cap->urm_art;
    delete p;
    return i;
}

// Programul principal
void  main (void)
{  LISTA  * p;                         // Pointerul p punctează la clasa de bază, LISTA
    COADA C;                           // Se implementează o listă de tip "coadă"
    p = & C;                           // Pointerul p punctează la obiectul de tip COADA
    p->stocheaza(1);                   // Lista de tip "coadă" conține elementele 1, 2, 3
    p->stocheaza(2);
    p->stocheaza(3);
    cout << "Coada: ";
    cout << p->regaseste() << " ";     // Se afișează cele trei elemente al listei de tip "coadă"
    cout << p->regaseste() << " ";
    cout << p->regaseste() << " ";
    cout << '\n';

    STIVA ST;                          // Se implementează o listă de tip "stivă"
    p = & ST;                          // Pointerul p punctează la obiectul de tip STIVA
    p->stocheaza(1);                   // Lista de tip "stivă" conține elementele 1, 2, 3
    p->stocheaza(2);
    p->stocheaza(3);
    cout << "Stiva: ";
    cout << p->regaseste() << " ";     // Se afișează cele trei elemente al listei de tip "stivă"
    cout << p->regaseste() << " ";
    cout << p->regaseste() << " ";
```

```
   cout << '\n';
}
```

Pe ecran se va afișa:

```
        Coada: 1 2 3
        Stiva:  3 2 1
```

Funcția main() din acest program ilustrează modul de operare al claselor de tip LISTA. Pentru a sesiza "puterea" polimorfismului, la momentul execuției, vom rescrie funcția main() ca mai jos:

```
void  main (void)
{  LISTA  * p;                    // Pointerul p punctează la clasa de bază, LISTA
   COADA OB_C;                    // Se creează obiectul OB_C "coadă"
   STIVA OB_S;                    // Se creează obiectul OB_S "stivă"
   char ch;
   int n;
   for ( n=0; n<10; n++ )
   {
        cout << "Stiva sau Coada ? (S / C) : ";
        cin >> ch;                                  // Se introduce caracterul de selecție
        ch = tolower (ch);
        if ((ch != 'c') && (ch != 's'))
        {       cout << "Selectati S sau C \n";
                n--;
                continue;
        }
        else if (ch == 'c')  p = & OB_C;
        else if (ch == 's')  p = & OB_S;
        p -> stocheaza(n);
    }
   cout << "Introduceti T pentru a termina \n";
   for ( ; ; ) {
        cout << "Sterge din Stiva sau Coada ? (S /C) : ";
        cin >> ch;                                  // Se introduce caracterul de selecție
        ch = tolower (ch);
        if (ch == 't') break;
        else if (ch == 'c')  p = & OB_C;
        else  p = & OB_S;
        cout << p -> regaseste() << '\n';
        }
    cout << '\n';
}
```

Această nouă funcție main() pune în evidență gestionarea flexibilă a evenimentelor care se produc aleator; aceasta se obține prin utilizarea funcțiilor virtuale și a polimorfismului, la momentul execuției programului. Programul execută o buclă **for**, având un contor de la 0 la 9. La fiecare iterație, suntem solicitați să selectăm tipul listei în care vom pune valorile - "stivă" sau "coadă". Pointerul p este inițializat să puncteze la obiectul adecvat, iar valoarea corespunzătoare este stocată în memorie, în funcție de răspunsul pe care îl dăm. La terminarea buclei, începe o alta, care ne cere să indicăm tipul listei din care vrem să ștergem o valoare. Și în acest caz, totul depinde de răspunsul nostru.