3. IMPLEMENTAREA CLASELOR ÎN LIMBAJUL C++

Cu acest capitol se începe studiul sistematic al claselor, un tip de date definit de programator, care realizează unificarea datelor și a procedurilor. Se vor aborda următoarele topici: declararea claselor și a obiectelor, accesul la date și la funcții membre ale unei clase, conceptul de ascundere a datelor, membrii unei clase care au caracterul **static**, structuri în C++, operatorul **this**, funcții **friend** - elemente preliminare.

3.1. Declararea claselor

Pentru a urmări în mod logic cum a apărut ideea declarării claselor, se va considera următoarea problemă: să se scrie un program care realizează translația de-a lungul axelor de coordonate a unui punct din spațiul tridimensional. O primă rezolvare constă în declararea a trei variabile independente x, y și z și a unei funcții denumită, de exemplu, translatează() ce realizează o translație a punctului respectiv cu tx, ty, tz, ca în Exemplul 3.1.

Exemplul 3.1.

```
int x, y, z;
void translatează( int tx, int ty, int tz )
{       x += tx;                    // x = x + tx
        y += ty;                    // y = y + ty
        z += tz;                    // z = z + tz
}
```

Deși o astfel de exprimare este corectă, ea nu reflectă nicidecum existența unei legături între cele trei variabile x, y și z. Nimic nu dă de înțeles ca ele au o trăsătură comună, adică descriu poziția unui punct în spațiu. Mult mai sugestivă ar fi o exprimare de genul:

Exemplul 3.2.

```
struct coordonate_3D  { int x, y, z; };
void translatează( struct coordonate_3D  punct, int tx, int ty, int tz )
{       punct .x += tx;
        punct .y += ty;
        punct .z += tz;
}
```

Cu toate că este mult mai sugestiv, și acest mod de abordare are un inconvenient. Este vorba de faptul că legătura existentă între structura definită și funcția de translație nu este pusă în evidență în nici un mod. Se simte deci nevoia unui formalism care să specifice clar ideea că funcția translatează() nu poate fi apelată decât pentru a opera asupra unor puncte; acesta se poate implementa prin folosirea claselor ca în Exemplul 3.3.

Exemplul 3.3.

```
class coordonate_3D          // Declararea clasei coordonate_3D
{ private:
        int x, y, z;         // Declarare variabile membru
public:
        void translatează( int tx, int ty, int tz );      // Declarare funcție membru
};
// Definirea funcției membru
void coordonate_3D :: translatează( int tx, int ty, int tz )
{       x += tx;
        y += ty;
        z += tz;
}
```

```
// Declararea a trei obiecte de tipul "coordonate_3D"
class coordonate_3D obiect_1, obiect_2, obiect_3;
```

Pentru claritate, prezentăm următoarele definiții:

- O *clasă* este un tip abstract de date definit de către utilizator, care încapsulează atât datele propriu-zise, cât și operațiile efectuate asupra acestora, dar care se comportă la fel ca un tip predefinit.
- Un *obiect* este o instanțiere a unei clase, deci o variabilă declarată ca fiind de tipul clasă respectiv.
- O *variabilă membru* este o variabilă declarată în cadrul unei clase.
- O *metodă* este o funcție declarată (sau definită) în cadrul unei clase și care poate accesa toate variabilele declarate în cadrul clasei. Metodele se mai numesc și *funcții membru*.

Variabilele și funcțiile declarate în interiorul clasei se numesc *membri* ai clasei. În exemplul de mai sus avem:

- trei obiecte de tipul "coordonate_3D": obiect_1, obiect_2, obiect_3.
- trei variabile membru: x, y, z.
- o singură funcție membru: translatează().

Membrii clasei "coordonate_3D" sunt de două tipuri:

- privați - aparțin secțiunii private și nu pot fi accesați decât de către funcțiile publice ale clasei;
- publici - aparțin secțiunii publice și pot fi accesați din orice punct al programului, dar trebuie prefixați de către un nume de obiect (o instanțiere a clasei) și de operatorul de selectare "**.**", la fel ca și câmpurile unei structuri.

O sintaxă foarte simplificată a descrierii unei clase arată astfel:

```
class Nume_clasă {
        private:    Listă_membri_1
        public:     Listă_membri_2
};
```

Cuvântul cheie **class** indică faptul că urmează descrierea unei clase având numele "Nume_clasă" (în Exemplul 3.1, "coordonate_3D"). Descrierea propriu-zisă a unei clase constă în cele două liste de membri, prefixate eventual de cuvintele cheie **private** și **public**, care împart clasa în cele două secțiuni: una privată și una publică.

O sintaxă mai rafinată a claselor, și deci mai apropiată de cea reală, ar fi următoarea:

```
Specificator_clasă Nume_clasă {
        private:    Listă_membri_1
        public:     Listă_membri_2
};
```

Diferența față de definiția precedentă constă în înlocuirea lui **class** cu "Specificator_clasă". Acesta din urmă poate fi unul din următoarele cuvinte cheie: **class**, **struct** sau **union**. Ultimele două descriu structuri de date având aceleași proprietăți ca și în limbajul C neobiectual, cu două modificări:

- li se pot atașa funcții membru;
- toate elementele (variabile și funcții) sunt de tip **public**.

Dacă în descrierea unei clase lipsesc specificatorii **public** și **private**, toți membrii **clasei** vor fi *implicit* privați.

În cazul specificatorului **union**, implicit, membrii uniunii vor fi toți de tip **public**.

Pentru fixarea celor prezentate mai sus, în Exemplul 3.4 se prezintă listingul unui program în care este declarată o clasă NUMARATOR folosită pentru contorizarea numărului de rotații ale unui dispozitiv magnetic sau mecanic.

Exemplul 3.4.

```
// Program  P3_1.CPP  Definirea si utilizarea clasei NUMARATOR
# define N_TURE     234
# include <iostream.h>
class NUMARATOR {                                   // Declararea clasei
        int cifra_sute, cifra_zeci, cifra_unitati;  // Variabile de tip private
public:
        void Setare (int cs, int cz, int cu);       // Aici, funcția Setare() este numai declarată
        void Avans  (void) {                        // Definitia functiei Avans()
                if (cifra_unitati++ == 9) {
                        cifra_unitati = 0;
                        if (cifra_zeci++ == 9) {
                                cifra_zeci = 0;
                                if (cifra_sute++ == 9) {
                                        cifra_sute = 0;
                                        cifra_unitati = 0;
                                }
                        }
                }
        }
        void Afisare (void);                        // Funcția Afișare() este numai declarată
};
// Definirea funcției Setare()
void NUMARATOR :: Setare (int cs, int cz, int cu)
{
        cifra_sute    = cs;
        cifra_zeci    = cz;
        cifra_unitati = cu;
}
// Definirea funcției Afișare()
void NUMARATOR :: Afisare (void)
{
cout << "Indicatorul = " << "<" << cifra_sute << "|" << cifra_zeci << "|" <<
                           cifra_unitati << ">" << endl;
}
// Programul apelant
void main (void)
{
        NUMARATOR nr;           // Declararea obiectului nr de tip NUMARATOR
        nr.Setare (0, 0, 0);    // Initializarea numaratorului nr
        int n = N_TURE;
        while (n--) {
        nr.Avans ();            // Avans una tura
        nr.Afisare ();          // Afisarea indicatorului
        }
}
```

Clasa NUMARATOR conține trei variabile de tipul **int** plasate în secțiunea **private** a acesteia și trei funcții membre în secțiunea **public**. Prima funcție, Setare() este numai declarată, definirea ei

făcându-se în afara clasei. A doua funcție, Avans() este definită în cadrul clasei. A treia funcție, Afisare(), definită de asemenea în afara clasei, servește la afișarea contorului. Două aspecte se constată cu precădere în cadrul acestei clase, și anume:

1. Lipsa constructorului și destructorului clasei (funcții ce au fost definite în exemplele din Cap. 2);
2. Faptul că unele funcții membru pot fi numai declarate, iar altele pot fi definite. Ce regulă este aici ?

În legătură cu aceste constatări se pot formula următoarele observații:

1. Cu acest exemplu se vede că se pot întâlni clase chiar fără definirea explicită a celor două funcții speciale: constructorul și destructorul clasei.
2. Regula este următoarea: dacă definiția funcției este scurtă (1-3 linii) acea funcție poate fi definită în cadrul clasei.

La momentul executării instrucțiunii NUMARATOR nr; chiar în absența constructorului, se alocă memorie pentru obiectul nr. Acest obiect este recunoscut în întreaga funcție main(). Variabilele de tip **private** cifra_sute, cifra_zeci, cifra_unitati nu pot fi modificate direct din orice linie a funcției main(), ci numai prin intermediul funcțiilor membru Setare() și Avans(). Funcția Afișare() doar inspectează variabilele respective. Referirea la o metodă (funcție membră a clasei) se realizează, așa cum se vede, prin intermediul construcției nume_obiect**.**nume_metodă. Operatorul "**.**" are același sens cu cel din limbajul C, întrebuințat la referirea unui "câmp" dintr-o înregistrare (**struct**, **union**). În C folosim cuplul nume_structură**.**nume_câmp. Nu trebuie confundat obiectul (variabila) nr cu șablonul său, clasa NUMARATOR. În C++, obiectul poate fi considerat precum o variabilă în limbajul C. Numai că (atenție!), se spune că prin instrucțiunea: NUMARATOR nr; declarăm obiectul nr, iar când se execută instrucțiunea: nr.Setare(0, 0, 0), definim valorile variabilelor sale de stare.

3.2. Accesul la date și funcții membre ale unei clasei

În general, șablonul unei clase este următorul:

```
class Nume_clasă {
        private:
        // Secțiunea privată; de regulă cuvântul cheie private lipsește
        protected:
        // Secțiunea protejată (opțional)
        public:
        // Secțiunea publică, interfața cu lumea exterioară clasei
};
```

Fiecare din aceste trei domenii (secțiuni) este într-un fel protejat împotriva accesului neavizat din exteriorul clasei. Membrii clasei (date, funcții) care aparțin domeniului **private** sunt într-un fel similari variabilelor *locale* dintr-o funcție în sensul limbajului C. *Scopul* (domeniul de recunoaștere al) unei astfel de variabile se limitează la cuplul de acolade din declarația clasei. În C++, la acești membri nu au acces decât ceilalți membri de tip **public** ai clasei. Nivelul **protected** este intermediar între **public** și **private**. Modul de acces la membrii protejați va fi prezentat ulterior, în cadrul conceptului de moștenire.

Pentru a explica conceptul de *ascundere* (*hiding*) a datelor să presupunem că în Exemplul 3.4, clasa NUMARATOR se declară sub forma:

```
class NUMARATOR {
        public:
                int cifra_sute, cifra_zeci, cifra_unitati;
                // Urmează cele trei funcții declarate mai sus
};
```

În această situație, vom avea acces direct din exteriorul clasei la cele trei variabile de stare, nu numai prin intermediul funcțiilor Setare(), Avans() și Afisare(). Este ca și când am fi declarat în C++ clasa sub forma structurii:

```
struct NUMARATOR {
        int cifra_sute, cifra_zeci, cifra_unitati;
        void Setare (int cs, int cz, int cu);
        void Avans  (void);
        void Afisare (void);
};
```

Într-o construcție de tipul **struct** totul este **public**, spre deosebire de aceea de tipul **class**. Revenind la listingul programului din Exemplul 3.4 se observă notația "::". Operatorul "::" cuplează de fapt nume_clasă :: nume_funcție_membru și definește domeniul (scopul) în care acea funcție va fi recunoscută. Altfel spus, operatorul "::" denumit *scope resolution operator* sau *scope access operator* definește relația de apartenență la o clasă.

În clasa NUMARATOR, variabilele cifra_sute, cifra_zeci, cifra_unitati pot fi considerate variabile locale în raport cu acest "bloc" delimitat de cele două acolade. Deși definițiile funcțiilor membre sunt în afara acoladelor, totuși operatorul "::" conectează aceste funcții la clasa NUMARATOR. În această situație, *scopul* celor trei variabile se extinde și în cadrul acestor funcții. Putem deci considera aceste variabile drept "globale" în raport cu funcțiile membre ale clasei NUMARATOR.

Modul de utilizare a operatorului "::" poate fi observat și în Exemplul 3.3 la definirea funcției translatează():

```
void coordonate_3D :: translatează( int tx, int ty, int tz )
{
        //  ...
}
```

3.3. Referirea la funcții proprii care au denumiri identice celor din bibliotecile standard

Exemplul următor arată modul de acces la două variabile, una globală și cealaltă locală, de tip **private**. De asemenea, se arată modul de folosire a unei funcții proprii puts(), o funcție cu aceeași denumire ca cea din fișierul antet **stdio.h**. Scopul acestui exemplu este de a lămuri notația legată de folosirea operatorului "::".

Exemplul 3.5.

```
// Program P3_2.CPP  Definirea unei functii proprii care are acelasi nume
//                   cu una din functiile din biblioteca
# include <stdio.h>
# include <math.h>
double v;                       // Variabila v globală
class ADHOC {
        double  v, w;           // Variabila v locală
public:
        void Init (double x, double y) {v = x; w = y;}         // v este variabila locală
        double Modul (void) {return sqrt (v*v + w*w);}          // v este variabila locală
        void puts (char *);                                     // Declarația propriei funcții puts()
};
void ADHOC :: puts (char *mesaj)                                // Definirea funcției proprii puts()
{
        :: puts("Sir scris prin intermediul propriei functii puts:");
        :: puts(mesaj);
```

```
        :: v += 1;                          // v este variabila globală
}
void main (void)
{
        ADHOC a;                            // Se creează obiectul "a"
        v = 3.;                             // v.global = 3.0
        a.Init (v, 4.);                     // v.local = v.global = 3.0, w = 4.0
        v = 15.                             // v.global = 15.0
        printf ( "Modulul = %f\n", a.Modul () );
        a.puts ("Apelez propriul puts");
}
```

Pentru a apela variabila v (globală) din interiorul unei funcții membre a clasei se folosește notația ::v. Simpla întrebuințare a lui v înseamnă referirea la variabila v (locală) de tip **private**. Instrucțiunea a.Init (v, 4.); din funcția main() inițializează variabila locală v a obiectului "a" cu valoarea variabilei globale v = 3 și variabila w cu valoarea 4.

În program se definește o funcție membru denumită puts(), funcție care are aceeași denumire cu a unei funcții ce are prototipul în fișierul antet **stdio.h** și care este utilizată pentru afișarea unui șir. Din definiția acestei funcții, se vede că aceasta apelează funcția puts() standard. Acest apel se realizează prin plasarea operatorului "::" în fața numelui funcției definite în exteriorul clasei, în cazul nostru puts(). În urma executării instrucțiunii :: v += 1; din funcția puts(), v (global) devine egal cu 16, în loc de 15.

3.4. Crearea mai multor obiecte

Prin crearea mai multor obiecte ale unei clase, fiecare obiect recepționează copia sa proprie în memorie, în care apar membrii celor trei categorii de domenii definite mai sus (dacă există toate). În Exemplul 3.4 am fi putut scrie: NUMARATOR nr[10]; și am fi creat zece obiecte identice.

Să considerăm următorul program, în care apare o variabilă globală n_obiecte care va servi la numărarea obiectelor "în viață" la un moment dat, Exemplul 3.6.

Exemplul 3.6.

```
// Program P3_3.CPP   Crearea obiectelor în cadrul blocurilor
# include <iostream.h>
int n_obiecte = 0;              // Variabila globală n_obiecte
class OB {                      // Obiectul nu conține date private
public:
        OB () {                 // Constructorul clasei crește cu 1 variabila n_obiecte
        n_obiecte ++;           // Putem să ne referim la variabile globale din interiorul clasei
        cout << "Numarul obiectelor in viata " << n_obiecte << endl;
        }
        ~OB () {                // Destructorul clasei descrește cu 1 variabila n_obiecte
        n_obiecte --;
        cout << "Au mai ramas doar " << n_obiecte << " obiecte"<< endl;
        }
};
void main (void)
{
        OB   a, b, c;           // Se creează primele 3 obiecte, notate a, b, c
        {                       // Se deschide un nou context
                OB   d, e;      // Se creează încă 2 obiecte, notate d, e
        }                       // Aici destructorul va distruge obiectele d și e
```

```
    {                       // Se redeschide un alt context
        OB f;               // Se creează obiectul f
    }                       // Aici destructorul va distruge obiectul f
}                           // Aici se eliberează memoria ocupata de obiectele a, b, c
```

Prin execuția acestui program se va obține numărul obiectelor alocate în memorie la un moment dat. Cum comentariile din acest program sunt lămuritoare, chiar exhaustive, considerăm că alte explicații nu mai sunt necesare. Precizăm totuși că variabila globală n_obiecte poate fi modificată nu numai de către funcțiile clasei OB, ci chiar și direct din funcția main().

Din Exemplul 3.6 se observă că la fiecare declarație a unui obiect (de exemplu, OB a, b, c;) este apelat în mod automat constructorul care alocă/rezervă spațiu pentru membrii clasei. Obiectele au o existență efemeră, deoarece la întâlnirea acoladei "}", destructorul va elibera spațiul ocupat de obiectele create. Obiectele de mai sus nu au avut date proprii. Ele au folosit o variabilă (n_obiecte) pe care constructorii și destructorii au modificat-o pe măsura creării și distrugerii acestor obiecte. Pentru ca starea (valoarea) unor variabile proprii obiectelor să fie păstrată (memorată) se poate folosi cuvântul cheie **static** care să prefixeze aceste variabile. Un exemplu în acest sens va fi dat într-un capitol următor.

3.5. Prevenirea redeclarării claselor

De regulă, declarațiile și definițiile claselor nu sunt plasate în același fișier cu funcția main(), ci într-un fișier antet inclus cu o comandă de genul: # **include** <nume_fișier_antet>. Pentru evitarea redeclarării unei clase se recomandă următoarea următorului grup de comenzi (instrucțiuni de preprocesare):

```
# ifndef nume
# define nume
// Declarația și definiția clasei
# endif
```

De exemplu, programul P3_1.CPP din Exemplul 3.4 poate fi alcătuit dintr-un fișier antet NUMARATOR.H și programul propriu-zis P3_1.CPP. Structura fișierului antet NUMARATOR.H ar putea fi următoarea:

Exemplul 3.7.

```
// Declararea și definirea clasei NUMARATOR
# ifndef NUMARATOR.H
# define NUMARATOR.H
# include <iostream.h>
class NUMARATOR
{
    //  Declarația și definiția din Exemplul 3.4
};
// Implementarea funcțiilor membre
# endif NUMARATOR.H
```

În acest caz, fișierul program P3_1.CPP devine:

```
// Programul P3_4.CPP
# define N_TURE     234
# include "NUMARATOR.H"
void main (void)
{
    NUMARATOR nr;   // Declararea obiectului nr de tip NUMARATOR
    nr.Setare (0, 0, 0);    // Initializarea numaratorului nr
```

```
        int n = N_TURE;
        while (n--) {
        nr.Avans ();            // Avans una tura
        nr.Afisare ();          // Afisarea indicatorului
        }
}
```

3.6. Structuri și uniuni în limbajul C++

În C++ structurile constituie un tip special de clasă. Structura acceptă ceea ce acceptă o clasă cu excepția membrilor de tip **private**. Deci, într-o structură se pot întâlni și funcții. De asemenea proprietățile unei structuri de bază pot fi moștenite de o structură derivată. Echivalența între o structură C și o clasă C++ este redată în Fig. 3.1.

```
struct AUTOMOBIL                    class AUTOMOBIL
{                                   {
        char *marca                 public:
        char *tip                           char *marca
        int viteza                          char *tip
        int nr_locuri                       int viteza
        int consum                          int nr_locuri
};                                          int consum
                                    };
```

Fig. 3.1. O paralelă între structură (C) și clasă (C++)

În C++, crearea unui obiect denumit my_car cu șablonul (*tag*-ul) AUTOMOBIL se poate face în două feluri: a) **struct** AUTOMOBIL my_car; b) AUTOMOBIL my_car;

Uniunea (în sens C) este de fapt o suprapunere a mai multor elemente (câmpuri) care încep toate de la aceeași adresă de memorie. În plus față de limbajul C, în corpul uniunii C++ pot să apară și funcții de tip membru, constructori etc. Ca și la structuri, toți membrii uniunii (date, funcții) vor fi publici. În Exemplul 3.8 declarăm o uniune UNIO, care conține doi constructori, un destructor și două funcții denumite ObtinvalInt(), care întoarce un rezultat întreg și respectiv ObtinvalVM(), care întoarce un rezultat în virgulă mobilă, dublă precizie. Ca date membre, UNIO conține o variabilă i întreagă și o variabilă d de tip **double**.

Exemplul 3.8.

```
// Program P3_5.CPP  Uniune cu constructori si membri
# include <iostream.h>
union UNIO {
        int  i;  double  d;
        UNIO (int ii) {i = ii;}                 // Primul constructor
        UNIO (double dd) {d = dd;}              // Al doilea constructor
         ~UNIO () {d = 0;}          // Destructorul plasează zero în cea mai mare zonă de memorie
// Funcții membru
        int ObtinvalInt () {return  i;}
        double ObtinvalVM () {return  d;}
} ;
void main (void)
{        int  k;
```

```
        double vm;
        UNIO intreg (1024);             // Se creează obiectul "intreg"
        UNIO real (3.14);               // Se creează obiectul "real"
        k = intreg.ObtinvalInt ();
        cout << "k = " << k << endl;
        intreg.i = intreg.i << 2;       // Se mărește de 4 ori obiectul "intreg"
        k = intreg.ObtinvalInt ();
        cout << "k = " << k << endl;
        vm = real.ObtinvalVM ();
        cout << "Valoarea in virgula mobila = " << vm << endl;
}
```

Obiectele "intreg" și "real" vor asambla valorile 1024 și 3.14 așa cum se vede în Fig. 3.2.

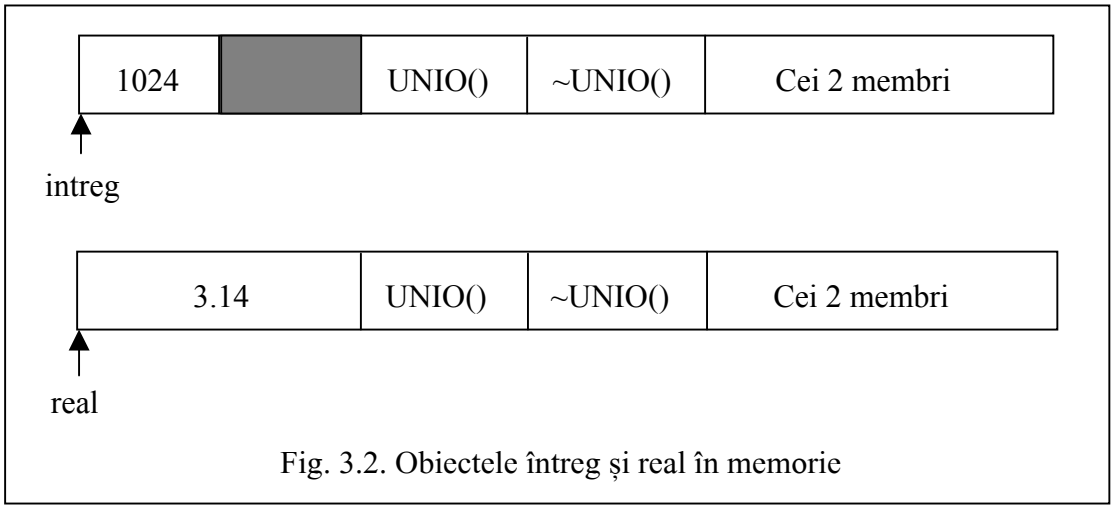

Fig. 3.2. Obiectele întreg și real în memorie

Se vede că pentru obiectul "intreg" cu valoarea 1024 se alocă tot 4 octeți ca și pentru obiectul "real" cu valoarea 3.14. Programul preia în variabilele k și vm valorile celor două obiecte utilizând funcții tip membru adecvate. Desigur că, dacă am fi citit cu funcția ObtinlvalVM valoarea celor patru octeți ai obiectului "intreg", am fi obținut o valoare ciudată.

3.7. Operatorul "this"

Din exemplele anterioare, se vede că pentru a defini funcțiile membre sunt necesare referiri la datele membre ale clasei, fără a specifica un obiect anume. Deși, după declarare, pentru fiecare obiect al unei clase există câte o copie a variabilelor membre, toate obiectele își împart un singur set de funcții membre. La apelare, o funcție membru este informată asupra identității obiectului asupra căruia va acționa prin transferul unui parametru implicit, care reprezintă adresa obiectului. Deci, la apelul unei funcții membru, compilatorul C++ modifică fiecare funcție membru dintr-o clasă făcând două schimbări:

1. Transmite un argument suplimentar cu numele **this**, reprezentând un pointer la obiectul specific care a apelat funcția.
2. Adaugă prefixul **this ->** tuturor variabilelor și funcțiilor membre.

De exemplu, funcția Setare din clasa NUMARATOR putea fi definită, folosind operatorul **this**, astfel:

```
// Definirea funcției Setare() folosind operatorul this
        void NUMARATOR :: Setare (int cs, int cz, int cu)
        {
                this -> cifra_sute     = cs;
                this -> cifra_zeci     = cz;
```

```
        this -> cifra_unitati = cu;
    }
```

Operatorul **this** va puncta pe începutul "șablonului" clasei, în cazul nostru, NUMARATOR, Fig. 3.3. Putem astfel să ne referim la oricare element (membru de tip dată sau funcție) al obiectului curent, nu însă și în interiorul unei funcții. **this** întoarce deci adresa de început a obiectului curent. Notația ***this** va însemna referirea la întregul obiect. De exemplu, dacă se dorește a se returna obiectul curent programului apelant, se poate folosi instrucțiunea:
return *this;

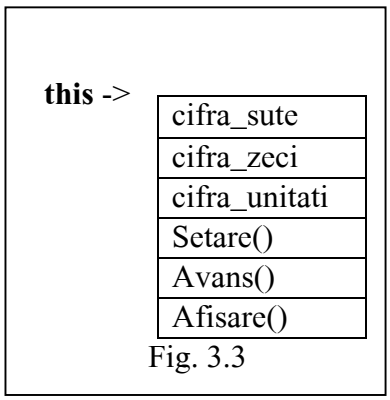

Fig. 3.3

3.8. Funcții de tipul friend - elemente preliminare

O funcție de tip **friend** este în esență o funcție standard, care nu este membră a clasei, ci are numai acces la membrii de tip **private** ai acelei clase. Considerăm o clasă denumită OWN care admite ca prieten funcția Visitor(), astfel:

Exemplul 3.9.

```
class OWN {
        double d_lobby;                 // d_lobby de tip private
public:
        double d_common;                // d_common de tip public
        void Lobby (double d);
        friend double Visitor (OWN *p);
} ;
void OWN :: Lobby (double d)
{
        d_lobby = d;
}
double Visitor (OWN *p)
{
        return p -> d_lobby;            // Se acceptă preluarea valorii d_lobby
}
double Intruder (OWN *p)
{
        return p -> d_common;           // Va apare o eroare
}
```

Funcția Lobby() membră a clasei OWN, inițializează variabila privată d_lobby. Funcția Visitor() este admisă ca prieten al clasei OWN, fapt evidențiat de utilizarea cuvântului cheie **friend**. Acest statut îi va da dreptul să citească și/sau să modifice variabilele clasei OWN. În cazul nostru ea preia valoarea variabilei d_lobby. Funcția Intruder() nu este nici membră a clasei OWN și nici de tip **friend**. De aceea, compilatorul va semnala eroare în încercarea acesteia de a avea acces la variabila publică d_common.