Capitolul 3

Definirea şi utilizarea claselor

După cum s-a precizat în capitolul precedent, noţiunea de ***clasă*** nu este specifică paradigmei programării orientate pe obiecte, însă ea reprezintă un element fundamental atât pentru abstractizarea datelor, cât şi pentru programarea orientată pe obiecte.

Dintr-un anumit punct de vedere, clasa poate fi privită ca o extensie a noţiunii de structură din limbajul C, fiind construcţia prin intermediul căreia limbajul C++ permite definirea unor noi tipuri de date. Însă diferentele între `struct` şi `union` pe de o parte şi `class` pe de altă parte sunt esenţiale, ele referindu-se în special la drepturile de acces ale membrilor şi la posibilitatea derivării claselor.

Din punct de vedere conceptual, o ***clasă*** reprezintă proprietaţile comune ale unei colecţii de obiecte înrudite, ceea ce justifică alegerea cuvântului `class` pentru această noţiune. Obiectele ce aparţin unei clase se numesc ***instanţe*** ale clasei sau ***obiecte instanţă***. Astfel, o clasă poate fi privită ca un tip de date reprezentat de mulţimea tuturor instanţelor sale.

Din punctul de vedere al compilatorului, un obiect reprezintă, ca orice variabilă, o zonă de stocare a datelor, a cărei adresă de memorie este unică. În această zonă se stochează valorile curente ale datelor membre corespunzătoare obiectului respectiv, dar permite în plus funcţiilor membre ale clasei din care face parte să opereze asupra acestor valori.

Din punct de vedere al implementarii, un program ce utilizează programarea orientată pe obiecte presupune atât definirea claselor ce se utilizează, precum şi declararea şi utilizarea obiectelor instanţă cu care se lucrează.

Definirea unei clase constă din două părţi distincte : ***declararea*** clasei şi ***implementarea*** clasei respective. În mod uzual, declaraţia unei clase este separată de implementare, ea fiind descrisă într-un fişier header.

Partea de declaraţie a unei clase trebuie să specifice atât ***numele*** clasei şi eventual clasele din care este derivată, cât şi ***componentele*** sau ***membrii*** clasei respective. Spre deosebire de limbajul C, componentele unei clase (şi în mod implicit componentele unei structuri sau uniuni) pot fi atât date (membrii de tip date), cât şi funcţii (membrii de tip funcţie).

Exemplul 3.1. Structura unui program simplu ce utilizează o clasă *stack* pentru definirea o stivă de caractere poate fi următoarea :

```
// fişierul stack.h - declaraţia clasei
class stack {
  int dim ;              // data membru
  char *buff ;           // data membru
public :
  stack(int) ;           // functie constructor
```

```
  ~stack() ;              // functie destructor
  void push(char) ;       // functie membru
  char& pop() ;           // functie membru
} ;

// fişierul stack.cpp – implementarea clasei
stack::stack(int n)
{
  // codul pentru constructor
}

stack::~stack(int n)
{
  // codul pentru destructor
}

void stack::push(char c)
{
  // codul pentru functia push
}

char& stack::pop()
{
  // codul pentru functia pop
}

// fişierul main.cpp – utilizarea clasei
void main() {
  // declararea obiectelor instanta
  stack st1(100), st2(50) ;
  // utilizarea obiectelor stack
  st1.push('a') ;
  // ...
}
```

Utilizarea claselor într-o aplicație orientată pe obiecte presupune crearea unei mulțimi de obiecte instanță şi transmiterea spre acestea a anumitor mesaje, precum şi recepționarea alor mesaje de la obiecte. Mecanismul de transmitere şi recepționare a mesajelor spre/dinspre obiecte îl reprezintă apelul funcțiilor membru.

3.1 Declararea claselor

Sintaxa pentru declararea unei clase este următoarea :

```
<declarare clasa>   ::= <antet clasa> [<declarare membri>] ;
<antet clasa>       ::= <specificator clasa> <nume clasa>
          [: <clasa de baza> {, <clasa de baza>}*]
<declarare membri>  ::= '{' {<membru specific>}* '}'
<specificator clasa>     ::= struct | class
<clasa de baza>     ::= [<modificator acces clasa> :]
```

```
          <nume clasa>
<membru specific>   ::= [<modificator acces membru> :]
          <declarare membru>
```

Se observă faptul că antetul unei clase conține în mod obligatoriu numele clasei şi specificatorul de clasă, care poate fi `struct` sau `class`. Este posibil ca acesta să fie şi `union`, dar în acest caz clasa nu poate fi nici clasă de bază pentru alte clase şi nici o clasă derivată din alte clase.

O clasă poate fi derivată din una sau mai multe clase, care se numesc în acest *caz* ***clase de bază*** pentru clasa curentă. În cazul în care clasa ce se declară este derivată din alte clase, acestea trebuie specificate prin numele lor şi tipul de acces la ele. Tipurile de acces pot fi `private` şi `public`. În mod implicit pentru `class` acesta este `private`, iar pentru `struct` tipul de acces este `public`. Despre tipul de acces la clasele de bază se va discuta ulterior, în cadrul paragrafului asociat moştenirii.

Din sintaxa anterioară se observă faptul că declararea membrilor unei clase este opțională, caz în care este vorba despre o declarație de clasă incompletă. Declararea incompletă a unei clase este folosită în mod uzual în cazul claseor definite recursiv. Evident, înainte de a declara un obiect instanță al unei clase, clasa trebuie declarată complet.

Secvența următoare reprezintă un exemplu de declarare incompletă. Fiecare dintre clasele *ClA* şi *ClB* utilizează un pointer spre un obiecte al celeilalte clase (clasa *ClB* nu se poate utiliza un obiect, deoarece *ClA* nu a fost încă definită).

```
struct ClA ;
struct ClB {
  // ...
  ClA *a ;
  // ...
} ;
struct ClA {
  // ...
  ClB *b ;
  // ...
} ;
```

Observație. Deoarece o structură permite definirea unei clase, aceasta poate fi utilizată direct ca nume pentru un tip de date, fără sa mai fie necesară utilizarea lui `typedef`.

Deoarece membrii unei clase pot fi date sau funcții, declararea lor se face în mod asemănător ca şi declararea variabilelor şi funcțiilor în limbajul C. Excepție fac două categorii speciale de membri funcție, numiți ***constructori*** şi ***destructori***, despre care se va vorbi ulterior.

În mod opțional, declararea unui membru al unei clase poate fi precedată de un ***modificator de acces*** al membrului respectiv (a nu se confunda cu modificatorul de acces al unei clase de bază). Acest modificator de acces poate fi `private`, `protected` sau `public`. Modificatorul de acces la un membru al unei clase specifică modul în care membrul respectiv poate fi vazut în exteriorul clasei. Un membru `public` este vizibil în exterior, un membru `private` este inaccesibil, iar un membru `protected` poate fi accesibil doar într-o clasă

derivată din clasa respectivă cu modificatorul de acces `public`. Cu alte cuvinte, în afară de clasele derivate public din clasa curentă, toți membri `protected` sunt inaccesibili în exterior. Despre tipul `protected` se va discuta ulterior.

Observație. Un modificator de acces afectează accesibilitatea tuturor membrilor declarați după acesta în clasa curentă, până la întâlnirea unui alt modificator de acces. Dacă primul membru declarat al clasei nu are specificat un modificator de acces, atunci în mod implicit acesta este `private` pentru `class` şi `public` pentru `struct`.

Exemplul 3.2. Clasa *poligon* memorează pointeri la vârfurule unui poligon (nu coordonatele acestora).

```
struct punct {
  double x, y ;
  punct(double x0=0, double y0=0)
     { x = x0; y = y0; }
} ;

class poligon {
  // membri private
  int nr_varfuri ;
  punct **varfuri ;
  double arie, perimetru ;
  void CalculPerimetru() ;
  void AjusteazaArie() ;
public :
  // membri public
  poligon() ;
  ~poligon() ;
  int NrVarfuri() const { return nr_varfuri ; }
  void AddVarf (punct*) ;
  punct* operator[](int) ;
  double Arie() const { return arie; }
  double Perimetru() const { return perimetru; }
} ;
```

Membrii publici ai unei clase pot fi accesibili în exterior şi reprezintă interfața prin intermediul căreia clasa comunică cu exteriorul ; membrii privați sunt locali clasei respective.

Din punct de vedere al domeniului de vizibilitate, numele declarate în interiorul unei clase sunt interne acesteia. Aceasta permite definirea unor membrii diferiți în clase diferite, dar cu acelaşi nume.

În mod uzual în cadrul unei declarații de clasă, funcțiile membru sunt doar declarate. Însă, aşa dupa cum se observă şi din exemplul anterior (funcțiile *NrVarfuri*, *Aria* şi *Perimetru*), în cazul funcțiior simple se pot utiliza funcții `inline`, definite chiar declarația clasei. Deoarece funcțiile `inline` sunt expandate la compilare, corpul lor trebuie să conțină instrucțiuni puține şi simple.

Ca şi în cazul altor tipuri de date, obiectele instanță ale unei clase pot fi declarate constante. În acest caz ar putea să apară probleme când se apelează funcțiile membru ale unui obiect constant, care pot modifica alți membrii ai obiectului respectiv.

Limbajul C++ permite ca în cazul unui obiect constant să poată fi apelate *doar* ***funcții membru constante*** ale clasei. O funcție membru constantă se specifică în declarația clase cu ajutorul cuvântului cheie `const` plasat imediat după antetul funcției. O asemenea funcție nu trebuie să modifice valorile datelor membre ale clasei din care face parte, iar acest lucru este verificat de către compilator.

Exemplul 3.3. Definirea unei clase *cerc* :

```
class cerc {
  double xc, yc ;
  double r ;
public :
  cerc(double a, double b, double c)
     { xc = a ; yc = b ; r = c ;}
  double GetXc() const { return xc ; }
  double GetYc() const { return yc ; }
  double GetR() const { return r ; }
  void Translate(double dx, double dy)
     { xc += dx ; yc += dy ; }
} ;
```

Utilizarea clasei cerc :

```
cerc c1(0, 0, 10) ;
const cerc c2(8, 7, 5) ;
c1.Translate(2, 3) ;     // corect
c2.Translate(2, 3) ;     // incorect
double x = c2.GetXc() ;  // corect
```

Funcții de acces

Funcțiile de acces reprezintă o categorie de funcții membru foarte utilizate. Acestea sunt funcții, în mod uzual definite `inline`, care permit citirea sau modificarea valorilor datelor membru private ale claselor, astfel încât utilizatorul să nu aibă acces direct asupra acestora. Funcțiile de citire se numesc în mod uzual *accesori*, iar celelalte *modificatori*.

Nu există reguli predefinite pentru alegerea numelor acestora, dar în mod uzual accesorii sunt prefixați de *Get*, iar modificatorii de *Set*. De exemplu, funcțiile *GetXc*, *GetYc* şi *GetR* sunt funcții accesor. O funcție modificator se poate defini astfel :

```
void SetXc(double x) { xc = x; }
```

O altă variantă des utilizată constă în scrierea unor funcții supraîncărcate, atât pentru accesori, cât şi pentru modificatori. De exemplu, pentru data membru *xc* a clasei *Cerc*, se pot defini următoarele funcții de acces :

```
void Xc(double x) { xc = x; }
```

```
  double Xc() const { return xc; }
```

Observație. Nu este indicat ca funcțiile de tip accesor să returneze referințe sau pointeri neconstanți la datele private ale claselor (în caz contrar s-ar permite utilizatorilor accesul direct la acestea).

3.2 Implementarea claselor. Operatorul de rezoluție

Pentru definirea completă a unei clase trebuie definite toate funcțiile membru din declarația clasei respective care nu sunt `inline`. În mod uzual definirea funcțiilor se face într-un fișier distinct de fișierul header ce conține declarația clasei.

În multe aplicații însă, un fișier header conține declarațiile mai multor clase înrudite ce pot forma una sau mai multe ierarhii de clase. În acest caz este posibil să existe mai multe funcții cu același nume declarate în clase diferite între care nu există relația de moștenire, care nu sunt supraîncărcate și care reprezintă funcții diferite (lucru permis de limbaj datorită domeniului de definiție al numelor declarate în interiorul unei clase).

Pentru a putea specifica în cazul fiecarei funcții membru clasa de care aparține, limbajul C++ pune la dispoziție un nou operator numit ***operator de rezoluție*** (notat **::**). Utilizând operatorul de rezoluție, specificarea completă a numelui unei funcții se face astfel :

```
<nume clasa> :: <nume funcție>
```

De exemplu, definirea funcțiilor *AddVarf, CalculPerimetru* și *AjusteazaArie* din cadrul clasei *poligon* se pot realiza astfel :

```
void poligon::AddVarf(punct* p) {
  punct **v = new punct*[nr_varfuri+1] ;
  for(int i=0 ; i<nr_varfuri ; i++)
    v[i] = varfuri[i] ;
  v[nr_varfuri++] = p ;
  delete[] varfuri ;    //eliberarea memoriei pt. varfuri
  varfuri = v;
  CalculPerimetru();
  AjusteazaArie();
}

void poligon::AjusteazaArie(void) {
  if (nr_varfuri > 2)
    arie = arie+(varfuri[0]->x*varfuri[nr_varfuri-2]->x +
      varfuri[nr_varfuri-2]->y*varfuri[nr_varfuri-1]->x +
      varfuri[nr_varfuri-1]->y*varfuri[0]->x -
      varfuri[0]->x*varfuri[nr_varfuri-2]->y -
      varfuri[nr_varfuri-2]->x*varfuri[nr_varfuri-1]->y -
      varfuri[nr_varfuri-1]->x*varfuri[0]->y ) / 2;
}

void poligon::CalculPerimetru(void) {
  double l;
```

```
  if (nr_varfuri > 1)
    for (int i=0; i<nr_varfuri-1; i++) {
      l = sqrt((varfuri[i]->x - varfuri[i+1]->x) *
          (varfuri[i]->x - varfuri[i+1]->x) +
          (varfuri[i]->y - varfuri[i+1]->y) *
          (varfuri[i]->y - varfuri[i+1]->y));
      perimetru += l;
    }
    l = sqrt((varfuri[0]->x - varfuri[nr_varfuri-1]->x) *
        (varfuri[0]->x - varfuri[nr_varfuri-1]->x) +
        (varfuri[0]->y - varfuri[nr_varfuri-1]->y) *
        (varfuri[0]->y - varfuri[nr_varfuri-1]->y));
    perimetru += l;
}
```

Operatorul de rezoluție poate fi utilizat şi în alte cazuri decât cel prezentat anterior. O utilizare frecventă se referă la specificarea unui nume care este ascuns în cadrul unui bloc datorită regulii domeniului de vizibilitate. De exemplu, o variabilă definită într-un fişier în afara oricarei funcții este vizibilă în toate funcțiile din fişierul respectiv, mai puțin în acele blocuri în care ea este redefinită :

```
int k ;

f1() {
  // k este vizibila in acest bloc
}

f2() {
  int k = 0 ;  // variabila k definita la nivelul
               // fisierului nu este vizibila
               // in mod normal în f2
  k = k+2 ;
  ::k = ::k+2 ;  // variabila k din domeniul exterior
}
```

Operatorul de rezoluție poate fi utilizat în acest caz ca un operator unar ce prefixează un nume, caz în care el referă cea mai din exterior apariție a numelui respectiv, declarat la nivelul de fişier. În exemplul anterior, instrucțiunea :

```
k = k+2 ;
```

referă variabila definită în interiorul funcției *f2*, pe când :

```
::k = ::k+2 ;
```

referă variabila definită la nivelul fişierului.

Deoarece constructorii şi destructorii unei clase sunt funcții membru ale clasei respective, în cazul în care nu sunt definiți ca funcții `inline`, ei trebuie definiti în fişierul de implementare al clasei. De exemplu, un constructor şi un destructor pentru clasa *poligon* se pot defini astfel :

```
poligon::poligon()       // contructor
{
  varfuri = 0 ;
  nr_varfuri = 0 ;
```

```
  arie = perimetru = 0;
}

poligon::~poligon()      // destructor
{
  delete[] varfuri ;
}
```

Asa cum s-a prezentat anterior, constructorii şi destructorii sunt categorii speciale de funcţii care nu au specificat nici un rezultat (nici măcar `void`). În plus, desctructorii nu pot avea nici argumente.

Observaţie . Constructorul clasei *poligon* nu alocă memorie pentru vârfurile poligonului, completatrea efectiva a vârfurilor trebuie făcută în cadrul programului cu ajutorul funcţiei *AddVarf*.

O altă categorie specială de funcţii membru o constituie ***operatorii***. Limbajul C++ permite supraâncărcarea operatorilor uzuali ai limbajului pentru a putea defini alte operaţii. Specificarea unui operator ca o funcţie se realizează cu ajutorul cuvântului cheie `operator` folosit ca prefix al operatorului respectiv. În exemplul clasei *poligon* s-a supraâncărcat operatorul de indexare :

```
punct* poligon::operator[](int k) {
  if (k < 0) {
    cout << "\nIndex negativ" ;
    return 0 ;
  }
  return varfuri[k] ;
}
```

3.3 Utilizarea claselor

După cum s-a precizat anterior, ***utilizarea*** claselor înseamnă crearea anumitor obiecte instanţă ale acestor clase şi comunicarea cu obiectele respective prin intermediul mesajelor, adică a funcţiilor membru ale lor.

Obiectele instanţă se comportă într-un mod asemănător variabilelor, în sensul că li se rezervă o zonă de memorie pentru stocarea valorilor datelor membre. Toate observaţiile referitoare la clasele de memorarea pentru variabile sunt valabile şi în cazul obiectelor, astfel încât alocarea memoriei pentru obiecte se poate face atât în zona de date, cât şi în zona stivă şi în zona heap.

Indiferent de zona de alocare, crearea unui obiect presupune două acţiuni distincte :

- alocarea de către compilator a unei zone de memorie de dimensiune corespunzătoare ;
- apelul unei funcţii constructor a clasei din care obiectul face parte, care realizează, printre altele, iniţializarea anumitor date membre cu valori.

Dimensiunea zonei de memorie alocată pentru un obiect instanţă este în general dată de suma dimensiunilor datelor membre ale obiectului, dar această dimensiune depinde de implementare. Există situaţii în care dimensiunea zonei de memorie a unui obiect este mai

mare decât suma componentelor, în special în cazul utilizării polimorfismului, sau a unor clase care nu conțin decât funcții membru.

De exemplu, pentru un anumit compilator Visual C++, următoarele declarații :

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

struct A {
  int n;
  A(int k) { n = k ;}
  int N() { return n; }
};

struct B {
  void Print() { cout<<"B"; }
};

void main() {
  A a;
  B b;
  cout << sizeof(a) << endl;
  cout << sizeof(b) << endl;
}
```

au ca efect afişarea valorilor 4 şi 1, deoareca valorile `int` se reprezintă pe 4 octeți. Valoarea 1 apare deoarece compilatorul nu permite existența unor obiecte cu dimensiunea zero.

Deşi zona de cod pentru funcțiile membre ale unei clase nu se copiază în zonele de memorie alocate obiectelor instanță ale clasei, apelul unei funcții membru a unui obiect este legat strict de obiectul respectiv, prin transmiterea unui parametru ascuns ce referă adresa de memorie a obiectului respectiv.

Vom discuta acest aspect pe baza unui exemplu. În cazul în care se doreşte ca o aplicație ce lucrează cu poligoane să poată utiliza mai multe poligoane memorate într-o listă dublu înlănțuită, la clasa *poligon* mai trebuie adăugați doi membri de tip pointer ce indică respectiv spre poligonul precedent şi cel următor poligonului curent. Clasa *poligon* se poate rescrie astfel (aici nu este vorba despre o derivare, ci pur şi simplu de scrierea unei alte clase distinctă de cea precedentă) :

```
class p_poligon {
  int nr_varfuri ;
  punct **varfuri ;
  double arie, perimetru ;
  p_poligon *succ, *pred ;
  void CalculPerimetru() ;
  void AjusteazaArie() ;
public :
  p_poligon()
  {
    varfuri = 0 ;
```

```
    nr_varfuri = 0 ;
    arie = perimetru = 0 ;
    succ = pred = 0;
  }
  ~p_poligon() ;
  int NrVarfuri() const { return nr_varfuri ; }
  void AddVarf (punct*) ;
  punct* operator[](int) ;
  double Arie() const { return arie; }
  double Perimetru() const { return perimetru; }
  p_poligon* Pred() const { return pred ; }
  p_poligon* Succ() const { return succ ; }
  void AddPoligon(p_poligon*) ;
} ;
```

Cuvântul cheie this

Implementările funcțiilor sunt la fel, cu excepția celor noi adaugate. Funcția *AddPoligon* adaugă un nou poligon în listă, ca succesor al poligonului curent :

```
void p_poligon::AddPoligon(p_poligon* p) {
  p->succ = 0 ;
  p->pred = this ;
  succ = p ;
}
```

Se observă utilizarea unui nou cuvânt cheie, `this`. Acesta este numit ***pointer la self*** şi indică întotdeauna spre obiectul curent. `this` poate fi privit ca un parametru ascuns (invizibil) în cadrul funcției *AddPoligon* declarat astfel :

```
p_poligon* this
```

Observație. În cazul unei clase oarecare X, parametrul :

```
X* this
```

este transmis în toate funcțiile membru nestatice ale clasei X.

La crearea unui obiect instanță ale unei clase după operația de alocare a memoriei pentru obiect se apelează în mod implicit un constructor pentru obiectul respectiv. Acest constructor va avea şi el `this` ca parametru ascuns inițializat cu adresa blocului de memorie asociat obiectului.

Să presupunem că în aplicația curentă s-au definit două obiecte :

```
p_poligon* p = new p_poligon ;
// ...
p_poligon d ;
// ...
```

În ambele cazuri se apelează constructorul clasei *p_poligon* : în primul caz el este apelat implicit de către operatorul `new`, iar memoria se alocă în zona `heap` a programului, iar în

cazul al doilea constructorul este de asemenea apelat implicit de către compilator, dar memoria pentru obiect se alocă în zona de date.

În ambele cazuri, obiectele create vor conține *o copie a tuturor datelor membre ale clasei p_poligon*, iar unii dintre membri (*nr_varfuri*, de exemplu) vor fi inițializati de către constructori cu anumite valori. În plus, toti parametrii `this` ai funcțiilor membru pentru fiecare obiect vor fi inițializati cu adresa blocului de memorie asociat obiectului (in primul caz, de exemplu, cu valoarea pointerului *p*).

Să presupunem că se mai crează un obiect :

```
p_poligon* p4 = new p_poligon ;
// ...
```

Adăugarea lui *p4* în lista de poligoane dupa poligonul *p* se poate descrie astfel :

```
p->AddPoligon(p4) ;
```

Acum se poate observa utilitatea pointerului `this`, pentru că fiecare obiect nou creat din clasa *p_poligon* va avea o altă adresă de memorie, care este necesară la eventuala adaugare în lista de poligoane.

Utilizarea parametrului ascuns `this` nu este neapărat necesară, decât în cazul în care trebuie facută o referire explicită la adresa de memorie a obiectului curent. De exemplu, funcția *AddPoligon* se mai poate scrie astfel :

```
void p_poligon::AddPoligon(p_poligon* p) {
  p->succ = 0 ;
  p->pred = this ;
  this->succ = p ;
}
```

Observație. În cazul utilizarii unui pointer la un obiect instanță, constructorul clasei nu este apelat dacă nu se apelează operatorul `new`. De exemplu, declarația :

```
p_poligon* pp;
```

nu creează efectiv un obiect din clasa *p_poligon*.

Membrii statici ai unei clase

Dupa cum s-a mai precizat, în mod normal fiecare obiect instanță a unei clase posedă o copie a datelor membre ale clasei de care aparține. Din acest motiv orice modificare a valorii unui membru a obiectului este locală instanței respective şi nu se vede în cadrul altor instanțe ale aceleiaşi clase.

Limbajul C++ oferă posibilitatea în plus a definirii unor membri ce pot fi utilizați în comun de toate instanțele unei clase. Aceşti membri se numesc ***membri statici*** şi sunt declarați cu ajutorul cuvântului cheie `static`.

Exemplul 3.4. Se consideră o clasă *Experiment* ce permite descrierea observațiilor efectuate asupra unei mărimi fizice. Fiecare obiect al clasei memorează o valoare măsurată a mărimii fizice. Clasa *Experiment* trebuie să determine numărul de observații la un moment dat,

precum şi media aritmetică a valorilor şirului de determinări. S-au utilizat două date membru statice (*n* şi *s*) care memorează numărul de obiecte instanţă create până la momentul curent şi suma valorilor lor, precum şi două funcţii membru statice (*N* şi *Med*) pentru determinarea numărului de obiecte curente ale clasei şi a mediei aritmetice a valorilor acestora.

```
//fisierul experiment.h
class Experiment {
  double x;
  static int n;
  static double s;
public:
  Experiment(double);
  double X() const { return x; }
  static double Med() { return s/n; }
  static int N() { return n; }
};
```

Datele membru de tip static sunt comune pentru toate obiectele unei clase şi au alocată o zonă de memorie diferită de datele nestatice. În acest mod se poate realiza comunicarea simplă şi eficientă între diferitele obiecte ale unei clase.

Datorită modului de stocare, definirea efectivă (alocarea memoriei şi iniţializarea cu valori) a datelor membre statice trebuie făcută în afara declaraţiei clasei şi într-un singur loc din program. În mod uzual definirea lor se face în fişierul ce conţine implementarea clasei respective, evitandu-se în acest mod definiţiile multiple.

Pentru exemplul anterior, în fişierul de implementare al clasei *experiment* trebuie adaugate :

```
int Experiment::n = 0 ;
double Experiment::s = 0 ;
```

ca în exemplul următor :

```
//fisierul experiment.cpp
int Experiment::n = 0;
double Experiment::s = 0;

Experiment::Experiment(double v) {
  x = v;
  n++;
  s += v;
}
```

Constructorul clasei *Experiment* trebuie să efectueze două acţiuni suplimentare pentru fiecare nou obiect creat : incrementarea numărului total de obiecte ale clasei, precum şi adăugarea valorii curente a obiectului la suma tuturor valorilor obiectelor clasei.

Datele statice pot avea orice tip de acces (`public`, `protected`, `private`), ca orice membru al unei clase. Deşi exemplul anterior ar putea duce la concluzia că datele statice pot fi folosite oriunde într-un program, această concluzie este falsă : fiind statice, compilatorul nu permite o altă iniţializare a lor, iar modificarea valorii unor date membru statice private în afara clasei în care au fost declarate nu este permisă.

Utilizarea datelor membru statice conduce la o structurare mai bună a informației într-un program, deoarece acestea sunt globale doar pentru obiectele clasei în care au fost declarate.

În exemplul anterior, *N* şi *Med* sunt două ***funcții membru statice*** a clasei *Experiment*. Ca şi datele membru statice, funcțiile membru statice ale unei clase sunt unice pentru toate instanțele clasei respective. Utilizarea lor este necesară în cazul în care anumite clase au date membru statice.

Funcțiile membru statice nu pot utiliza parametrul ascuns `this`. Această particularitate atrage după sine o altă restricție : *funcțiile membru statice nu pot avea acces decât la membrii statici* ale clasei respective (date sau funcții). În exemplul anterior, funcția *Med* nu poate modifica valoarea membrului *x* şi nici nu poate apela alte funcții membru ale clasei (*X* de exemplu).

O modalitate prin care o funcție membru statică poate avea acces la membrii nestatici ai clasei din care face parte o constituie transmiterea ca parametru în funcția statică a unui obiect al clasei respective.

Exemplul 3.5. Clasa *Folder* memorează calea pentru un folder curent şi o cale predefinită, unică pentru toate obicetele clasei. Funcția statică *preset* permite setarea căii curente pentru un folder care este transmis ca parametru.

```
class Folder {
public:
  static void setcale(char const *calenoua);
  static void preset(Folder &dir, char const *cale);
private:
  string caleCurenta;
  static char cale[];
};

char Folder::cale[200] = "C:\\";

void Folder::setcale(char const *calenoua) {
  strcpy(cale, calenoua);
}

void Folder::preset(Folder &dir, char const *calenoua)
{
  dir.caleCurenta = calenoua;
}

void main() {
   Folder dir;
   Folder::setcale("D:\\");
   dir.setcale("D:\\");
   Folder::preset(dir, "D:\\OOP");
   dir.preset(dir, "D:\\OOP");
}
```

O altă particularitate a acestor funcţii o constituie faptul că membrii statici publici se pot apela şi direct, fără ajutorul unui obiect din care fac parte, utilizând operatorul de rezoluție. De exemplu, un fişier de utilizare pentru clasa *Experiment* ar putea fi următorul (s-a considerat şirul de valori 0.5, 1.5, 2.5, …, 9.5) :

```
// fisierul main.cpp
void main() {
  for (int i=0; i<10; i++)
    Experiment e(i+0.5);
  int n = Experiment::N();
  double m = Experiment::Med();
  cout << "n = " << n << endl << "Med = " << m << endl;
}
```

Se observă faptul că funcțiile *N* şi *Med* au fost apelate fără intermediul vreunui obiect al clasei *Experiment*.

Observații.

1. Funcțiile membru nestatice pot referi membrii statici ai clasei respective (de exemplu, cazul constructorului clasei *Experiment*).
2. O funcție membru statică privată nu poate fi apelată prin intermediul unui obiect al clasei respective.
3. Dacă o funcție membru a fost declarată `static` în cadrul unei clase, dar nu a fost definită `inline`, în definiția efectivă a acesteia nu mai trebuie specificat cuvântul `static`. De exemplu :

```
class A
{
  // ...
  static int x ;
  static void SetX(int) ;
  // ...
} ;

void A::SetX(int k)
{
  x = k ;
}
```