2. Clase. Abstractizarea datelor (II)

În capitolul precedent am introdus noțiunea de clasă şi am văzut că un tip abstract de dată reprezintă un set de date şi un set de operații care se aplică datelor. Vom prezenta în acest capitol funcțiile membre de acces, cele utilitare, constructorii şi destructorii.

- *Funcțiile de acces şi funcțiile utilitare*

Nu toate funcțiile membre trebuie să fie publice, adică să facă parte din interfața clasei. Unele pot fi *funcții utilitare* pentru alte funcții ale clasei şi rămân private.

Funcțiile membre pot fi împărțite în mai multe categorii:

- funcții care citesc şi returnează valorile datelor membre private
- funcții care setează valorile datelor membre private
- funcții care implementează serviciile clasei
- funcții care realizează operații de bază
  - inițializarea obiectelor
  - asignare a obiectelor clasei
  - conversii între clase şi tipuri predefinite sau alte clase
  - funcții de gestiune a memoriei pentru obiectele clasei
- funcții utilitare (utility functions, helper functions) care sunt apelate de alte funcții membre.

Funcțiile de acces citesc sau afişează datele. O altă utilizare comună a funcțiilor de acces este testarea valorii de adevăr a unor condiții. Acestea sunt *funcții predicat*. Un exemplu de funcție predicat este `isEmpty` care testează dacă o listă de valori este vidă. Un alt exemplu de funcție predicat este `isFull` care determină dacă într-o listă mai poate fi adăugat un nou element. Un set de funcții predicat utile clasei `Time` ar putea fi `isAM` sau `isPM`.

În exemplul următor ilustrăm noțiunea de funcție utilitară care este declarată în secțiunea `private` a clasei, nefiind accesibilă clienților clasei.

Exemplu

salesp.h

```
//Definitia clasei SalesPerson
#ifndef SALESP_H
#define SALESP_H
class SalesPerson
{
  public:
    SalesPerson();
    void getSalesFromUser();
    void setSales(int, double);
    void printAnnualSales();
  private:
    double totalAnnualSales();
    double sales[12];
};
#endif
```

salesp.cpp

```
//Functiile membre ale clasei SalesPerson
#include <iostream>
```

```
using std::cout;
using std::cin;
using std::endl;

#include <iomanip>
using std::setprecision;
using std::setiosflags;
using std::ios;

#include "salesp.h"

//Constructorul initializeaza tabloul
SalesPerson::SalesPerson()
{
  for(int i = 0; i < 12; i++)
    sales[i] = 0.0;
}
//Citeste cele 12 volume de vanzari de la tastatura
void SalesPerson::getSalesFromUser()
{
  double salesFigure;
  for(int i = 1; i <= 12; i++)
  {
    cout << "Introduceti volumul de vanzari pentru luna "
         << i << ": ";
    cin >> salesFigure;
    setSales(i, salesFigure);
  }
}
//Seteaza volumul de vanzari pentru luna selectata
void SalesPerson::setSales(int month, double amount)
{
  if(month >= 1 && month <= 12 && amount > 0)
    sales[month - 1] = amount;
  else
    cout << "Luna invalida sau volum de vanzari incorect"
         << endl;
}
//Tipareste volumul anual de vanzari
void SalesPerson::printAnnualSales()
{
  cout << setprecision(2)
       << setiosflags(ios::fixed | ios::showpoint)
       << endl << "Volumul anual de vanzari este: "
       << totalAnnualSales() << " EUR" << endl;
}
//Functie privata utilitara care calculeaza
//volumul anual de vanzari
double SalesPerson::totalAnnualSales()
{
  double total = 0.0;
```

```
  for(int i = 0; i < 12; i++)
    total += sales[i];

  return total;
};
```

test_salesp.cpp

```
#include "salesp.h"
int main()
{
  SalesPerson s;

  s.getSalesFromUser();
  s.printAnnualSales();

  return 0;
}
```

Programul de mai sus înregistrează vânzările lunare pe care le realizează un agent de vânzări al unei companii şi tipăreşte totalul anual al acestor valori. Clasa `SalesPerson` are ca dată membră un tablou care păstrează volumele lunare de vânzări de-a lungul a celor 12 luni dintr-un an. Constructorul inițializează aceste valori cu zero, în timp ce funcția membră `setSales` citeşte de la tastatură noile valori pentru volumele de vânzări şi le stochează în tabloul sales. Funcția publică `printAnnualSales` tipăreşte totalul vânzărilor folosind valoarea întoarsă de funcția utilitară `totalAnnualSales`.

O variantă de rulare a programului este următoarea:

```
Introduceti volumul de vanzari pentru luna 1: 5314.76
Introduceti volumul de vanzari pentru luna 2: 4292.38
Introduceti volumul de vanzari pentru luna 3: 4589.83
Introduceti volumul de vanzari pentru luna 4: 5534.03
Introduceti volumul de vanzari pentru luna 5: 4376.34
Introduceti volumul de vanzari pentru luna 6: 5698.45
Introduceti volumul de vanzari pentru luna 7: 4439.22
Introduceti volumul de vanzari pentru luna 8: 5893.57
Introduceti volumul de vanzari pentru luna 9: 4909.67
Introduceti volumul de vanzari pentru luna 10: 5123.45
Introduceti volumul de vanzari pentru luna 11: 4024.97
Introduceti volumul de vanzari pentru luna 12: 5923.92

Volumul anual de vanzari este: 60120.59 EUR
```

- ***Constructorii***

Constructorul este o funcție membră care nu returnează nicio valoare şi care se apelează automat atunci când într-un program se creează un obiect ale unei clase. Constructorul are acelaşi nume cu cel al clasei. Constructorii pot fi supraîncărcați pentru a oferi diverse metode de inițializare a obiectelor clasei. Datele membre pot fi inițializate prin intermediul constructorilor, dar valorile lor pot fi modificate şi ulterior prin metode ale clasei. Inițializarea obiectelor prin constructori este considerată o bună practica în ingineria software pentru că asigură stocarea unor valori valide în datele membre înaintea oricărui apel al vreunei funcții membre de către codul client.

Constructori care folosesc argumente cu valori implicite

În capitolul precedent am definit următorul constructor al clasei `Time`:

```
Time::Time()
{
  hour = minute = second = 0;
}
```

Este un *constructor implicit* pentru că poate fi invocat fără listă de argumente. Conform definiției sale, inițializează datele membre `hour`, `minute` şi `second` cu valoarea `0`. Ne propunem să modificăm semnătura acestei funcții introducând o listă de argumente cu valori implicite. Atunci când sunt transmise, valorile parametrilor vor putea înlocui valorile `0` implicite.

Exemplu

time2.h

```
#ifndef TIME2_H
#define TIME2_H
class Time
{
  public:
    Time(int = 0, int = 0, int = 0); //constructor
    void setTime(int, int, int);  //asignarea valorilor
    void printShort();  //tiparire in format scurt
    void printLong ();  //tiparire in format lung
  private:
    int hour;    //0-23
    int minute;  //0-59
    int second;  //0-59
};
#endif
```

time2.cpp

```
#include <iostream>
using std::cout;

#include "time2.h"

Time::Time(int hr, int min, int sec)
{
  setTime(hr, min, sec);
}

void Time::setTime(int h, int m, int s)
{
  hour = (h >= 0 && h < 24) ? h : 0;
  minute = (m >= 0 && m < 60) ? m : 0;
  second = (s >= 0 && s < 60) ? s : 0;
}

void Time::printShort()
{
  cout << (hour < 10 ? "0" : "") << hour << ":"
       << (minute < 10 ? "0" : "") << minute;
```

```
}

void Time::printLong()
{
  cout << ((hour == 0 || hour == 12) ?
         12 : hour % 12)
       << ":" << (minute < 10 ? "0" : "") << minute
       << ":" << (second < 10 ? "0" : "") << second
       << (hour < 12 ? " AM" : " PM");
}
```

test_time2.cpp

```
#include <iostream>
using std::cout;
using std::endl;

#include "time2.h"

int main()
{
  Time t1,             //toate argumentele implicite
       t2(2),          //minute si second implicite
       t3(21, 34),     //second implicit
       t4(12, 25, 42), //toate valorile specificate
       t5(27, 74, 99); //toate valorile eronate

  cout << "Obiect creat cu: " << endl;
  cout << "toate argumentele implicite:  " << endl;
  t1.printShort();
  cout << endl << "  ";
  t1.printLong();

  cout << "\nhour specificat; minute si second implicite:"
       << endl << "  ";
  t2.printShort();
  cout << endl << "  ";
  t2.printLong();

  cout << "\nhour si minute specificate; second implicit:"
       << endl << "  ";
  t3.printShort();
  cout << endl << "  ";
  t3.printLong();

  cout << "\nhour, minute si second specificate:"
       << endl << "  ";
  t4.printShort();
  cout << endl << "  ";
  t4.printLong();

  cout << "\nvalori invalide pentru hour, minute si second:"
       << endl << "  ";
```

```
  t5.printShort();
  cout << endl << "  ";
  t5.printLong();
  cout << endl;

  return 0;
}
```

Rulând acest exemplu obținem următorul rezultat:

```
Obiect creat cu:
toate argumentele implicite:
00:00
  12:00:00 AM
hour specificat; minute si second implicite:
  02:00
  2:00:00 AM
hour si minute specificate; second implicit:
  21:34
  9:34:00 PM
hour, minute si second specificate:
  12:25
  12:25:42 PM
valori invalide pentru hour, minute si second:
  00:00
  12:00:00 AM
```

Declarațiile obiectelor `t1`, `t2`, `t3`, `t4` şi `t5` sunt însoțite de diverse variante de apel ale constructorului clase `Time` care are argumente cu valori implicite. Parametrii actuali care sunt transmişi constructorului sunt enumerați în lista dintre parantezele rotunde care este plasată după numele obiectelor.

Constructorul introdus în acest exemplu are valori implicite pentru toate argumentele, astfel că atunci când lista de argumente este omisă complet, apelul este similar cu cel al unui constructor fără listă de argumente. Se consideră astfel că şi acesta este tot un constructor implicit. Să remarcăm faptul că o clasă poate avea un singur constructor implicit.

În programul de mai sus, toate datele membre din obiectul `t1` sunt inițializate cu valorile implicite `0`. Pentru obiectele `t2` şi `t3` doar pentru o parte dintre datele membre sunt transmise valori explicite, restul fiind inițializate cu `0`. Datele membre ale obiectului `t4` sunt inițializate cu valori transmise explicit, în timp ce pentru obiectul `t5` valorile nu sunt valide şi funcția `setTime` care este apelată de constructor le va înlocui cu valori nule.

În situația în care o clasă nu are niciun constructor, compilatorul creează un constructor implicit care, însă, nu face nicio inițializare, neexistând garanția ca, la creare, obiectul se va găsi într-o stare consistentă.

- *Destructorii*

Un destructor este o altă funcție membră specială a unei clase. Numele destructorului unei clase este format din caracterul *tilda* (~) urmat de numele clasei.

Destructorul unei clase este apelat automat atunci când un obiect al clasei este şters din memorie. Pentru obiectele de tip automatic, destructorul se invocă atunci când programul părăseşte domeniul în care a fost instanțiat obiectul.

Destructorul, aşadar, nu distruge obiectul, ci este apelat chiar înainte de a fi distrus obiectul. În această funcție, programatorul poate să includă acele operații care consideră că sunt necesare înainte de eliberarea memoriei.

Un destructor nu are niciodată parametri şi nu returnează nicio valoare. O clasă poate avea un singur destructor. Nu toate clasele au nevoie de destructori. Aceştia îşi găsesc utilitatea în special atunci când obiectele conțin referințe către zone de memorie alocate dinamic, aşa cum vom vedea într-unul dintre capitolele care urmează.

- *Apelul constructorilor şi al destructorilor*

Constructorii şi destructorii sunt apelați automat. Ordinea apelurilor depinde de ordinea în care firul execuției programului intră şi iese din domeniile în care sunt declarate obiectele. În general, apelurile destructorilor se fac în ordine inversă celei în care au fost apelați constructorii, dar sunt şi excepții de la această regulă.

Pentru obiectele declarate în domeniul global, constructorii sunt apelați înaintea apelurilor celorlalte funcții, inclusiv `main`. Destructorii acestor obiecte sunt apelați atunci când se încheie execuția funcției `main` sau când programul se termină prin apelul funcției `exit`. Destructorii nu sunt apelați dacă programul se termină apelând funcția `abort`. În cazul obiectelor automatice, destructorii nu sunt apelați dacă programul se termină prin `exit` sau `abort`. Pentru obiectele de tip `static` locale, constructorii sunt apelați doar o singură dată, la prima parcurgere a blocului în care acestea sunt declarate. Destructorii corespunzători acestor obiecte sunt apelați când funcția `main` se termină normal sau când programul se încheie prin apelul funcției `exit`, dar nu şi atunci când se încheie prin funcția `abort`.

Programul de mai jos introduce clasa `CreateAndDestroy` care are un constructor şi un destructor. Clasa este testată prin crearea şi distrugerea obiectelor declarate în diverse domenii de existență.

Exemplu

create.h

```
#ifndef CREATE_H
#define CREATE_H
class CreateAndDestroy
{
  public:
    CreateAndDestroy(int); //constructor
    ~CreateAndDestroy();   //destructor
  private:
    int data;
};
#endif
```

create.cpp

```
#include <iostream>
using std::cout;
using std::endl;

#include "create.h"

CreateAndDestroy::CreateAndDestroy(int value)
{
  data = value;
```

```
  cout << "Constructorul obiectului " << data;
}

CreateAndDestroy::~CreateAndDestroy()
{
  cout << "Destructorul obiectului " << data << endl;
}
```

test_create.cpp

```
#include <iostream>
using std::cout;
using std::endl;

#include "create.h"

void Create();
CreateAndDestroy first(1); //obiect global

int main()
{
  cout << "   (global creat inainte de main)" << endl;

  CreateAndDestroy second(2); //obiect local
  cout << "   (local automatic in main)" << endl;

  static CreateAndDestroy third(3); //obiect local
  cout << "   (local static in main)" << endl;

  Create(); //apel al functiei create

  CreateAndDestroy fourth(4); //obiect local
  cout << "   (local automatic in main)" << endl;

  return 0;
}

void Create()
{
  CreateAndDestroy fifth(5);
  cout << "   (local automatic in functia create)" << endl;

  static CreateAndDestroy sixth(6);
  cout << "   (local static in functia create)" << endl;

  CreateAndDestroy seventh(7);
  cout << "   (local automatic in functia create)" << endl;
}
```

Rulând programul obținem următorul rezultat:

```
Constructorul obiectului 1   (global creat inainte de main)
Constructorul obiectului 2   (local automatic in main)
Constructorul obiectului 3   (local static in main)
Constructorul  obiectului  5     (local  automatic  in  functia
```

```
create)
Constructorul obiectului 6   (local static in functia create)
Constructorul obiectului 7     (local automatic in functia
create)
Destructorul obiectului 7
Destructorul obiectului 5
Constructorul obiectului 4   (local automatic in main)
Destructorul obiectului 4
Destructorul obiectului 2
Destructorul obiectului 6
Destructorul obiectului 3
Destructorul obiectului 1
```

Programul defineşte obiectul `first` în domeniul global. Constructorul este apelat la începutul execuției programului, iar destructorul este apelat la încheierea programului, după ce toate celelalte obiecte au fost distruse.

Funcția `main` declară trei obiecte. Obiectele `second` şi `fourth` sunt locale şi automatice, iar obiectul `third` este local `static`. Constructorii fiecăruia dintre aceste obiecte se apelează în momentul în care execuția programlui ajunge la punctele în care sunt declarate. Destructorii obiectelor `fourth` şi `second` sunt apelați, în această ordine, când se termină funcția `main`. Obiectul `third` este `static` şi există în memoria calculatorului până la încheierea programului. Destuctorul obiectului `third` este apelat înaintrea destructorului lui `first`, dar după ce toate celelalte obiecte sunt distruse.

Funcția create declară trei obiecte, `fifth` şi `seventh` ca obiecte automatice şi locale şi `sixth` ca obiect local `static`. Destuctorii obiectelor `seventh` şi `fifth` sunt apelate în această ordine când se încheie funcția `create`. Destuctorul obiectului `sixth` este apelat înaintea destructorilor obiectelor `third` şi `first`, dar după ce toate celelalte obiecte au fost distruse.

- *Asignarea prin copierea implicită membru cu membru*

Operatorul de asignare `=` poate fi folosit pentru asignarea unui obiect altui obiect de acelaşi tip. Această asignare se face, în mod implicit, prin copiere membru cu membru, adică fiecare dată membră dintr-un obiect este copiată individual în membrul cu acelaşi nume din celălalt obiect. Copierea membru cu membru, însă, poate conduce la serioase probleme pentru clasele care conțin date membre alocate dinamic, aşa cum vom vedea într-unul dintre capitolele următoare.

Obiectele pot fi transmise funcțiilor, iar funcțiile pot returna obiecte. Aceste două operații se realizează prin transmiterea prin valoare a obiectelor, adică se transmite sau se returnează o copie a obiectului.

Când funcțiile manipulează obiecte de dimensiuni mari, transmiterea prin valoare poate duce la degradări semnificative ale performanțelor. Se preferă, în astfel de situații, transmiterea obiectelor prin referință sau prin pointer. Acest mecanism este, însă, dezavantajos din punct de vedere al securității pentru că funcția apelată are acces la obiectul original. Apelul prin referințe constante este o alternativă sigură şi convenabilă din punct de vedere al performanței.

În exemplul următor vom folosi clasa `Date` pentru a ilustra mecanismul implicit de copiere a obiectelor.

Exemplu

```
#include <iostream>
using std::cout;
using std::endl;

//Varianta simplificata a clasei Date
class Date
{
  public:
    //constructor implicit
    Date(int = 1, int = 1, int = 1990);
    void print();
  private:
    int day;
    int month;
    int year;
};

//constructor fara verificarea valorilor
Date::Date(int d, int m, int y)
{
  day = d;
  month = m;
  year = y;
}

//Tipareste data in forma zi-luna-an
void Date::print()
{
  cout << day << '-' << month << '-' << year;
}

int main()
{
  Date date1(5, 3, 2007), date2;

  cout << "date1 = ";
  date1.print();
  cout << endl << "date2 = ";
  date2.print();

  date2 = date1;//asignare prin copierea membru cu membru
  cout << endl << endl
       << "Dupa copierea membru cu membru, date2 = ";
  date2.print();
  cout << endl;

  return 0;
}
```

Rularea acestui program produce următorul rezultat:

```
date1 = 5-3-2007
date2 = 1-1-1990

Dupa copierea membru cu membru, date2 = 5-3-2007
```

- ***Reutilizarea codului***

Scrierea programelor orientate pe obiecte presupune şi implementarea unor clase care pot fi folosite de alți programatori. Oferirea în permanență de noi biblioteci de clase membrilor accesibile comunității de programatori din întreaga lume este o preocupare constantă a diverselor grupuri de dezvoltatori. Există tendința din ce în ce mai pregnantă de a dezvolta aplicații software pornind de la componente existente, testate cu atenție, bine documentate, portabile. Această manieră de reutilizare a software-ului duce la o mai mare eficiență a scrierii aplicațiilor prin reducerea semnificativă a perioadei de timp dedicate dezvoltării şi prin generarea de produse de o calitate mult mai bună. *Rapid application develpoment* (RAD) prin mecanismul componentelor reutilizabile a devenit o tehnică tot mai populară.