Manual de Informatică pentru licenţă 2020

Specializarea Matematică-Informatică

Tematica generală:

Algoritmică şi programare.

1. Căutari (secvențială și binară), sortări (selecție, bubblesort, quicksort). Metoda "Divide et Impera".
2. Algoritmi și specificări. Scrierea unui algoritm pornind de la o specificație dată. Se dă un algoritm; se cere rezultatul execuției lui.
3. Concepte OOP în limbaje de programare (Python, C++, Java, C#): Clase și obiecte, Membrii unei clase și specificatorii de acces. Constructori și destructori.
4. Relaţii între clase. Clase derivate şi relaţia de moştenire. Suprascrierea metodelor. Polimorfism. Legare dinamică. Clase abstracte şi interfeţe.

1. Căutări și sortări

1.1. Căutări

Datele se află în memoria internă, într-un şir de articole. Vom căuta un articol după un câmp al acestuia pe care îl vom considera cheie de căutare. În urma procesului de căutare va rezulta poziţia elementului căutat (dacă acesta există).

Notând cu k_1, k_2, ..., k_n cheile corespunzătoare articolelor şi cu a cheia pe care o căutăm, problema revine la a găsi (dacă există) poziţia p cu proprietatea $a = k_p$.

De obicei articolele sunt păstrate în ordinea crescătoare a cheilor, deci vom presupune că

$k_1 < k_2 < .... < k_n$.

Uneori este util să aflăm nu numai dacă există un articol cu cheia dorită ci şi să găsim în caz contrar locul în care ar trebui inserat un nou articol având cheia specificată, astfel încât să se păstreze ordinea existentă.

Deci *problema căutării* are următoarea specificare:

Date a,n,(k_i, i=1,n);

Precondiţia: $n \in N$, $n \geq 1$ şi $k_1 < k_2 < .... < k_n$;

Rezultate p;

Postcondiţia: ($p=1$ şi $a \leq k_1$) sau ($p=n+1$ şi $a > k_n$) sau ($1<p\leq n$) şi ($k_{p-1} < a \leq k_p$).

1.1.1. Căutare secvenţială

O primă metodă este căutarea **secvenţială**, în care sunt examinate succesiv toate cheile. Sunt deosebite trei cazuri: $a \leq k_1$, $a > k_n$, respectiv $k_1 < a \leq k_n$, căutarea având loc în al treilea caz.

```
Subalgoritmul CautSecv(a, n, K, p) este:          {n∈N, n≥1 şi k1 < k2 < .... < kn}
                                                  {Se caută p astfel ca: (p=1 şi a ≤ k1) sau}
                                                  { (p=n+1 şi a>kn) sau (1<p≤n) şi (kp-1 < a ≤ kp).
  Fie p := 0;                                                    {Cazul "încă negasit"}
  Dacă a ≤ k1 atunci p := 1 altfel
     Dacă a > kn atunci p := n + 1 altfel
          Pentru i := 2; n execută
               Dacă (p = 0) şi (a ≤ ki) atunci p := i sfdacă
          sfpentru
     sfdacă
  sfdacă
sf-CautSecv
```

Se observă că prin această metodă se vor executa în cel mai nefavorabil caz *n*-1 comparări, întrucât contorul *i* va lua toate valorile de la 2 la *n*. Cele *n* chei împart axa reală în *n*+1 intervale. Tot atâtea comparări se vor efectua în *n*-1 din cele n+1 intervale în care se poate afla cheia căutată, deci complexitatea medie are acelaşi ordin de mărime ca şi complexitatea în cel mai rău caz.

Evident că în multe situaţii acest algoritm face calcule inutile. Atunci când a fost deja găsită cheia dorită este inutil a parcurge ciclul pentru celelalte valori ale lui *i*. Cu alte cuvinte este posibil să înlocuim ciclul **PENTRU** cu un ciclu **CÂTTIMP**.

Ajungem la un al doilea algoritm, dat în continuare.

```
Subalgoritmul CautSucc(a, n, K, p) este:          {n∈N, n≥1 şi k1 < k2 < .... < kn}
                                                  {Se caută p astfel ca: p=1 şi a ≤ k1) sau }
                                                  {(p=n+1 şi a>kn) sau (1<p≤n) şi (kp-1 < a ≤ kp).
   Fie p:=1;
   Dacă a>k1 atunci
      Câttimp p≤n şi a>kp execută p:=p+1 sfcât
   sfdacă
sf-CautSucc
```

În cel mai rău caz şi acest algoritm face acelaşi număr de operaţii ca şi subalgoritmul *Cautsecv*. În medie numărul operaţiilor este jumătate din numărul mediu de operaţii efecuat de subalgoritmul *Cautsecv* deci complexitatea este aceeaşi. Menţionăm, că acest tip de căutare se poate aplica şi în cazul în care cheile nu sunt în ordine crescătoare.

1.1.2. Căutare binară

O altă metodă, numită **căutare binară**, care este mult mai eficientă, utilizează tehnica "divide et impera" privitor la date. Se determină în ce relaţie se află cheia articolului aflat în mijlocul colecţiei cu cheia de căutare. În urma acestei verificări căutarea se continuă doar într-o jumătate a colecţiei. În acest mod, prin înjumătăţiri succesive se micşorează volumul colecţiei rămase pentru căutare.

Căutarea binară se poate realiza practic prin apelul funcţiei `CautareBinara(a, n, K, 1, n)`, descrisă mai jos, folosită în subalgoritmul dat în continuare.

```
Subalgoritmul CautBin(a, n, K, p) este:           {n∈N, n≥1 şi k1 < k2 < .... < kn}
                                                  {Se caută  p astfel ca: (p=1 şi a ≤ k1) sau}
                                                  {(p=n+1 şi a>kn) sau (1<p≤n) şi (kp-1 < a ≤ kp)}
 Dacă a ≤ k1 atunci p := 1 altfel
     Dacă a > kn atunci p := n+1 altfel
```

```
        P := BinarySearch(a, n, K, 1, n)
    sfdacă
 sfdacă
sf-CautBin
Funcţia CautareBinara(a, n, K, St, Dr) este:
  Dacă St ≥ Dr - 1
     atunci CautareBinara := Dr
     altfel m := (St+Dr) Div 2;
       Dacă a ≤ km
          atunci CautareBinara := CautareBinara(a, n, K, St, m)
          altfel CautareBinara := CautareBinara(a, n, K, m, Dr)
       sfdacă
  sfdacă
sf-CautareBinara
```

În funcţia `CautareBinara` descrisă mai sus, variabilele `St` şi `Dr` reprezintă capetele intervalului de căutare, iar `m` reprezintă mijlocul acestui interval. Prin această metodă, într-o colecţie având `n` elemente, rezultatul căutării se poate furniza după cel mult $log_2 n$ comparări. Deci complexitatea în cel mai rău caz este direct proporţională cu $log_2 n$. Fără a insista asupra demonstraţiei, menţionăm că ordinul de mărime al complexităţii medii este acelaşi.

Se observă că funcţia `CautareBinara` se apelează recursiv. Se poate înlătura uşor recursivitatea, aşa cum se poate vedea în următoarea funcţie:

```
Funcţia CautareBinaraNerec(a, n, K, St, Dr) este:
 Câttimp Dr - St > 1 execută
         m := (St+Dr) Div 2;
         Dacă a ≤ km atunci Dr := m altfel St := m sfdacă
 sfcât
 CautareBinaraNerec := Dr
sf-CautareBinaraNerec
```

1.2. Sortări

Prin sortare internă vom înţelege o rearanjare a unei colecţii aflate în memoria internă astfel încât cheile articolelor să fie ordonate crescător (eventual descrescător).

Din punct de vedere al complexităţii algoritmilor problema revine la ordonarea cheilor. Deci specificarea problemei de **sortare internă** este următoarea:

Date n,K; *{K=(k_1,k_2,...,k_n)}*

Precondiţia: $k_i \in R$, i=1,n

Rezultate K';

Postcondiţia: K' este o permutare a lui K, dar ordonată crescător.

Deci $k_1 \le k_2 \le ... \le k_n$.

1.2.1. Sortare prin selecţie

O primă tehnică numită "*Selecţie*" se bazează pe următoarea idee: se determină poziţia elementului cu cheie de valoare minimă (respectiv maximă), după care acesta se va interschimba cu primul element. Acest procedeu se repetă pentru subcolecţia rămasă, până când mai rămâne doar elementul maxim.

```
Subalgoritmul Selectie(n, K) este:                    {Se face o permutare a celor}
                                                      {n componente ale vectorului K astfel}
                                                      {ca k1 ≤ k2 ≤ .... ≤ kn }
 Pentru i := 1; n-1 execută
   Fie ind := i;
   Pentru j := i + 1; n execută
      Dacă kj < kind atunci ind := j sfdacă
   sfpentru
   Dacă i < ind atunci t := ki; ki := kind; kind := t sfdacă
 sfpentru
sf-Selectie
```

Se observă că numărul de comparări este:

$$(n - 1) + (n - 2) + ... + 2 + 1 = n(n - 1) / 2$$

indiferent de natura datelor. Deci complexitatea medie, dar şi în cel mai rău caz este $\Theta(n^2)$.

1.2.2. Bubble sort

Metoda "*BubbleSort*", compară două câte două elemente consecutive iar în cazul în care acestea nu se află în relaţia dorită, ele vor fi interschimbate. Procesul de comparare se va încheia în momentul în care toate perechile de elemente consecutive sunt în relaţia de ordine dorită.

```
Subalgoritmul BubbleSort(n, K) este:
  Repetă
    Fie kod := 0;                                {Ipoteza "este ordine"}
    Pentru i := 2; n execută
      Dacă ki-1 > ki atunci
        t := ki-1;
```

```
        k[i-1] := k[i];
        k[i] := t;
    kod := 1                                          {N-a fost ordine!}
      sfdacă
    sfpentru
  pânăcând kod = 0 sfrep                                        {Ordonare}
sf-BubbleSort
```

Acest algoritm execută în cel mai nefavorabil caz (n-1)+(n-2)+ ... +2+1 = n(n-1)/2 comparări; complexitatea lui este $O(n^2)$.

O variantă optimizată a algoritmului "*BubbleSort*" este :

```
Subalgoritmul BubbleSort(n, K) este:
  Fie  s := 0
  Repetă
    Fie kod := 0;                                {Ipoteza "este ordine"}
    Pentru i := 2; n-s execută
      Dacă k[i-1] > k[i] atunci
        t := k[i-1];
        k[i-1] := k[i];
        k[i] := t;
    kod := 1                                          {N-a fost ordine!}
      sfdacă
    sfpentru
    s := s + 1
  pânăcând kod = 0 sfrep                                        {Ordonare}
sf-BubbleSort
```

1.2.3. Quicksort

O metodă mai performantă de ordonare, care va fi prezentată în continuare, se numeşte "*QuickSort*" şi se bazează pe tehnica "divide et impera" după cum se poate observa în continuare. Metoda este prezentată sub forma unei proceduri care realizează ordonarea unui subşir precizat prin limita inferioară şi limita superioară a indicilor acestuia.

Apelul procedurii pentru ordonarea întregului şir este : `QuickSort(n, K, 1, n)`, unde `n` reprezintă numărul de articole ale colecţiei date. Deci,

```
Subalgoritmul SortareRapidă(n, K) este:
  Cheamă QuickSort(n, K, 1, n)
sf-SortareRapidă
```

Procedura `QuickSort(n, K, St, Dr)` va realiza ordonarea subşirului k_{St}, k_{St+1}, ..., k_{Dr}. Acest subşir va fi rearanjat astfel încât k_{St} să ocupe poziţia lui finală (când şirul este ordonat). Dacă `i` este această poziţie, şirul va fi rearanjat astfel încât următoarea condiţie să fie îndeplinită:

$k_j \le k_i \le k_l$, pentru $st \le j < i < l \le dr$ (*)

Odată realizat acest lucru, în continuare va trebui doar să ordonăm subşirul k_{St}, k_{St+1}, ... ,k_{i-1} prin apelul recursiv al procedurii `QuickSort(n, K, St, i-1)` şi apoi subşirul k_{i+1}, ...,k_{Dr} prin apelul `QuickSort(n, K, i+1, Dr)`. Desigur ordonarea acestor două subşiruri (prin apelul recursiv al procedurii) mai este necesară doar dacă acestea conţin cel puţin două elemente.

Procedura `QuickSort` este prezentată în continuare :

```
Subalgoritmul QuickSort (n, K, St, Dr) este:
   Fie i := St; j := Dr; a := ki;
   Repetă
     Câttimp kj ≥ a şi (i < j) execută j := j - 1 sfcât
     ki := kj;
     Câttimp ki ≤ a şi (i < j) execută i := i + 1 sfcât
     kj := ki ;
  pânăcând i = j sfrep
  Fie ki := a;
  Dacă St < i-1 atunci Cheamă QuickSort(n, K, St, i - 1) sfdacă
  Dacă i+1 < Dr atunci Cheamă QuickSort(n, K, i + 1, Dr) sfdacă
sf-QuickSort
```

Complexitatea algoritmului prezentat este $\Theta(n^2)$ în cel mai nefavorabil caz, dar complexitatea medie este de ordinul $\boldsymbol{\Theta}(nLog_2n)$.

1.3. Metoda "divide et impera"

Strategia "Divide et Impera" în programare presupune:

- împarţirea datelor ("divide and conquer");
- împartirea problemei în subprobleme ("top-down").

Metoda se aplica problemelor care pot fi descompuse în subprobleme independente, similar problemei iniţiale, de dimensiuni mai mici şi care pot fi rezolvabile foarte uşor.

Observații:

- Împărțirea se face până când se obține o problemă rezolvabilă imediat.
- Tehnica admite și o implementare recursivă.

Formalizare

```
Sublalgoritmul NumeAlg(D) este:
    Dacă dim(D) ≤ a atunci
         @problema se rezolva
    altfel
         @ Descompune D in d1, d2,..., dk
         Cheama NumeAlg(d1)
         Cheama NumeAlg(d2)
         .
         .
         Cheama NumeAlg(dk)
         @ construieste rezultatul final prin utilizarea rezultatelor
         partiale din apelurile de mai sus
    sfdacă
sf-NumeAlg
```

2. Algoritmi şi specificări

Algoritmi și specificări. Scrierea unui algoritm pornind de la o specificație dată. Se dă un algoritm; se cere rezultatul execuției lui. Scrierea unui algoritm pornind de la o specificaţie dată

Problema 1

Scrieți o funcție care satisface următoarea specificație:

Date nr;

Precondiţia: $nr \in N, nr \geq 1$

Rezultate $l_1,l_2,...,l_n$*;*

Postcondiţia: $n \in N^*, \left[\frac{nr}{l_i}\right] \cdot l_i = nr \ \forall 1 \leq i \leq n, l_i \neq l_j \ \forall i \neq j, 1 \leq i, j \leq n$, n este maximal

Problema 2

Scrieți o funcție care satisface următoarea specificație:

Date $n, L=(l_1,l_2,...,l_n)$*;*

Precondiţia: $l_i \in R, i=1,n$

Rezultate $R=(r_1,r_2,...,r_n)$*;*

Postcondiţia: R este o permutare a lui L, $r_1 \geq r_2 \geq ... \geq r_n$.

Problema 3

Se cere să se scrie un algoritm/program pentru rezolvarea următoarei probleme: Când merge la cumpărături, Ana își pregătește tot timpul o listă de cumpărături: denumire, cantitate, raion (alimente, îmbrăcăminte, încălțăminte, consumabile), preț estimat. Se cere să se afișeze lista de cumpărături a Anei ordonată alfabetic după raion, lista ordonată descrescător după cantitate, precum și lista Anei de la un anumit raion. Se cere să se calculeze și un preț estimativ al cheltuielilor Anei.

Problema 4

Se cere să se scrie un algoritm/program pentru rezolvarea următoarei probleme: Să se scrie un program care citește un șir de numere întregi nenule. Introducerea unui șir se încheie odată cu citirea valorii 0. În șirul citit programul va elimina secvențele de elemente consecutive strict pozitive de lungime mai mare decât 3 (dacă există), dupa care va tipări șirul obținut.

Problema 5

Ce face următorul program C++?

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

bool Prim(int p){
	int d = 2;
	while ((d * d <= p) && (p % d > 0)){
		d++;
	}
	return d * d > p;
}

bool Desc(int n, int &p1, int &p2){
	p1 = 2;
	while ((p1 <= n / 2) && (!Prim(p1) || !Prim(n - p1))){
		p1++;
	}
	p2 = n - p1;
	return p1 <= n / 2;
}

int main(){
	int a;
	int b = 0;
	int c = 0;

	do{
		cout << "Dati nr numerelor: ";
		cin >> a;
		if (a > 0){
			if (Desc(a, b, c))
				cout << a << ", " << b << ", " << c << endl;
			else
				cout << "Nu ex. desc.";
		}
	} while (a > 0);
	return 0;
}
```

Problema 6

Pentru următorul program C++ precizați:

a) ce face acest program;

b) ce realizează fiecare subprogram;

c) ce rezultate sunt tipărite pentru 12 1233 1132 2338 8533 10000 21500 0 ?

```cpp
#include <iostream>
#include <fstream>
#include <set>

using namespace std;

void functie1(int x[], int &n){
	ifstream fin("date.in");
	n = 0;
	int v = 0;
	do{

		fin >> v;
		if (v > 0){
			x[n++] = v;
		}
	} while (v > 0);
	fin.close();
}

void functie2(int x[], int n){
	ofstream fout("date.out");
	for (int i = 0; i < n; i++){
		fout << x[i] << " ";
	}
	fout << endl;
	fout.close();
}

void functie3(int a, std::set<int> &s){
	s.clear();
	do{
		s.insert(a % 10);
		a /= 10;
	} while (a > 0);
}

bool functie4(int a, int b){
	std::set<int> Ma;
	std::set<int> Mb;
	functie3(a, Ma);
	functie3(b, Mb);
	return Ma == Mb;
}

void functie5(int &p){
	int v = 0;
	do{
		v = v * 10 + p % 10;
		p /= 10;
	} while (p > 0);
	p = v;
}
```

```
void functie6(int x[], int n){
	for (int i = 0; i < n - 1; i++){
		if (functie4(x[i], x[i + 1]))
			functie5(x[i]);
	}
}

int main(){
	int x[100];
	int n = 0;
	functie1(x, n);
	functie6(x, n);
	functie2(x, n);
	return 0;
}
```

```
Raspuns c) 12 3321 1132 2338 8533 10000 21500
```

Problema 7

Precizaţi ce realizează următorul program, apoi scrieţi programul C++ pentru *funcţia* inversă.

```
#include <cctype>
#include <string>
#include <iostream>

using namespace std;

char UrmL(char l){
	switch (l){
	case 'Z':
		return 'A';
	case 'z':
		return 'a';
	default:
		return l + 1;
	}
}

char ModC(char c){
	if (std::isalpha(c))
		return UrmL(c);
	else
		return c;
}

std::string Modif(std::string s){
	for (int i = 0; i < s.length(); i++){
		s[i] = ModC(s[i]);
	}
	return s;
}
```

```
int main(){
	std::string s;
	do{
		cin >> s;
		cout << Modif(s) << endl;
	} while (s != "stop");
	return 0;
}
```

Obs. Presupunând că acest program realizează o *codificare* a unui text, scrieţi programul care realizează *decodificarea*!

3. Concepte OOP în limbaje de programare

Concepte OOP în limbaje de programare (C++, Java, C#): Clase și obiecte, Membrii unei clase și specificatorii de acces, Constructori și destructori, Clase derivate și relația de moștenire, Suprascrierea metodelor, Polimorfism, Legare dinamica, Clase abstracte și interfețe

3.1. Realizarea protecţiei datelor prin metoda programării modulare

Dezvoltarea programelor prin programare procedurală înseamnă folosirea unor funcţii şi proceduri pentru scrierea programelor. În limbajul C lor le corespund funcţiile care returnează o valoare sau nu. Însă în cazul aplicaţiilor mai mari ar fi de dorit să putem realiza şi o protecţie corespunzătoare a datelor. Acest lucru ar însemna că numai o parte a funcţiilor să aibă acces la datele problemei, acelea care se referă la datele respective. Programarea modulară oferă o posibilitate de realizare a protecţiei datelor prin folosirea clasei de memorie static. Dacă într-un fişier se declară o dată aparţinând clasei de memorie statică în afara funcţiilor, atunci ea poate fi folosită începând cu locul declarării până la sfârşitul modulului respectiv, dar nu şi în afara lui.

Să considerăm următorul exemplu simplu referitor la prelucrarea vectorilor de numere întregi. Să se scrie un modul referitor la prelucrarea unui vector cu elemente întregi, cu funcţii corespunzătoare pentru iniţializarea vectorului, eliberarea zonei de memorie ocupate şi ridicarea la pătrat, respectiv afişarea elementelor vectorului. O posibilitate de implementare a modulului este prezentată în fişierul **vector1.h**:

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

static int* e;              //elementele vectorului
static int d;               //dimensiunea vectorului

void init(int* e1, int d1){  //initializare
     d = d1;
     e = new int[d];
     for (int i = 0; i < d; i++)
          e[i] = e1[i];
}

void distr(){                //eliberarea zonei de memorie ocupata
     delete[] e;
}

void lapatrat(){             //ridicare la patrat
     for (int i = 0; i < d; i++)
          e[i] *= e[i];
}
```

```
void afiseaza(){            //afisare
      for (int i = 0; i < d; i++)
            cout << e[i] << ' ';
      cout << endl;
}
```

Modulul se compilează separat obţinând un program obiect. Un exemplu de program principal este prezentat în fişierul **vector2.cpp**:

```
#include "functii.h"
extern void init(int*, int);  //extern poate fi omis
extern void distr();
extern void lapatrat();
extern void afiseaza();
//extern int* e;

int main() {
      int x[5] = { 1, 2, 3, 4, 5 };
      init(x, 5);
      lapatrat();
      afiseaza();
      distr();
      int y[] = { 1, 2, 3, 4, 5, 6 };
      init(y, 6);
      //e[1]=10;                  eroare, datele sunt protejate
      lapatrat();
      afiseaza();
      distr();
      return 0;
}
```

Observăm că deşi în programul principal se lucrează cu doi vectori nu putem să-i folosim împreună, deci de exemplu modulul **vector1.cpp** nu poate fi extins astfel încât să realizeze şi adunarea a doi vectori. În vederea înlăturării acestui neajuns s-au introdus tipurile abstracte de date.

3.2. Tipuri abstracte de date

Tipurile abstracte de date realizează o legătură mai strânsă între datele problemei şi operaţiile (funcţiile) care se referă la aceste date. Declararea unui tip abstract de date este asemănătoare cu declararea unei structuri, care în afară de date mai cuprinde şi declararea sau definirea funcţiilor referitoare la acestea.

De exemplu în cazul vectorilor cu elemente numere întregi putem declara tipul abstract:

```
struct vect {
      int* e;
      int d;
      void init(int* e1, int d1);
      void distr() { delete[] e; }
      void lapatrat();
      void afiseaza();
};
```

Funcţiile declarate sau definite în interiorul structurii vor fi numite *funcţii membru* iar datele *date membru*. Dacă o funcţie membru este definită în interiorul structurii (ca şi funcţia *distr* din exemplul de mai sus), atunci ea se consideră funcţie *inline*. Dacă o funcţie membru se defineşte în afara structurii, atunci numele funcţiei se va înlocui cu numele tipului abstract urmat de operatorul de rezoluţie (::) şi numele funcţiei membru. Astfel funcţiile *init*, *lapatrat* şi *afiseaza* vor fi definite în modul următor:

```
void vect::init(int *e1, int d1){
      d = d1;
      e = new int[d];
      for (int i = 0; i < d; i++)
            e[i] = e1[i];
}

void vect::lapatrat(){
      for (int i = 0; i < d; i++)
            e[i] *= e[i];
}

void vect::afiseaza(){
      for (int i = 0; i < d; i++)
            cout << e[i] << ' ';
      cout << endl;
}
```

Deşi prin metoda de mai sus s-a realizat o legătură între datele problemei şi funcţiile referitoare la aceste date, ele nu sunt protejate, deci pot fi accesate de orice funcţie utilizator, nu numai de funcţiile membru. Acest neajuns se poate înlătura cu ajutorul claselor.

3.3. Declararea claselor

Un tip abstract de date clasă se declară ca şi o structură, dar cuvântul cheie struct se înlocuieşte cu *class*. Ca şi în cazul structurilor referirea la tipul de dată clasă se face cu numele după cuvântul cheie class (numele clasei). Protecţia datelor se realizează cu modificatorii de protecţie: *private*, *protected* şi *public*. După modificatorul de protecţie se pune caracterul ‘:’. Modificatorul *private* şi *protected* reprezintă date protejate, iar *public* date neprotejate. Domeniul de valabilitate a modificatorilor de protecţie este până la următorul modificator din interiorul clasei, modificatorul

implicit fiind *private*. Menţionăm că şi în cazul structurilor putem să folosim modificatori de protecţie, dar în acest caz modificatorul implicit este *public*.

De exemplu clasa vector se poate declara în modul următor:

```
class vector {
  int* e;  //elementele vectorului
  int d;   //dimensiunea vectorului
public:
  vector(int* e1, int d1);
  ~vector() { delete [] e; }
  void lapatrat();
  void afiseaza();
};
```

Se observă că datele membru *e* şi *d* au fost declarate ca date de tip *private* (protejate), iar funcţiile membru au fost declarate publice (neprotejate). Bineînţeles, o parte din datele membru pot fi declarate publice, şi unele funcţii membru pot fi declarate protejate, dacă natura problemei cere acest lucru. În general, datele membru protejate pot fi accesate numai de funcţiile membru ale clasei respective şi eventual de alte funcţii numite *funcţii prietene (*sau funcţii *friend).*

O altă observaţie importantă referitoare la exemplul de mai sus este că iniţializarea datelor membru şi eliberarea zonei de memorie ocupată s-a făcut prin funcţii membru specifice.

Datele declarate cu ajutorul tipului de dată clasă se numesc *obiectele* clasei, sau simplu *obiecte*. Ele se declară în mod obişnuit în forma:

```
nume_clasă listă_de_obiecte;
```

De exemplu, un obiect de tip vector se declară în modul următor:

```
vector v;
```

Iniţializarea obiectelor se face cu o funcţie membru specifică numită *constructor*. În cazul distrugerii unui obiect se apelează automat o altă funcţie membru specifică numită *destructor*. În cazul exemplului de mai sus

```
vector(int* e1, int d1);
```

este un constructor, iar

```
~vector() { delete [] e; }
```

este un destructor.

Tipurile abstracte de date de tip *struct* pot fi şi ele considerate clase cu toate elementele neprotejate. Constructorul de mai sus este declarat în interiorul clasei, dar nu este definit, iar destructorul este definit în interiorul clasei. Rezultă că destructorul este o funcţie inline. Definirea funcţiilor membru care sunt declarate, dar nu sunt definite în interiorul clasei se face ca şi în cazul tipurilor abstracte de date de tip struct, folosind operatorul de rezoluţie.

3.3.1. Membrii unei clase. Pointerul this

Referirea la datele respectiv funcţiile membru ale claselor se face cu ajutorul operatorilor . sau -> ca şi în cazul referirii la elementele unei structuri. De exemplu, dacă se declară:

```
vector v;
vector* p;
```

atunci afişarea vectorului v respectiv a vectorului referit de pointerul p se face prin:

```
v.afiseaza();
p->afiseaza();
```

În interiorul funcţiilor membru însă referirea la datele respectiv funcţiile membru ale clasei se face simplu prin numele acestora fără a fi nevoie de operatorul punct (.) sau săgeată (->). De fapt compilatorul generează automat un pointer special, pointerul *this*, la fiecare apel de funcţie membru, şi foloseşte acest pointer pentru identificarea datelor şi funcţiilor membru.

Pointerul *this* va fi declarat automat ca pointer către obiectul curent. În cazul exemplului de mai sus pointerul *this* este adresa vectorului *v* respectiv adresa referită de pointerul *p*.

Dacă în interiorul corpului funcţiei membru *afiseaza* se utilizează de exemplu data membru *d*, atunci ea este interpretată de către compilator ca şi this->d.

Pointerul *this* poate fi folosit şi în mod explicit de către programator, dacă natura problemei necesită acest lucru.

3.3.2. Constructorul

Iniţializarea obiectelor se face cu o funcţie membru specifică numită constructor. Numele constructorului trebuie să coincidă cu numele clasei. O clasă poate să aibă mai mulţi constructori. În acest caz aceste funcţii membru au numele comun, ceea ce se poate face datorită posibilităţii de supraîncărcare a funcţiilor. Bineînţeles, în acest caz numărul şi/sau tipul parametrilor formali trebuie să fie diferit, altfel compilatorul nu poate să aleagă constructorul corespunzător.

Constructorul nu returnează o valoare. În acest caz nu este permis nici folosirea cuvântului cheie *void*.

Prezentăm în continuare un exemplu de tip clasa cu mai mulţi constructori, având ca date membru numele şi prenumele unei persoane, şi cu o funcţie membru pentru afişarea numelui complet.

Fişierul **persoana.h**:

```
class Persoana {
	char* nume;
	char* prenume;
public:
	Persoana();                       //constructor implicit
	Persoana(char* n, char* p);       //constructor
	Persoana(const Persoana& p1);     //constructor de copiere
	~Persoana();                      //destructor
	char* toString();
};
```

Fişierul **persoana.cpp**:

```
#include <iostream>
#include <cstring>
#include "Persoana.h"

using namespace std;

Persoana::Persoana(){
	nume = new char[1];
	*nume = 0;
	prenume = new char[1];
	*prenume = 0;
	cout << "Apelarea constructorului implicit." << endl;
}

Persoana::Persoana(char* n, char* p){
	nume = new char[strlen(n) + 1];
	strcpy_s(nume, strlen(n) + 1, n);

	prenume = new char[strlen(p) + 1];
	strcpy_s(prenume, strlen(p) + 1, p);

	cout << "Apelare constructor (nume, prenume).\n";
}

Persoana::Persoana(const Persoana& p1){
	nume = new char[strlen(p1.nume) + 1];
	strcpy_s(nume, strlen(p1.nume) + 1, p1.nume);

	prenume = new char[strlen(p1.prenume) + 1];
	strcpy_s(prenume, strlen(p1.prenume) + 1, p1.prenume);

	cout << "Apelarea constructorului de copiere." << endl;
}

Persoana::~Persoana(){
	delete[] nume;
	delete[] prenume;
}
```

```
char* Persoana::toString(){
	int l = strlen(prenume) + 1 + strlen(nume) + 1;
	char* s = new char[l];
	strcpy_s(s, l, prenume);
	strcat_s(s, l, "-");
	strcat_s(s, l, nume);
	strcat_s(s, l, "\0");
	return s;
}
```

Fişierul **`persoanaTest.cpp`**:

```
#include "Persoana.h"
#include <iostream>

using namespace std;

int main() {
	Persoana A;			//se apeleaza constructorul implicit
	char* s = A.toString();
	cout << s << endl;
	delete[] s;
	Persoana B("Stroustrup", "Bjarne");
	s = B.toString();
	cout << s << endl;
	delete[] s;
	Persoana* C = new Persoana("Kernighan", "Brian");
	s = C->toString();
	cout << s << endl;
	delete[] s;
	delete C;
	Persoana D(B);		//echivalent cu Persoana D = B;
	//se apeleaza constructorul de copire
	s = D.toString();
	cout << s << endl;
	delete[] s;
	return 0;
}
```

Observăm prezenţa a doi constructori specifici: *constructorul implicit* şi *constructorul de copiere*. Dacă o clasă are constructor fără parametri atunci el se va numi *constructor implicit*. *Constructorul de copiere* se foloseşte la iniţializarea obiectelor folosind un obiect de acelaşi tip (în exemplul de mai sus o persoană cu numele şi prenumele identic). Constructorul de copiere se declară în general în forma:

```
nume_clasă(const nume_clasă& obiect);
```

Cuvântul cheie *const* exprimă faptul că argumentul constructorului de copiere nu se modifică. O clasă poate să conţină ca date membru obiecte ale unei alte clase. Declarând clasa sub forma:

```
class nume_clasa {
  nume_clasa_1 ob_1;
  nume_clasa_2 ob_2;
  ...
  nume_clasa_n ob_n;
  ...
};
```

antetul constructorului clasei *nume_clasa* va fi de forma:

```
nume_clasa(lista_de_argumente):
      ob_1(l_arg_1), ob_2(l_arg_2), ..., ob_n(l_arg_n)
```

unde `lista_de_argumente` respectiv `l_arg_i` reprezintă lista parametrilor formali ai constructorului clasei `nume_clasa` respectiv ai obiectului `ob_i`.

Din lista `ob_1(l_arg_1), ob_2(l_arg_2), ..., ob_n(l_arg_n)` pot să lipsească obiectele care nu au constructori definiţi de programator, sau obiectul care se iniţializează cu un constructor implicit, sau cu toţi parametrii impliciţi.

Dacă clasa conţine date membru de tip obiect atunci se vor apela mai întâi constructorii datelor membru, iar după aceea corpul de instrucţiuni al constructorului clasei respective.

Fişierul **`pereche.cpp`**:

```
#include <iostream>
#include "Persoana.h"

using namespace std;

class Pereche {
      Persoana sot;
      Persoana sotie;
public:
      Pereche(){              //definitia constructorului implicit
            //se vor apela constructorii impliciti
      }                      //pentru obiectele sot si sotie
      Pereche(Persoana& sotul, Persoana& sotia);
      Pereche(char* nume_sot, char* prenume_sot,
            char* nume_sotie, char* prenume_sotie) :
            sot(nume_sot, prenume_sot),
            sotie(nume_sotie, prenume_sotie) {
      }
      char* toString();
};

inline Pereche::Pereche(Persoana& sotul, Persoana& sotia) :
sot(sotul), sotie(sotia){
}
```

```
char* Pereche::toString(){
	char* s_sot = sot.toString();
	char* s_sotie = sotie.toString();
	int l = 5 + strlen(s_sot) + 2 + 7 + strlen(s_sotie) + 1;
	char* s = new char[l];
	strcpy_s(s, l, "Sot: ");
	strcat_s(s, l, s_sot);
	strcat_s(s, l, "; Sotie: ");
	strcat_s(s, l, s_sotie);
	strcat_s(s, l, "\0");
	return s;
}

int main() {
	Persoana A("Pop", "Ion");
	Persoana B("Popa", "Ioana");
	Pereche AB(A, B);
	char* s = AB.toString();
	cout << s << endl;
	delete[] s;
	Pereche CD("C", "C", "D", "D");
	s = CD.toString();
	cout << s << endl;
	delete[] s;
	Pereche EF;
	s = EF.toString();
	cout << s << endl;
	delete[] s;
	return 0;
}
```

Observăm că în cazul celui de al doilea constructor, parametrii formali *sot* şi *sotie* au fost declaraţi ca şi referinţe la tipul *persoana*. Dacă ar fi fost declaraţi ca parametri formali de tip *persoana*, atunci în cazul declaraţiei:

```
pereche AB(A, B);
```

constructorul de copiere s-ar fi apelat de patru ori. În astfel de situaţii se creează mai întâi obiecte temporale folosind constructorul de copiere (două apeluri în cazul de faţă), după care se execută constructorii datelor membru de tip obiect (încă două apeluri).

3.3.3. Destructorul

Destructorul este funcţia membru care se apelează în cazul distrugerii obiectului. Destructorul obiectelor globale se apelează automat la sfârşitul funcţiei *main* ca parte a funcţiei *exit*. Deci, nu este indicată folosirea funcţiei *exit* într-un destructor, pentru că acest lucru duce la un ciclu infinit. Destructorul obiectelor locale se execută automat la terminarea blocului în care s-au definit. În cazul obiectelor alocate dinamic, de obicei destructorul se apelează indirect prin operatorul *delete* (obiectul trebuie să fi fost creat cu operatorul *new*). Există şi un mod explicit de apelare a

destructorului, în acest caz numele destructorului trebuie precedat de numele clasei şi operatorul de rezoluţie.

Numele destructorului începe cu caracterul ~ după care urmează numele clasei. Ca şi în cazul constructorului, destructorul nu returnează o valoare şi nu este permisă nici folosirea cuvântului cheie void. Apelarea destructorului în diferite situaţii este ilustrată de următorul exemplu.

Fişierul **`destructExemplu.cpp`**:

```
Persoana persGlobal("Toma", "Maria");

#include <iostream>
#include "Persoana.h"

using namespace std;

void functie(){
	cout << "Apelare functie " << endl;
	Persoana persLocal("Pop", "Ana");
}

int main() {
	Persoana* persDinamic = new Persoana("Moldovan", "Ioana");
	functie();
	cout << "Se continua programul principal" << endl;
	delete persDinamic;
	return 0;
}
```

4. Relaţii între clase

4.1. Bazele teoretice

Prin folosirea tipurilor abstracte de date, se creează un tot unitar pentru gestionarea datelor şi a operaţiilor referitoare la aceste date. Cu ajutorul tipului abstract clasă se realizează şi protecţia datelor, deci în general elementele protejate nu pot fi accesate decât de funcţiile membru ale clasei respective. Această proprietate a obiectelor se numeşte încapsulare (encapsulation).

În viaţa de zi cu zi însă ne întâlnim nu numai cu obiecte separate, dar şi cu diferite legături între aceste obiecte, respectiv între clasele din care obiectele fac parte. Astfel se formează o ierarhie de clase. Rezultă a doua proprietate a obiectelor: moştenirea (inheritance). Acest lucru înseamnă că se moştenesc toate datele şi funcţiile membru ale clasei de bază de către clasa derivată, dar se pot adăuga elemente noi (date membru şi funcţii membru) în clasa derivată. În cazul în care o clasă derivată are mai multe clase de bază se vorbeşte despre moştenire multiplă.

O altă proprietate importantă a obiectelor care aparţin clasei derivate este că funcţiile membru moştenite pot fi supraîncărcate. Acest lucru înseamnă că o operaţie referitoare la obiectele care aparţin ierarhiei are un singur identificator, dar funcţiile care descriu această operaţie pot fi diferite. Deci, numele funcţiei şi lista parametrilor formali este aceeaşi în clasa de bază şi în clasa derivată, dar descrierea funcţiilor diferă între ele. Astfel, în clasa derivată funcţiile membru pot fi specifice clasei respective, deşi operaţia se identifică prin acelaşi nume. Această proprietate se numeşte polimorfism.

4.2. Declararea claselor derivate

O clasă derivată se declară în felul următor:

```
class nume_clasă_derivată : lista_claselor_de_bază {
  //date membru noi şi funcţii membru noi
};
```

unde lista_claselor_de_bază este de forma: $elem_1$, $elem_2$, ..., $elem_n$ şi $elem_i$ pentru orice $1 \leq i \leq n$ poate fi

```
public clasă_de_bază_i
```

sau

```
protected clasă_de_bază_i
```

sau

```
private clasă_de_bază_i
```

Cuvintele cheie *public*, *protected* şi *private* se numesc şi de această dată modificatori de *protecţie*. Ei pot să lipsească, în acest caz modificatorul implicit fiind private. Accesul la elementele din clasa derivată este prezentat în Tabel 1.

Accesul la elementele din clasa de bază	**Modificatorii de protecţie referitoare la clasa de bază**	**Accesul la elementele din clasa derivată**
public	public	public
protected	public	protected
private	public	inaccesibil
public	protected	protected
protected	protected	protected
private	protected	inaccesibil
public	private	private
protected	private	private
private	private	inaccesibil

Tabel 1: *Accesul la elementele din clasa derivată*

Observăm că elementele de tip *private* ale clasei de bază sunt inaccesibile în clasa derivată. Elementele de tip *protected* şi *public* devin de tip *protected*, respectiv *private* dacă modificatorul de protecţie referitor la clasa de bază este *protected* respectiv *private*, şi rămân neschimbate dacă modificatorul de protecţie referitor la clasa de bază este *public*. Din acest motiv în general datele membru se declară de tip *protected* şi modificatorul de protecţie referitor la clasa de bază este *public*. Astfel datele membru pot fi accesate, dar rămân protejate şi în clasa derivată.

4.3. Funcţii virtuale

Noţiunea de polimorfism ne conduce în mod firesc la problematica determinării funcţiei membru care se va apela în cazul unui obiect concret. Să considerăm următorul exemplu. Declarăm clasa de bază *baza*, şi o clasă derivată din acestă clasă de bază, clasa *derivata*. Clasa de bază are două funcţii membru: *functia_1* şi *functia_2*. În interiorul funcţiei *membru functia_2* se apelează *functia_1*. În clasa derivată se supraîncarcă funcţia membru *functia_1*, dar funcţia membru *functia_2* nu se supraîncarcă. În programul principal se declară un obiect al clasei derivate şi se

apelează funcţia membru *functia_2* moştenită de la clasa de bază. În limbajul C++ acest exemplu se scrie în următoarea formă.

Fişierul `virtual1.cpp`:

```
#include <iostream>

using namespace std;

class baza {
public:
	void functia_1();
	void functia_2();
};

class derivata : public baza {
public:
	void functia_1();
};

void baza::functia_1() {
	cout << "S-a apelat functia membru functia_1" << " a clasei de baza" << endl;
}

void baza::functia_2() {
	cout << "S-a apelat functia membru functia_2" << " a clasei de baza" << endl;
	functia_1();
}

void derivata::functia_1() {
	cout << "S-a apelat functia membru functia_1" << " a clasei derivate" << endl;
}

int main() {
	derivata D;
	D.functia_2();
	return 0;
}
```

Prin execuţie se obţine următorul rezultat:

```
S-a apelat functia membru functia_2 a clasei de baza
S-a apelat functia membru functia_1 a clasei de baza
```

Însă acest lucru nu este rezultatul dorit, deoarece în cadrul funcţiei *main* s-a apelat funcţia membru *functia_2* moştenită de la clasa de bază, dar funcţia membru *functia_1* apelată de *functia_2* s-a determinat încă în faza de compilare. În consecinţă, deşi funcţia membru *functia_1* s-a supraîncărcat în clasa derivată nu s-a apelat funcţia supraîncărcată ci funcţia membru a clasei de bază.

Acest neajuns se poate înlătura cu ajutorul introducerii noţiunii de funcţie membru *virtuală*. Dacă funcţia membru este virtuală, atunci la orice apelare a ei, determinarea funcţiei membru corespunzătoare a ierarhiei de clase nu se va face la compilare ci la execuţie, în funcţie de natura obiectului pentru care s-a făcut apelarea. Această proprietate se numeşte *legare dinamică*, iar dacă determinarea funcţiei membru se face la compilare, atunci se vorbeşte de legare statică.

Am văzut că dacă se execută programul `virtual1.cpp` se apelează funcţiile membru *functia_1* şi *functia_2* ale clasei de bază. Însă funcţia membru *functia_1* fiind supraîncărcată în clasa derivată, ar fi de dorit ca funcţia supraîncărcată să fie apelată în loc de cea a clasei de bază.

Acest lucru se poate realiza declarând *functia_1* ca funcţie membru virtuală. Astfel, pentru orice apelare a funcţiei membru functia_1, determinarea acelui exemplar al funcţiei membru din ierarhia de clase care se va executa, se va face la execuţie şi nu la compilare. Ca urmare, funcţia membru functia_1 se determină prin legare dinamică.

În limbajul C++ o funcţie membru se declară virtuală în cadrul declarării clasei respective în modul următor: antetul funcţiei membru se va începe cu cuvântul cheie virtual.

Dacă o funcţie membru se declară virtuală în clasa de bază, atunci supraîncărcările ei se vor considera virtuale în toate clasele derivate ale ierarhiei.

În cazul exemplului de mai sus declararea clasei de bază se modifică în felul următor.

```
class baza {
public:
     virtual void functia_1();
     void functia_2();
};
```

Rezultatul obţinut prin execuţie se modifică astfel:

```
S-a apelat functia membru functia_2 a clasei de baza
S-a apelat functia membru functia_1 a clasei derivate
```

Deci, într-adevăr se apelează funcţia membru *functia_1* a clasei derivate. Prezentăm în continuare un alt exemplu în care apare necesitatea introducerii funcţiilor membru virtuale. Să se definească clasa *fractie* referitoare la numerele raţionale, având ca date membru numărătorul şi numitorul fracţiei. Clasa trebuie să aibă un constructor, valoarea implicită pentru numărător fiind zero, iar pentru numitor unu, precum şi două funcţii membru: *produs* pentru a calcula produsul a două fracţii şi *inmulteste* pentru înmulţirea obiectului curent cu fracţia dată ca şi parametru. De asemenea, clasa *fractie* trebuie să aibă şi o funcţie membru pentru afişarea unui număr raţional. Folosind clasa *fractie* ca şi clasă de bază se va defini clasa derivată *fractie_scrie*, pentru care se va supraîncărca funcţia *produs*, astfel încât concomitent cu efectuarea înmulţirii să se afişeze pe *stdout*

operaţia respectivă. Funcţia *inmulteste* nu se va supraîncărca, dar operaţia efectuată trebuie să se afişeze pe dispozitivul standard de ieşire şi în acest caz.

Fişierul `fvirt1.cpp`:

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

class fractie {
protected:
	int numarator;
	int numitor;
public:
	fractie(int numarator1 = 0, int numitor1 = 1);
	fractie produs(fractie& r); //calculeaza produsul a doua fractii, dar nu simplifica
	fractie& inmulteste(fractie& r);
	std::string toString();
};

fractie::fractie(int numarator1, int numitor1) {
	numarator = numarator1;
	numitor = numitor1;
}

fractie fractie::produs(fractie& r) {
	return fractie(numarator * r.numarator, numitor * r.numitor);
}

fractie& fractie::inmulteste(fractie& q) {
	*this = this->produs(q);
	return *this;
}

std::string fractie::toString() {
	char* s = new char[5 + 3 + 5 + 1];
	if (numitor){
		std::string s1 = std::to_string(numarator);
		std::string s2 = std::to_string(numitor);
		std::string s = s1 + " / " + s2;
		return s;
	}
	else
		return "Fractie incorecta";
}


class fractie_scrie : public fractie{
public:
	fractie_scrie(int numarator1 = 0, int numitor1 = 1);
	fractie produs(fractie& r);
};
```

```
inline fractie_scrie::fractie_scrie(int numarator1, int numitor1) : fractie(numarator1,
numitor1) {
}

fractie fractie_scrie::produs(fractie& q) {
	fractie r = fractie(*this).produs(q);
	cout << "(" << this->toString() << ") * (" << q.toString() << ") = " << r.toString()
<< endl;
	return r;
}

int main() {
	fractie p(3, 4), q(5, 2), r;
	r = p.inmulteste(q);
	cout << p.toString() << endl;
	cout << r.toString() << endl;
	fractie_scrie p1(3, 4), q1(5, 2);
	fractie r1, r2;
	r1 = p1.produs(q1);
	r2 = p1.inmulteste(q1);
	cout << p1.toString() << endl;
	cout << r1.toString() << endl;
	cout << r2.toString() << endl;
	return 0;
}
```

Prin execuţie se obţine:

```
15 / 8
15 / 8
(3 / 4) * (5 / 2) = 15 / 8
15 / 8
15 / 8
15 / 8
```

Observăm că rezultatul nu este cel dorit, deoarece operaţia de înmulţire s-a afişat numai o singură dată, şi anume pentru expresia `r1 = p1.produs(q1)`. În cazul expresiei `r2 = p1.inmulteste(q1)` însă nu s-a afişat operaţia de înmulţire. Acest lucru se datorează faptului că funcţia membru *inmulteste* nu s-a supraîncărcat în clasa derivată. Deci s-a apelat funcţia moştenită de la clasa *fractie*. În interiorul funcţiei *inmulteste* s-a apelat funcţia membru *produs*, dar deoarece această funcţie membru s-a determinat încă în faza de compilare, rezultă că s-a apelat funcţia referitoare la clasa *fractie* şi nu cea referitoare la clasa derivată *fractie_scrie*. Deci, afişarea operaţiei s-a efectuat numai o singură dată. Soluţia este, ca şi în exemplul anterior, declararea unei funcţii membru virtuale, şi anume funcţia *produs* se va declara ca funcţie virtuală.

Declararea clasei de bază se modifică în felul următor:

```
class fractie {
protected:
	int numarator;
	int numitor;
public:
	fractie(int numarator1 = 0, int numitor1 = 1);
	virtual fractie produs(fractie& r); //calculeaza produsul a doua fractii, dar nu
                                        //simplifica
	fractie& inmulteste(fractie& r);
	std::string toString();
};
```

După efectuarea acestei modificări prin executarea programului obţinem:

```
15 / 8
15 / 8
(3 / 4) * (5 / 2) = 15 / 8
(3 / 4) * (5 / 2) = 15 / 8
15 / 8
15 / 8
15 / 8
```

Deci, se observă că afişarea operaţiei s-a făcut de două ori, pentru ambele expresii. Funcţiile virtuale, ca şi alte funcţii membru de fapt, nu trebuie neapărat supraîncărcate în clasele derivate. Dacă nu sunt supraîncărcate atunci se moşteneşte funcţia membru de la un nivel superior.

Determinarea funcţiilor membru virtuale corespunzătoare se face pe baza unor tabele construite şi gestionate în mod automat. Obiectele claselor care au funcţii membru virtuale conţin şi un pointer către tabela construită. De aceea gestionarea funcţiilor membru virtuale necesită mai multă memorie şi un timp de execuţie mai îndelungat.

4.4. Clase abstracte

În cazul unei ierarhii de clase mai complicate, clasa de bază poate avea nişte proprietăţi generale despre care ştim, dar nu le putem defini numai în clasele derivate. De exemplu să considerăm ierarhia de clase din Figura 1

Observăm că putem determina nişte proprietăţi referitoare la clasele derivate. De exemplu greutatea medie, durata medie de viaţă şi viteza medie de deplasare. Aceste proprietăţi se vor descrie cu ajutorul unor funcţii membru. În principiu şi pentru clasa *animal* există o greutate medie, durată medie de viaţă şi viteză medie de deplasare. Dar aceste proprietăţi ar fi mult mai greu de determinat şi ele nici nu sunt importante pentru noi într-o generalitate de acest fel. Totuşi pentru o tratare generală ar fi bine, dacă cele trei funcţii membru ar fi declarate în clasa de bază şi definite în clasele derivate. În acest scop s-a introdus noţiunea de *funcţie membru virtuală pură*.

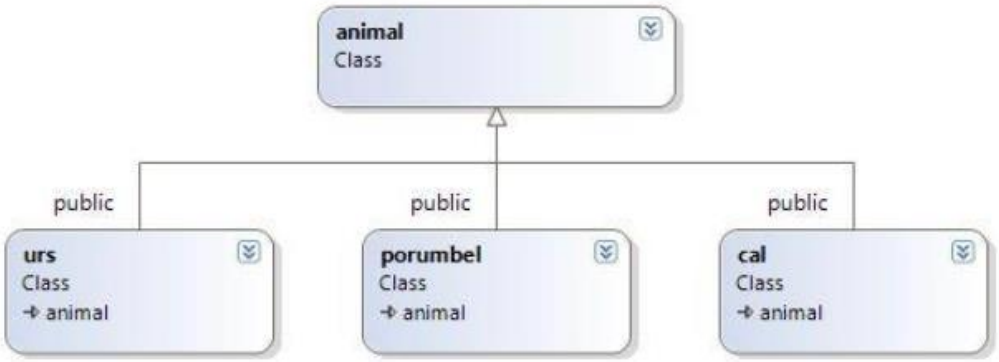


Figura 1. *Ierarhie de clase referitoare la animale*

Funcţia virtuală pură este o funcţie membru care este declarată, dar nu este definită în clasa respectivă. Ea *trebuie* definită într-o clasă derivată. Funcţia membru virtuală pură se declară în modul următor. Antetul obişnuit al funcţiei este precedat de cuvântul cheie *virtual*, şi antetul se termină cu = *0*. După cum arată numele şi declaraţia ei, funcţia membru virtuală pură este o funcţie virtuală, deci selectarea exemplarului funcţiei din ierarhia de clase se va face în timpul execuţiei programului.

Clasele care conţin cel puţin o funcţie membru virtuală pură se vor numi *clase abstracte*. Deoarece clasele abstracte conţin funcţii membru care nu sunt definite, nu se pot crea obiecte aparţinând claselor abstracte. Dacă funcţia virtuală pură nu s-a definit în clasa derivată atunci şi clasa derivată va fi clasă abstractă şi ca atare nu se pot defini obiecte aparţinând acelei clase.

Să considerăm exemplul de mai sus şi să scriem un program, care referitor la un *porumbel*, *urs* sau *cal* determină dacă el este gras sau slab, rapid sau încet, respectiv tânăr sau bătrân. Afişarea acestui rezultat se va face de către o funcţie membru a clasei *animal* care nu se supraîncarcă în clasele derivate.

Fişierul `abstract1.cpp`:

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

class animal {
protected:
	double greutate; // kg
	double virsta; // ani
	double viteza; // km / h
public:
	animal(double g, double v1, double v2);
	virtual double greutate_medie() = 0;
```

```
	virtual double durata_de_viata_medie() = 0;
	virtual double viteza_medie() = 0;
	int gras() {
		return greutate > greutate_medie();
	}
	int rapid() {
		return viteza > viteza_medie();
	}
	int tanar() {
		return 2 * virsta < durata_de_viata_medie();
	}
	std::string toString();
};

animal::animal(double g, double v1, double v2){
	greutate = g;
	virsta = v1;
	viteza = v2;
}

std::string animal::toString(){
	return gras() ? "gras, " : "slab, ";
	return tanar() ? "tanar, " : "batran, ";
	return rapid() ? "rapid" : "incet";
}


class porumbel : public animal {
public:
	porumbel(double g, double v1, double v2) : animal(g, v1, v2) {}
	double greutate_medie() { return 0.5; }
	double durata_de_viata_medie() { return 6; }
	double viteza_medie() { return 90; }
};


class urs : public animal {
public:
	urs(double g, double v1, double v2) : animal(g, v1, v2) {}
	double greutate_medie() { return 450; }
	double durata_de_viata_medie() { return 43; }
	double viteza_medie() { return 40; }
};


class cal : public animal {
public:
	cal(double g, double v1, double v2) : animal(g, v1, v2) {}
	double greutate_medie() { return 1000; }
	double durata_de_viata_medie() { return 36; }
	double viteza_medie() { return 60; }
};
```

```
int main() {
	porumbel p(0.6, 1, 80);
	urs u(500, 40, 46);
	cal c(900, 8, 70);
	cout << p.toString() << endl;
	cout << u.toString() << endl;
	cout << c.toString() << endl;
	return 0;
}
```

Observăm că deşi clasa *animal* este clasă abstractă, este utilă introducerea ei, pentru că multe funcţii membru pot fi definite în clasa de bază şi moştenite fără modificări în cele trei clase derivate.

4.5. Interfeţe

În limbajul C++ nu s-a definit noţiunea de interfaţă, care există în limbajele Java sau C#. Dar orice clasă abstractă, care conţine numai funcţii virtuale pure, se poate considera o interfaţă. Bineînţeles, în acest caz nu se vor declara nici date membru în interiorul clasei. Clasa abstractă *animal* conţine atât date membru, cât şi funcţii membru nevirtuale, deci ea nu se poate considera ca şi un exemplu de interfaţă.

În continuare introducem o clasă abstractă *Vehicul*, care nu conţine numai funcţii membru virtuale pure, şi două clase derivate din această clasă abstractă. Fişierul `vehicul.cpp`:

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

class Vehicul {
public:
	virtual void Porneste() = 0;
	virtual void Opreste() = 0;
	virtual void Merge(int km) = 0;
	virtual void Stationeaza(int min) = 0;
};

class Bicicleta : public Vehicul {
public:
	void Porneste();
	void Opreste();
	void Merge(int km);
	void Stationeaza(int min);
};

void Bicicleta::Porneste() {
	cout << "Bicicleta porneste." << endl;
}

void Bicicleta::Opreste() {
	cout << "Bicicleta se opreste." << endl;
}
```

```
void Bicicleta::Merge(int km) {
	cout << "Bicicleta merge " << km << " kilometri." << endl;
}

void Bicicleta::Stationeaza(int min) {
	cout << "Bicicleta stationeaza " << min << " minute." << endl;
}

class Masina : public Vehicul{
public:
	void Porneste();
	void Opreste();
	void Merge(int km);
	void Stationeaza(int min);
};

void Masina::Porneste() {
	cout << "Masina porneste." << endl;
}

void Masina::Opreste() {
	cout << "Masina se opreste." << endl;
}

void Masina::Merge(int km) {
	cout << "Masina merge " << km << " kilometri." << endl;
}

void Masina::Stationeaza(int min) {
	cout << "Masina stationeaza " << min << " minute." << endl;
}

void Traseu(Vehicul *v) {
	v->Porneste();
	v->Merge(3);
	v->Stationeaza(2);
	v->Merge(2);
	v->Opreste();
}

int main() {
	Vehicul *b = new Bicicleta;
	Traseu(b);
	Vehicul *m = new Masina;
	Traseu(m);
	delete m;
	delete b;
}
```

În funcția *main* s-au declarat două obiecte dinamice de tip *Bicicleta*, respectiv *Masina*, și în acest fel, apelând funcția *Traseu* obținem rezultate diferite, deși această funcție are ca parametru formal numai un pointer către o clasă abstractă *Vehicul*.

5. Probleme propuse

1. Scrieți un program într-unul din limbajele de programare Python, C++, Java, C# care:
    a. Definește o clasă **Student** având:
        - un atribut ***nume*** de tip șir de caractere;
        - un atribut ***note*** conținând un șir de note (numere întregi),

       constructori, accesori și o metodă care calculează media notelor studentului.
    b. Definește o funcție care primind un obiect de tip **Student** returnează adevărat dacă toate notele elevului sunt >4.
    c. Scrieți specificațiile metodelor definite în clasa **Student** precum și a funcției de la punctul b.

2. Scrieți un program într-unul din limbajele de programare Python, C++, Java, C# care:
    a. Definește o clasă **Student** având:
        - un atribut ***nume*** de tip șir de caractere;
        - un atribut ***note*** conținând un șir de note (numere întregi),

       constructori, accesori și o metodă care calculează media notelor studentului.
    b. Definește un subprogram care primind un obiect de tip **Student** tipărește numele studentului și notele acestuia în ordine descrescătoare.
    c. Scrieți specificațiile metodelor definite în clasa **Student** precum și a subprogramului de la punctul b.

3. Scrieți un program într-unul din limbajele de programare Python, C++, Java, C# care:
    a. Definește o clasă **Punct2D** având ca atribute protejate:
        - ***denumire*** de tip caracter;
        - ***coordonataX*** de tip număr real;
        - ***coordonataY*** de tip număr real,

       iar ca metode publice:
        - constructor cu parametri pentru inițializarea tuturor atributelor;
        - metoda ***toString*** care returnează următoarea reprezentare sub forma unui șir de caractere: *denumire(coordonataX, coordonataY)* de ex. *A(2, 3)*;
        - metoda fără parametri ***distanțaPânăLaOrigine*** care calculează și returnează distanța Euclideana dintre punctul curent și originea (0, 0) a unui sistem de coordonate 2D.

b. Definește o clasă **MulțimeDePuncte** având ca atribute private:
   - ***nrPuncte*** de tip întreg;
   - ***puncte*** de tip tablou (șir) cu elemente de tipul **Punct2D**,

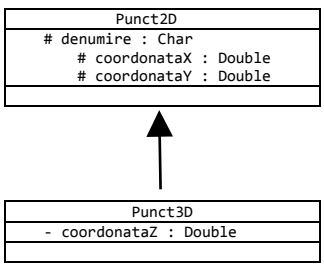
   iar ca metode publice:
   - constructor fără parametri;
   - metode accesor de tip get pentru ambele atribute,
   - metoda ***add(p)*** pentru adăugarea unui punct *p* în tabloul *puncte*,
   - metoda ***filtruPuncte(limită)*** unde parametrul *limită* este un număr real, care păstrează în mulțimea de puncte doar pe acelea care au distanța față de origine mai mare decât *limită*,
   - metoda ***sortare*** care sortează alfabetic crescător după câmpul *nume* punctele din mulțime.

c. Definește o funcție ***afișare(mulțime)***, unde parametrul *mulțime* este de tipul **MulțimeDePuncte**, care afișează la ieșirea standard punctele din *mulțime*.

d. Definește o funcție ***prelucrare1()*** care:
   - construiește o mulțime de tipul **MulțimeDePuncte**, formată din următoarele puncte: C(1,2), A(2,3), B(1,2), A(2,4), D(2,5),
   - sortează aceste puncte după denumire (folosind metoda ***sortare***), și
   - afișează această mulțime ordonată (folosind funcția ***afișare***).

e. Definește o clasă **Punct3D** derivată din clasa **Punct2D** având ca atribut privat:
   - **coordonataZ** de tip număr real,

   iar ca metode publice:
   - constructor cu parametri pentru inițializarea tuturor atributulelor;
   - metoda **toString** care returnează următoarea reprezentare sub forma unui șir de caractere: *denumire(coordonataX, coordonataY, coordonatăZ)*, de ex. *B(5,2,4)*;
   - metoda fără parametri **distanțaPânăLaOrigine** care calculează și returnează distanța Euclideana dintre punctul curent și originea (0, 0, 0) a unui sistem de coordonate 3D.



f. Definește o funcție **prelucrare2(val)** care:
   - construiește o mulțime de tipul **MulțimeDePuncte**, formată din următoarele puncte: A(1,2), B(1,2,3), C(1,2), D(3,4,5);
   - determină și afișează punctele din mulțime aflate la o distanță față de origine mai mare decât *val*.

g. Definește funcția principală a programului și:
   - apelează funcția ***prelucrare1()***;
   - apelează funcția ***prelucrare2(10)***.

Bibliografie generală

1. Alexandrescu, *Programarea modernă in C++. Programare generică si modele de proiectare aplicate*, Editura Teora, 2002.
2. Angster Erzsébet: *Objektumorientált tervezés és programozás Java*, 4KÖR Bt, 2003.
3. Bjarne Stroustrup: *A C++ programozási nyelv*, Kiskapu kiadó, Budapest, 2001.
4. Bjarne Stroustrup: *The C++ Programming Language Special Edition*, AT&T, 2000.
5. Boian F.M. Frentiu M., Lazăr I. Tambulea L. *Informatica de bază*. Presa Universitară Clujeana, Cluj, 2005
6. Bradley L. Jones: *C# mesteri szinten 21 nap alatt*, Kiskapu kiadó, Budapest, 2004.
7. Bradley L. Jones: *SAMS Teach Yourself the C# Language in 21 Days*, Pearson Education,2004.
8. Cormen, T., Leiserson, C., Rivest, R., *Introducere în algoritmi*, Editura Computer Libris Agora, Cluj, 2000
9. Eckel B., *Thinking in C++*, vol I-II, http://www.mindview.net
10. Ellis M.A., Stroustrup B., *The annotated C++ Reference Manual*, Addison-Wesley, 1995
11. Frentiu M., Lazăr I. *Bazele programării*. Partea I-a: Proiectarea algoritmilor
12. Herbert Schildt: Java. *The Complete Reference*, Eighth Edition, McGraw-Hill, 2011.
13. Robert Sedgewick: *Algorithms*, Addison-Wesley, 1984
14. Simon Károly, *Kenyerünk Java. A Java programozás alapjai*, Presa Universitară Clujeană, 2010.