

Manual de Informatică pentru licență 2018

Specializarea Matematică-Informatică

Tematica generală:

Algoritmică și programare.

1. Căutari (secvențială și binară), sortări (selecție, bubblesort, quicksort).
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. Relații între clase. Clase derivate și relația de moștenire. Suprascrierea metodelor. Polimorfism. Legare dinamică. Clase abstracte și interfețe.

1. Căutări și sortări

1.1. Căutări

Datele se află în memoria internă, într-un șir 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 poziția elementului căutat (dacă acesta există).

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

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

$$k_1 < k_2 < \dots < k_n.$$

Uneori este util să aflăm nu numai dacă există un articol cu cheia dorită ci și 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 $a, n, (k_i, i=1, n)$;

Precondiția: $n \in \mathbb{N}, n \geq 1$ și $k_1 < k_2 < \dots < k_n$;

Rezultate p ;

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

1.1.1. Căutare secvențială

O primă metodă este căutarea **secvențială**, 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 și k1 < k2 < ... < kn}
                                              {Se caută p astfel ca: (p=1 și a ≤ k1) sau}
                                              {(p=n+1 și a > kn) sau (1 < p ≤ n) și (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) și (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 $n-1$ comparații, întrucât contorul i va lua toate valorile de la 2 la n . Cele n chei împart axa reală în $n+1$ intervale. Tot atâtea comparații se vor efectua în $n-1$ din cele $n+1$ intervale în care se poate afla cheia căutată, deci complexitatea medie are același ordin de mărime ca și complexitatea în cel mai rău caz.

Evident că în multe situații 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: $\{n \in \mathbb{N}, n \geq 1 \text{ și } k_1 < k_2 < \dots < k_n\}$
 {Se caută p astfel ca: $p=1$ și $a \leq k_1$ } sau }
 {($p=n+1$ și $a > k_n$) sau ($1 < p \leq n$) și ($k_{p-1} < a \leq k_p$)}.

```
Fie p:=1;
Dacă a>k1 atunci
    Cât timp p≤n și a>kp execută p:=p+1 sf-cât
sf-dacă
sf-CautSucc
```

În cel mai rău caz și acest algoritm face același număr de operații ca și subalgoritmul *Cautsecv*. În medie numărul operațiilor este jumătate din numărul mediu de operații efectuate de subalgoritmul *Cautsecv* deci complexitatea este aceeași. Menționăm, că acest tip de căutare se poate aplica și 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 relație se află cheia articolului aflat în mijlocul colecției cu cheia de căutare. În urma acestei verificări căutarea se continuă doar într-o jumătate a colecției. În acest mod, prin înjumătățiri succesive se micșorează volumul colecției rămase pentru căutare. Căutarea binară se poate realiza practic prin apelul funcției *CautareBinara(a, n, K, 1, n)*, descrisă mai jos, folosită în subalgoritmul dat în continuare.

Subalgoritmul CautBin(a, n, K, p) este: $\{n \in \mathbb{N}, n \geq 1 \text{ și } k_1 < k_2 < \dots < k_n\}$
 {Se caută p astfel ca: ($p=1$ și $a \leq k_1$) sau }
 {($p=n+1$ și $a > k_n$) sau ($1 < p \leq n$) și ($k_{p-1} < a \leq k_p$)}

```
Dacă a ≤ k1 atunci p := 1 altfel
Dacă a > kn atunci p := n+1 altfel
P := BinarySearch(a, n, K, 1, n)
sf-dacă
sf-dacă
sf-CautBin
```

Funcția `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 funcția `CautareBinara` descrisă mai sus, variabilele `St` și `Dr` reprezintă capetele intervalului de căutare, iar `m` reprezintă mijlocul acestui interval. Prin această metodă, într-o colecție având n elemente, rezultatul căutării se poate furniza după cel mult $\log_2 n$ comparații. Deci complexitatea în cel mai rău caz este direct proporțională cu $\log_2 n$. Fără a insista asupra demonstrației, menționăm că ordinul de mărime al complexității medii este același.

Se observă că funcția `CautareBinara` se apelează recursiv. Se poate înlătura ușor recursivitatea, așa cum se poate vedea în următoarea funcție:

```
Funcția CautareBinaraNerec(a, n, K, St, Dr) este:
Cât timp 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 înțelege o rearanjare a unei colecții aflate în memoria internă astfel încât cheile articolelor să fie ordonate crescător (eventual descrescător).

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

Date $n, K;$ $\{K=(k_1, k_2, \dots, k_n)\}$

Precondiția: $k_i \in R, i=1, n$

Rezultate $K';$

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

Deci $k_1 \leq k_2 \leq \dots \leq k_n$.

1.2.1. Sortare prin selecție

O primă tehnică numită "*Selecție*" se bazează pe următoarea idee: se determină poziția elementului cu cheie de valoare minimă (respectiv maximă), după care acesta se va interschimba cu primul element. Acest procedeu se repetă pentru subcolecția 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  $k_1 \leq k_2 \leq \dots \leq k_n$  }

Pentru i := 1; n-1 execută
  Fie ind := i;
  Pentru j := i + 1; n execută
    Dacă  $k_j < k_{ind}$  atunci ind := j sfdacă
  sfpentru
  Dacă i < ind atunci t :=  $k_i$ ;  $k_i := k_{ind}$ ;  $k_{ind} := t$  sfdacă
  sfpentru
sf-Selectie
```

Se observă că numărul de comparații este:

$$(n - 1) + (n - 2) + \dots + 2 + 1 = n(n - 1) / 2$$

indiferent de natura datelor. Deci complexitatea medie, dar și în cel mai rău caz este $\mathcal{O}(n^2)$.

1.2.2. Bubble sort

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

```
Subalgoritmul BubbleSort(n, K) este:
Repetă
  Fie kod := 0;                                                     {Ipoteza "este ordine"}
  Pentru i := 2; n 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
  pânăcând kod = 0 sfrep                                           {Ordonare}
sf-BubbleSort
```

Acest algoritm execută în cel mai nefavorabil caz $(n-1)+(n-2)+ \dots +2+1 = n(n-1)/2$ comparații; 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 numește "QuickSort" și 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 subșir precizat prin limita inferioară și limita superioară a indicilor acestuia. Apelul procedurii pentru ordonarea întregului șir este : QuickSort (n, K, 1, n), unde n reprezintă numărul de articole ale colecției date. Deci

Subalgoritmul SortareRapidă (n, K) este:

```

Cheamă QuickSort (n, K, 1, n)
sf-SortareRapidă

```

Procedura QuickSort (n, K, St, Dr) va realiza ordonarea subșirului $k_{St}, k_{St+1}, \dots, k_{Dr}$. Acest subșir va fi rearanjat astfel încât k_{St} să ocupe poziția lui finală (când șirul este ordonat). Dacă i este această poziție, șirul va fi rearanjat astfel încât următoarea condiție să fie îndeplinită:

$$k_j \leq k_i \leq k_l, \text{ pentru } st \leq j < i < l \leq dr \quad (*)$$

Odată realizat acest lucru, în continuare va trebui doar să ordonăm subșirul $k_{St}, k_{St+1}, \dots, k_{i-1}$ prin apelul recursiv al procedurii QuickSort (n, K, St, i-1) și apoi subșirul k_{i+1}, \dots, k_{Dr}

prin apelul `QuickSort(n, K, i+1, Dr)`. Desigur ordonarea acestor două subșiruri (prin apelul recursiv al procedurii) mai este necesară doar dacă acestea conțin cel puțin două elemente.

Procedura `QuickSort` este prezentată în continuare :

Subalgoritmul `QuickSort (n, K, St, Dr)` este:

```
Fie  $i := St$ ;  $j := Dr$ ;  $a := k_i$ ;  
Repetă  
  Cât timp  $k_j \geq a$  și  $(i < j)$  execută  $j := j - 1$  sf-cât  
   $k_i := k_j$ ;  
  Cât timp  $k_i \leq a$  și  $(i < j)$  execută  $i := i + 1$  sf-cât  
   $k_j := k_i$  ;  
până când  $i = j$  sf-rep  
Fie  $k_i := a$ ;  
Dacă  $St < i-1$  atunci Cheamă QuickSort(n, K, St, i - 1) sf-dacă  
Dacă  $i+1 < Dr$  atunci Cheamă QuickSort(n, K, i + 1, Dr) sf-dacă  
sf-QuickSort
```

Complexitatea algoritmului prezentat este $\Theta(n^2)$ în cel mai nefavorabil caz, dar complexitatea medie este de ordinul $\Theta(n \log_2 n)$.

2. Metoda "divide et impera"

Strategia "Divide et Impera" în programare presupune:

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

Metoda se aplica problemelor care pot fi descompuse în subprobleme independente, similar problemei inițiale, de dimensiuni mai mici și care pot fi rezolvabile foarte ușor.

Observații:

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

Formalizare

Sublalgortmul $\text{NumeAlg}(D)$ este:

Dacă $\dim(D) \leq a$ atunci

 @problema se rezolva

altfel

 @ Descompune D in d_1, d_2, \dots, d_k

 Cheama $\text{NumeAlg}(d_1)$

 Cheama $\text{NumeAlg}(d_2)$

 .

 .

 Cheama $\text{NumeAlg}(d_k)$

 @ construiește rezultatul final prin utilizarea rezultatelor
 partiale din apelurile de mai sus

sfdacă

sf-NumeAlg

3. Algoritmi și 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 specificație dată

Problema 1

Scrieți o funcție care satisface următoarea specificație:

Date nr ;

Precondiția: $nr \in \mathbb{N}, nr \geq 1$

Rezultate l_1, l_2, \dots, l_n ;

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

maximal

Problema 2

Scrieți o funcție care satisface următoarea specificație:

Date $n, L = (l_1, l_2, \dots, l_n)$;

Precondiția: $l_i \in \mathbb{R}, i = 1, n$

Rezultate $R = (r_1, r_2, \dots, r_n)$;

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

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), după 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:

- ce face acest program;
- ce realizează fiecare subprogram;
- ce rezultate sunt tipărite pentru 12 1233 1132 2338 8533 10000 21500 0 ?

```

#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

Precizați ce realizează următorul program, apoi scrieți programul C++ pentru *funcția* 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, scrieți programul care realizează *decodificarea*!

4. 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

4.1. Realizarea protecției datelor prin metoda programării modulare

Dezvoltarea programelor prin programare procedurală înseamnă folosirea unor funcții și proceduri pentru scrierea programelor. În limbajul C lor le corespund funcțiile care returnează o valoare sau nu. Însă în cazul aplicațiilor mai mari ar fi de dorit să putem realiza și o protecție corespunzătoare a datelor. Acest lucru ar însemna că numai o parte a funcțiilor să aibă acces la datele problemei, acelea care se referă la datele respective. Programarea modulară oferă o posibilitate de realizare a protecției datelor prin folosirea clasei de memorie static. Dacă într-un fișier se declară o dată aparținând clasei de memorie statică în afara funcțiilor, atunci ea poate fi folosită începând cu locul declarării până la sfârșitul modulului respectiv, dar nu și 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 funcții corespunzătoare pentru inițializarea vectorului, eliberarea zonei de memorie ocupate și ridicarea la pătrat, respectiv afișarea elementelor vectorului. O posibilitate de implementare a modulului este prezentată în fișierul **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 obținând un program obiect. Un exemplu de program principal este prezentat în fișierul **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 deși 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 și adunarea a doi vectori. În vederea înlăturării acestui neajuns s-au introdus tipurile abstracte de date.

4.2. Tipuri abstracte de date

Tipurile abstracte de date realizează o legătură mai strânsă între datele problemei și operațiile (funcțiile) 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 și declararea sau definirea funcțiilor 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();
};

```

Funcțiile declarate sau definite în interiorul structurii vor fi numite *funcții membru* iar datele *date membru*. Dacă o funcție membru este definită în interiorul structurii (ca și funcția *distr* din exemplul de mai sus), atunci ea se consideră funcție *inline*. Dacă o funcție membru se definește în afara structurii, atunci numele funcției se va înlocui cu numele tipului abstract urmat de operatorul de rezoluție (::) și numele funcției membru. Astfel funcțiile *init*, *lapatrat* și *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;
}

```

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

4.3. Declararea claselor

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

implicit fiind *private*. Menționăm că și în cazul structurilor putem să folosim modificatori de protecție, 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* și *d* au fost declarate ca date de tip *private* (protejate), iar funcțiile membru au fost declarate publice (neprotejate). Bineînțeles, o parte din datele membru pot fi declarate publice, și unele funcții membru pot fi declarate protejate, dacă natura problemei cere acest lucru. În general, datele membru protejate pot fi accesate numai de funcțiile membru ale clasei respective și eventual de alte funcții numite *funcții prietene* (sau funcții *friend*).

O altă observație importantă referitoare la exemplul de mai sus este că inițializarea datelor membru și eliberarea zonei de memorie ocupată s-a făcut prin funcții membru specifice.

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

```
nume_clasă listă_de_obiecte;
```

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

```
vector v;
```

Inițializarea obiectelor se face cu o funcție membru specifică numită *constructor*. În cazul distrugerii unui obiect se apelează automat o altă funcție 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 și 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 funcție inline. Definirea funcțiilor membru care sunt declarate, dar nu sunt definite în interiorul clasei se face ca și în cazul tipurilor abstracte de date de tip *struct*, folosind operatorul de rezoluție.

4.3.1. Membrii unei clase. Pointerul *this*

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

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

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

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

În interiorul funcțiilor membru însă referirea la datele respectiv funcțiile 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 funcție membru, și folosește acest pointer pentru identificarea datelor și funcțiilor 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 funcției membru *afiseaza* se utilizează de exemplu data membru `d`, atunci ea este interpretată de către compilator ca și `this->d`.

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

4.3.2. Constructorul

Inițializarea obiectelor se face cu o funcție membru specifică numită constructor. Numele constructorului trebuie să coincidă cu numele clasei. O clasă poate să aibă mai mulți constructori. În acest caz aceste funcții membru au numele comun, ceea ce se poate face datorită posibilității de supraîncărcare a funcțiilor. Bineînțeles, în acest caz numărul și/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 clasă cu mai mulți constructori, având ca date membru numele și prenumele unei persoane, și cu o funcție membru pentru afișarea numelui complet.

Fișierul `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();
};
```

Fișierul `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(ume) + 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;
}

```

Fișierul `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 copiere
    s = D.toString();
    cout << s << endl;
    delete[] s;
    return 0;
}

```

Observăm prezența a doi constructori specifici: *constructorul implicit* și *constructorul de copiere*. Dacă o clasă are constructor fără parametri atunci el se va numi *constructor implicit*. *Constructorul de copiere* se folosește la inițializarea obiectelor folosind un obiect de același tip (în exemplul de mai sus o persoană cu numele și 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 conțină 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 definiți de programator, sau obiectul care se inițializează cu un constructor implicit, sau cu toți parametrii implicați.

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

Fișierul *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* și *sotie* au fost declarați ca și referințe la tipul *persoana*. Dacă ar fi fost declarați ca parametri formali de tip *persoana*, atunci în cazul declarației:

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

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

4.3.3. Destructorul

Destructorul este funcția membru care se apelează în cazul distrugerii obiectului. Destructorul obiectelor globale se apelează automat la sfârșitul funcției *main* ca parte a funcției *exit*. Deci, nu este indicată folosirea funcției *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 și un mod explicit de apelare a

destructorului, în acest caz numele destructorului trebuie precedat de numele clasei și operatorul de rezoluție.

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

Fișierul `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;
}
```


5. Relații între clase

5.1. Bazele teoretice

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

În viața de zi cu zi însă ne întâlnim nu numai cu obiecte separate, dar și 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: moștenirea (inheritance). Acest lucru înseamnă că se moștenesc toate datele și funcțiile membru ale clasei de bază de către clasa derivată, dar se pot adăuga elemente noi (date membru și funcții membru) în clasa derivată. În cazul în care o clasă derivată are mai multe clase de bază se vorbește despre moștenire multiplă.

O altă proprietate importantă a obiectelor care aparțin clasei derivate este că funcțiile membru moștenite pot fi supraîncărcate. Acest lucru înseamnă că o operație referitoare la obiectele care aparțin ierarhiei are un singur identificator, dar funcțiile care descriu această operație pot fi diferite. Deci, numele funcției și lista parametrilor formali este aceeași în clasa de bază și în clasa derivată, dar descrierea funcțiilor diferă între ele. Astfel, în clasa derivată funcțiile membru pot fi specifice clasei respective, deși operația se identifică prin același nume. Această proprietate se numește polimorfism.

5.2. Declarația claselor derivate

O clasă derivată se declară în felul următor:

```
class nume_clasă_derivată : lista_claselor_de_bază {  
    //date membru noi și funcții membru noi  
};
```

unde lista_claselor_de_bază este de forma: elem₁, elem₂, ..., elem_n și 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* și *private* se numesc și de această dată modificatori de *protecție*. 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 protecție 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* și *public* devin de tip *protected*, respectiv *private* dacă modificatorul de protecție referitor la clasa de bază este *protected* respectiv *private*, și rămân neschimbate dacă modificatorul de protecție referitor la clasa de bază este *public*. Din acest motiv în general datele membru se declară de tip *protected* și modificatorul de protecție referitor la clasa de bază este *public*. Astfel datele membru pot fi accesate, dar rămân protejate și în clasa derivată.

5.3. Funcții virtuale

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

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

Fișierul `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 execuție se obține 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 funcției *main* s-a apelat funcția membru *functia_2* moștenită de la clasa de bază, dar funcția membru *functia_1* apelată de *functia_2* s-a determinat încă în faza de compilare. În consecință, deși funcția membru *functia_1* s-a supraîncărcat în clasa derivată nu s-a apelat funcția supraîncărcată ci funcția membru a clasei de bază.

Acest neajuns se poate înlătura cu ajutorul introducerii noțiunii de funcție membru *virtuală*. Dacă funcția membru este virtuală, atunci la orice apelare a ei, determinarea funcției membru corespunzătoare a ierarhiei de clase nu se va face la compilare ci la execuție, în funcție de natura obiectului pentru care s-a făcut apelarea. Această proprietate se numește *legare dinamică*, iar dacă determinarea funcției membru se face la compilare, atunci se vorbește de legare statică.

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

Acest lucru se poate realiza declarând *functia_1* ca funcție membru virtuală. Astfel, pentru orice apelare a funcției membru *functia_1*, determinarea aceluia exemplar al funcției membru din ierarhia de clase care se va executa, se va face la execuție și nu la compilare. Ca urmare, funcția membru *functia_1* se determină prin legare dinamică.

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

Dacă o funcție 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 obținut prin execuție 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 funcția membru *functia_1* a clasei derivate. Prezentăm în continuare un alt exemplu în care apare necesitatea introducerii funcțiilor membru virtuale. Să se definească clasa *fractie* referitoare la numerele raționale, având ca date membru numărătorul și numitorul fracției. Clasa trebuie să aibă un constructor, valoarea implicită pentru numărător fiind zero, iar pentru numitor unu, precum și două funcții membru: *produs* pentru a calcula produsul a două fracții și *inmulteste* pentru înmulțirea obiectului curent cu fracția dată ca și parametru. De asemenea, clasa *fractie* trebuie să aibă și o funcție membru pentru afișarea unui număr rațional. Folosind clasa *fractie* ca și clasă de bază se va defini clasa derivată *fractie_scrie*, pentru care se va supraîncărca funcția *produs*, astfel încât concomitent cu efectuarea înmulțirii să se afișeze pe *stdout*

operația respectivă. Funcția *inmulteste* nu se va supraîncărca, dar operația efectuată trebuie să se afișeze pe dispozitivul standard de ieșire și în acest caz.

Fișierul `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 execuție se obține:

```

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

```

Observăm că rezultatul nu este cel dorit, deoarece operația de înmulțire s-a afișat numai o singură dată, și anume pentru expresia `r1 = p1.produs(q1)`. În cazul expresiei `r2 = p1.inmulteste(q1)` însă nu s-a afișat operația de înmulțire. Acest lucru se datorează faptului că funcția membru `inmulteste` nu s-a supraîncărcat în clasa derivată. Deci s-a apelat funcția moștenită de la clasa `fractie`. În interiorul funcției `inmulteste` s-a apelat funcția membru `produs`, dar deoarece această funcție membru s-a determinat încă în faza de compilare, rezultă că s-a apelat funcția referitoare la clasa `fractie` și nu cea referitoare la clasa derivată `fractie_scrie`. Deci, afișarea operației s-a efectuat numai o singură dată. Soluția este, ca și în exemplul anterior, declararea unei funcții membru virtuale, și anume funcția `produs` se va declara ca funcție virtuală. Deci 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 obținem:

```

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 afișarea operației s-a făcut de două ori, pentru ambele expresii. Funcțiile virtuale, ca și alte funcții membru de fapt, nu trebuie neapărat supraîncărcate în clasele derivate. Dacă nu sunt supraîncărcate atunci se moștenește funcția membru de la un nivel superior.

Determinarea funcțiilor membru virtuale corespunzătoare se face pe baza unor tabele construite și gestionate în mod automat. Obiectele claselor care au funcții membru virtuale conțin și un pointer către tabela construită. De aceea gestionarea funcțiilor membru virtuale necesită mai multă memorie și un timp de execuție mai îndelungat.

5.4. Clase abstracte

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

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

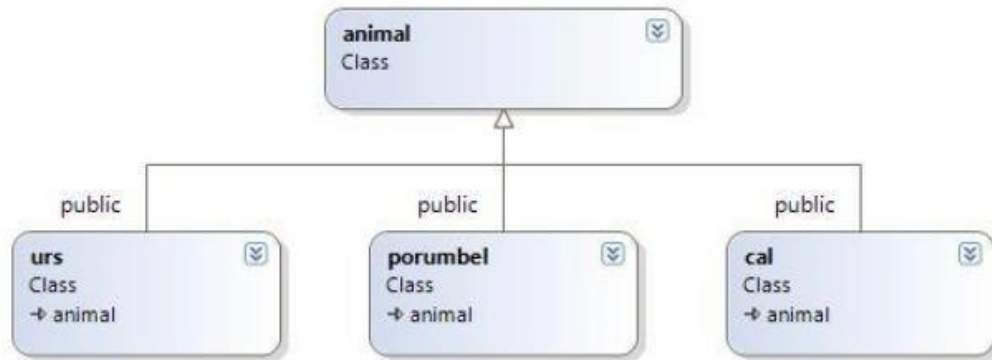


Figura 1. Ierarhie de clase referitoare la animale

Funcția virtuală pură este o funcție membru care este declarată, dar nu este definită în clasa respectivă. Ea *trebuie* definită într-o clasă derivată. Funcția membru virtuală pură se declară în modul următor. Antetul obișnuit al funcției este precedat de cuvântul cheie *virtual*, și antetul se termină cu $= 0$. După cum arată numele și declarația ei, funcția membru virtuală pură este o funcție virtuală, deci selectarea exemplarului funcției din ierarhia de clase se va face în timpul execuției programului.

Clasele care conțin cel puțin o funcție membru virtuală pură se vor numi *clase abstracte*. Deoarece clasele abstracte conțin funcții membru care nu sunt definite, nu se pot crea obiecte aparținând claselor abstracte. Dacă funcția virtuală pură nu s-a definit în clasa derivată atunci și clasa derivată va fi clasă abstractă și ca atare nu se pot defini obiecte aparținând acelei clase. Să considerăm exemplul de mai sus și 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. Afișarea acestui rezultat se va face de către o funcție membru a clasei *animal* care nu se supraîncarcă în clasele derivate.

Fișierul `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 deși clasa *animal* este clasă abstractă, este utilă introducerea ei, pentru că multe funcții membru pot fi definite în clasa de bază și moștenite fără modificări în cele trei clase derivate.

5.5. Interfețe

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

În continuare introducem o clasă abstractă *Vehicul*, care nu conține numai funcții membru virtuale pure, și două clase derivate din această clasă abstractă. Fișierul `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*.

6. Probleme propuse

1. Scrieți un program într-unul din limbajele de programare Python, C++, Java, C# care:
 - a. Definiște 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. Definiște 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. Definiște 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. Definiște 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.

7. Bibliografie generală

1. Alexandrescu, *Programarea modernă în C++*. Programare generică și modele de proiectare aplicative, 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.