Tehnici de programare

fișiere text; scriere; citire

Privit în mod abstract, un fișier este un șir de octeți care are asociat un nume. Șirul de octeți poate stoca orice fel de date, spre exemplu muzică, filme, documente, cod C, etc.

Din punct de vedere al conținutului, fișierele se împart în două mari categorii:

- Fișierele care conțin informații sub formă de text în diferite formate, care pot fi vizualizate și modificate cu orice editor text și pot fi citite și înțelese direct de către om se numesc **fișiere text**. Spre exemplu codul sursă în limbajul C este păstrat în fișiere text. Formatele (.txt, .xml, .html, .json, .csv) sunt exemple de fișiere text.
- Fișierele al căror conținut nu poate fi înțeles direct de om și necesită programe specifice pentru a fi interpretat se numesc **fișiere binare**. Exemple de fișiere binare sunt fișierele audio (.mp3, .wav), fișierele video (.avi, .mpeg), fișierele document Word (.doc), etc. În principiu este posibil ca prin practică și după multă experiență acumulată cineva să poată înțelege direct și conținutul anumitor fișiere binare, uitându-se doar la valorile numerice din șirul de octeți, dar acestea sunt cazuri speciale și rare.

Fișierele sunt stocate de regulă pe disc (hard-disk, stick USB, CD, DVD, etc.). Fișierele sunt organizate ierarhic, în așa numite directoare (sau foldere). Spre exemplu în directorul */bin* se găsesc mai multe fișiere binare care sunt de fapt programe executabile din Linux (fișierele *cat*, *less*, etc., care corespund comenzilor discutate în laboratorul 1). În directorul */var/log* se găsesc fișiere text care sunt jurnale de mesaje generate de programe care rulează în Linux (spre exemplu fișierul */var/log/messages* care poate fi vizualizat cu orice editor text).

Calea unui fișier este concatenarea numelor tuturor directoarelor în care se află fișierul împreună cu numele fișierului, separate de așa numitul **separator de cale**. Separatorul de cale este un caracter care diferă de la un sistem de operare la altul. În Linux separatorul de cale este */*. Spre exemplu calea */var/log/messages* se traduce prin: fișierul *messages* care se află în directorul *log* care la rândul lui se află în directorul *var* care la rândul lui se află în directorul rădăcină. În Windows separatorul de cale este \. Spre exemplu calea *"C:\Documents and Settings\student\p3.c"* se traduce prin: fișierul *p3.c* care se află în directorul *student* care la rândul lui se află în directorul *"Documents and Settings*" care la rândul lui se află în directorul rădăcină al partiției C: În general, în C se poate folosi ca separator de cale */* și pentru programele care vor rula în Windows.

Fișierele text pot fi vizualizate din terminal rulând comanda *"less <nume fișier>".* Conținutul fișierului este afișat pe ecran și se poate naviga sus/jos folosind tastele cu săgeți. Pentru a ieși din program se apasă tasta Q.

Fișierele binare se vizualizează cu programe diferite, în funcție de tipul fișierului. De exemplu, fișierele video pot fi redate cu VLC player, cele audio cu Audacious, etc. Dacă se dorește vizualizarea octeților din fișier, se poate folosi un editor hexa. În Linux se poate folosi *"hexedit <nume fișier>".* Editorul afișează în baza 16 conținutul fișierului al cărui nume este transmis ca parametru. Pentru a ieși din program se tastează combinația CTRL+C. *hexedit* poate fi folosit și pentru a vizualiza valorile octeților din fișierele text. Mai multe detalii despre fișierele binare vom discuta în laboratorul următor. În acest laborator vom opera cu fișierele text.

Deschiderea și închiderea unui fișier

Pentru a se putea lucra cu fișiere, acestea trebuie mai întâi deschise. Aceasta se face cu funcția:

FILE ***fopen**(const char *nume_fisier, const char *mod);

Funcția *fopen* (file open) este declarată în *stdio.h* (ca de altfel marea majoritate a funcțiilor de intrare/ieșire - standard input/output) și are două argumente, fiecare de tip șir de caractere:

- *nume_fisier* - numele fișierului care va fi deschis. Acest nume se poate da fie absolut, fie relativ. Dacă este relativ, atunci se consideră ca fiind relativ la directorul curent. Când se folosește un IDE gen *Code::Blocks*, directorul curent este directorul unde s-a generat executabilul corespunzător codului sursă. Exemple:
  - dacă într-un IDE compilăm programul *prg1.c* și rezultă executabilul *prg1*, directorul curent este cel în care se află *prg1*
  - dacă suntem în terminal și din linie de comandă apelăm *prg1*, directorul curent este cel curent din terminal
- *mod* - modul în care va fi deschis fișierul. Este un șir din una sau mai multe litere, având semnificațiile de mai jos:
  - **r** - (read) fișierul este deschis pentru citire (se va permite doar citirea datelor din el), începând cu începutul fișierului.
  - **r+** - fișierul este deschis simultan pentru citire și scriere, începând cu începutul fișierului. Dacă se vor scrie date, conținutul originar va fi suprascris.
  - **w** - (write) fișierul este deschis pentru scriere. Dacă nu există, se va crea un fișier nou. Dacă fișierul există deja, atunci i se șterge tot conținutul anterior.
  - **a** - (append) fișierul este deschis pentru adăugare (scriere la sfârșit)
  - **a+** - fișierul este deschis pentru citire și adăugare. Scrierea întotdeauna se face la sfârșit de fișier. În standardul de C nu se specifică de unde anume se face citirea.

  După una dintre opțiunile de mai sus, se mai poate adăuga una dintre literele: **b** - binar - nu se face niciun fel de procesare asupra caracterelor citite/scrise: **t** (implicit) - text - se face translatarea caracterului **\n** (newline) în/din convenția folosită pentru sistemul de operare respectiv pentru fișiere text (ex: pentru Windows se traduce *\n* în *\r\n* la scriere și invers la citire).

  În general fișierele text se deschid sau pentru citire, sau pentru scriere. Deschiderea simultană atât pentru citire cât și pentru scriere se folosește mai mult la fișiere binare.

Dacă *fopen* reușește să deschidă fișierul, ea va returna un pointer către un descriptor (handler) de fișier. Acest pointer are tipul **FILE*** și prin intermediul lui se vor realiza toate operațiile cu fișierul. Dacă *fopen* nu reușește să deschidă fișierul, va returna NULL.

Atenție: întotdeauna se va testa dacă deschiderea unui fișier a avut loc cu succes.

După ce s-au terminat operațiile cu un fișier, acesta trebuie închis. Închiderea se face cu funcția:

int **fclose**(FILE *fis)

Dacă fis este NULL sau este un fișier deja închis, *fclose* nu are niciun efect.

Lucrul cu fișiere text este destul de simplu în C, deoarece se pot folosi la fel ca și până acum funcțiile *printf/scanf*, doar că le mai adăugăm un **f** în fața numelui (f - file) și pe prima poziție în lista de argumente vom pune descriptorul de fișier. Vom avea astfel funcțiile **fprintf** / **fscanf**. La acestea se adaugă **fgets** / **fputs**, care deja operează pe fișiere.

La orice operație cu fișiere pot să apară erori. De exemplu, când scriem într-un fișier, s-ar putea să nu mai fie spațiu suficient pe disc. Când citim dintr-un fișier, iar acesta este pe un stick USB, stickul poate fi scos din

calculator înainte de a se fi terminat citirea. O aplicație trebuie să testeze acest gen de erori și să le raporteze utilizatorului (ex: *fprintf* returnează un număr negativ în caz de eroare). Pentru simplificarea codului, la laborator nu vom testa dacă operațiile de scriere/citire din fișiere au avut loc cu succes, ci doar vom verifica deschiderea fișierului.

Scrierea într-un fișier

Pentru scriere vom deschide fișierul în mod de scriere (**w** - write), sau de adăugare (**a** - append) iar apoi vom folosi funcții cum sunt **fprintf**, **fputs**, **fputc**.

Exemplu: să se scrie un program care citește de la tastatură un *n* întreg și scrie într-un fișier cu numele ”out.txt” pentru fiecare număr din intervalul [0,n] dacă este par sau impar:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   FILE *fis;
   int i,n;
   printf("n=");scanf("%d",&n);
   if((fis=fopen("out.txt","w"))==NULL){                  // deschidere fisier pentru scriere (w - write)
      printf("eroare deschidere fisier\n");
      exit(EXIT_FAILURE);
      }
   for(i=0;i<=n;i++){
      if(i%2==0){
         fprintf(fis,"%d este par\n",i);                  // fprintf(fis,...) - file printf
         }else{
         fprintf(fis,"%d este impar\n",i);
         }
      }
   fclose(fis);                                           // închidere fisier
   return 0;
}
```

În programul de mai sus, după ce s-a citit *n* de la tastatură, s-a deschis fișierul și s-a testat dacă s-a putut deschide. Dacă nu s-a putut deschide, se afișează o eroare și se iese din program cu un cod de eroare.

Se poate constata că scrierea în fișier s-a făcut practic la fel ca și când s-ar fi scris pe ecran, doar că în loc de ”*printf*” s-a folosit ”*fprintf(fis,...”.* După ce s-au scris toate informațiile în fișier, acesta s-a închis cu *fclose*.

După execuția programului se va verifica dacă pe disc a apărut într-adevăr fișierul *out.txt* cu conținutul dorit. Acest fișier trebuie să apară în același director în care este și executabilul codului scris de noi. Se va mai rula încă o dată programul, cu alt *n*, pentru a se verifica faptul că opțiunea *”w”* șterge tot conținutul anterior.

O metodă simplă pentru a se dezvolta programe care scriu în fișiere text este următoarea: se scrie programul dar, în loc să se inițializeze fișierul *fis* folosind *fopen*, se va inițializa direct cu *stdout* (ieșirea standard, pe ecran). Când se rulează programul, tot ceea ce ar fi trebuit scris în fișier va fi scris pe ecran. După ce s-a testat corectitudinea programului, se face inițializarea lui *fis* folosind *fopen*. Acum nu se va mai scrie pe ecran, ci în fișierul deschis cu *fopen*.

Aplicația 4.1: Se citesc de la tastatură maxim 100 numere reale, până la întâlnirea numărului 0. Să se sorteze aceste numere și să se scrie într-un fișier, toate numerele fiind pe o singură linie, separate prin **|** (bară verticală).

Citirea din fișier

Dacă se specifică în avans câte date vom avea de citit (de exemplu citind prima linie din fișier), vom citi datele din fișier folosind o buclă *for*. Dacă nu știm de la început câte date sunt și dorim să citim tot fișierul, va trebui să avem o condiție de testare a sfârșitului de fișier sau a faptului că nu se mai pot citi date.

Dacă folosim pentru citire funcția *fscanf*, valoarea returnată de ea (de tip *int*) are următoarea semnificație: dacă a apărut o eroare de citire din fișier, se returnează EOF (End Of File - o constantă predefinită în C). În caz de succes, *fscanf* returnează numărul de valori citite. Dacă acest număr este 0, înseamnă că s-a ajuns la sfârșit de fișier.

Funcția *scanf* nu ține cont de spații, linii noi sau TAB-uri, astfel încât ea încearcă să citească valorile specificate chiar dacă ele sunt pe aceeași linie, linii diferite sau separate prin mai multe linii vide.

Exemplu: un fișier conține pe fiecare linie coordonatele unor puncte în plan (x,y), definite prin numere reale. Să se citească aceste puncte într-un vector alocat dinamic, astfel încât să ocupe memoria minimă necesară, iar apoi să se afișeze perechea de puncte care sunt cele mai depărtate unele de altele:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

typedef struct{
   double x,y;
   }Pt;

FILE *fis;        // declarat global pentru a putea fi inchis din adauga() in caz de eroare
Pt *puncte;       // initializat implicit cu NULL
int nPuncte;

void adauga(double x,double y)    // adauga un punct in puncte
{
   Pt *p=(Pt*)realloc(puncte,(nPuncte+1)*sizeof(Pt));
   if(!p){
      printf("memorie insuficienta");
      free(puncte);
      fclose(fis);
      exit(EXIT_FAILURE);
      }
   puncte=p;
   puncte[nPuncte].x=x;
   puncte[nPuncte].y=y;
   nPuncte++;
}

// transmitere de structuri prin valoare
double dist(Pt p1,Pt p2)
{
   double dx=p1.x-p2.x, dy=p1.y-p2.y;
```

```
    return sqrt(dx*dx+dy*dy);
}

int main(void)
{
    int i,j;
    double x,y;
    if((fis=fopen("pt.txt","r"))==NULL){
        printf("eroare deschidere fisier\n");
        exit(EXIT_FAILURE);
        }
    while(fscanf(fis,"%lg%lg",&x,&y)==2){       // atata timp cat se citesc puncte
        adauga(x,y);                            // adauga punctul citit in vector
        }
    fclose(fis);                    // nu mai este nevoie de fisier
    if(!nPuncte){
        printf("fisier vid\n");
        exit(EXIT_FAILURE);
        }
    double dMax=0;                  // presupunem distanta maxima ca fiind 0
    int iMax=0,jMax=0;              // indecsii punctelor celor mai departate intre ele
    for(i=0;i<nPuncte;i++){
        for(j=i+1;j<nPuncte;j++){
            double d=dist(puncte[i],puncte[j]);
            if(d>dMax){             // daca s-a gasit o pereche de puncte mai distantate, actualizeaza rezultatul
                iMax=i;
                jMax=j;
                dMax=d;
                }
            }
        }
    printf("punctele cele mai departate: (%g,%g) - (%g,%g)\n",
            puncte[iMax].x, puncte[iMax].y, puncte[jMax].x, puncte[jMax].y);
    free(puncte);
    return 0;
}
```

Se poate constata că de fapt partea de citire din fișier este destul de simplă. A fost nevoie de mai mult cod pentru adăugarea unui punct într-un vector redimensionat dinamic și pentru stabilirea punctelor cele mai îndepărtate.

Reamintim faptul că *fscanf* (la fel ca *scanf*), dacă citește șiruri de caractere (%s), atunci se va opri la primul spațiu sau terminator de linie. Din acest motiv, *fscanf* poate fi folosit pentru a se citi din fișier cuvânt cu cuvânt, dar este puțin mai complex să citim de exemplu nume formate din mai multe cuvinte.

Dacă avem de citit din fișier linie cu linie, se poate folosi funcția *fgets*. Dacă *fgets* reușește să citească o linie, va returna un pointer la bufferul în care s-a făcut citirea. Dacă s-a ajuns la sfârșit de fișier sau dacă a intervenit o eroare de citire, *fgets* va returna NULL.

Exemplu: Să se citească dintr-un fișier toate liniile de text, să se elimine liniile vide, să se sorteze alfabetic și se se scrie liniile rezultate într-un fișier de ieșire. O linie poate avea maxim 1000 de caractere. Liniile citite vor fi memorate într-un vector alocat dinamic:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char **linii;      // vector de linii: char *linii[]
int nLinii;
FILE *fin;         // fisierul de intrare
FILE *fout;        // fisierul de iesire

void eliberare()
{
   int i;
   for(i=0;i<nLinii;i++)free(linii[i]);
   free(linii);
   fclose(fin);
   fclose(fout);
}

void adauga(const char *linie)    // adauga o linie in linii
{
   // duplica linia
   char *linieNoua=(char*)malloc((strlen(linie)+1)*sizeof(char));
   if(!linieNoua){
      printf("memorie insuficienta");
      eliberare();
      exit(EXIT_FAILURE);
      }
   strcpy(linieNoua,linie);
   // face loc in linii pentru linia noua
   char **p=(char**)realloc(linii,(nLinii+1)*sizeof(char*));
   if(!p){
      printf("memorie insuficienta");
      free(linieNoua);
      eliberare();
      exit(EXIT_FAILURE);
      }
   linii=p;
   linii[nLinii]=linieNoua;
   nLinii++;
}

// functia de comparare pentru qsort
// deoarece qsort transmite elementele vectorului prin adresele lor
// si fiecare element din linii este un sir de caractere (char*)
// inseamna ca de fapt se va transmite char** (adresa unui sir de caractere)
int cmp(const void *p1,const void *p2)
{
   // se transmit lui strcmp valorile pointate de sirurile de caractere transmise prin adresa
   // strcmp are aceeasi conventie pentru valorile returnate ca si qsort
   // deci valoarea returnata de strcmp se poate returna direct lui qsort
   return strcmp(*(char **)p1,*(char**)p2);
```

```
}

int main(void)
{
    char linie[1000];
    int i;
    // citeste liniile din fisierul de intrare
    if((fin=fopen("linii.txt","rt"))==NULL){
        printf("eroare deschidere fisier de intrare\n");
        exit(EXIT_FAILURE);
        }
    while(fgets(linie,1000,fin)!=NULL){    // atata timp cat se citeste o linie de intrare
        // daca linia este vida, nu o adauga in vectorul de linii
        // linia e vida daca s-a ajuns la sfarsit de fisier sau daca este doar un \n
        if(strlen(linie)<=1)continue;
        adauga(linie);
        }
    // nu mai este nevoie de fisierul de intrare, deci se putea inchide aici. va fi inchis la sfarsit, de eliberare()

    qsort(linii,nLinii,sizeof(char*),cmp);

    // scrie liniile sortate in fisierul de iesire
    if((fout=fopen("sortate.txt","wt"))==NULL){
        printf("eroare deschidere fisier de iesire\n");
        exit(EXIT_FAILURE);
        }
    for(i=0;i<nLinii;i++)fputs(linii[i],fout);
    fclose(fout);
    eliberare();
    return 0;
}
```

Deoarece SO diferite (ex: Windows și Linux) folosesc secvențe diferite de caractere pentru a reprezenta în fișiere sfârșitul de linie, s-a folosit **"t"** în *fopen*, pentru a se transforma aceste secvențe specifice într-un singur caracter **\n** (la citire) sau dintr-un singur caracter **\n** într-o secvență specifică (la scriere).

Dacă o linie de fișier conține date mai complexe, pentru care nu este suficient spațiul ca separator, ci trebuie să folosim alte separatore (ex: virgulă), atunci avem două posibilități:

- citim o linie întreagă cu *fgets* și apoi o despărțim în componente folosind funcții gen *strtok*
- citim caracter cu caracter folosind *fgetc(fis)* și, în funcție de caracterul citit, împărțim intrarea pe componente

Exemplu: Un fișier conține pe fiecare linie următoarele componente, separate prin virgulă: numele întreg al unui student (mai multe cuvinte) și una sau mai multe note ale sale, date ca numere reale. Se cere să se citească toți studenții din fișier și să se afișeze pentru fiecare media sa:

```
#include <stdio.h>
#include <stdlib.h>

// imparte buf in componente delimitate de sep(aparator)
// insereaza in buf '\0' in locul fiecarui separator, pentru a crea siruri distincte
```

```c
// in vectorul componente se vor depune pointeri la fiecare sir de caractere format in buf
// returneaza numarul de componente
int split(char **componente,char *buf,char sep)
{
    int start,i,n=0;        // start - pozitia de inceput a componentei curente; n - numarul de componente gasite
    for(start=i=0;buf[i];i++){
        if(buf[i]==sep){
            componente[n]=buf+start;
            buf[i]='\0';
            n++;
            start=i+1;
            }
        }
    if(start!=i){           // componenta de dupa ultimul separator, sau singura componenta daca nu s-a gasit sep
        componente[n]=buf+start;
        n++;
        }
    return n;
}

int main(void)
{
    FILE *fis;
    if((fis=fopen("studenti.txt","r"))==NULL){
        printf("eroare deschidere fisier\n");
        exit(EXIT_FAILURE);
        }
    char linie[1000];
    // numele va fi la pozitia 0, iar pe urmatoarele pozitii sunt note
    char *componente[100];
    int i,n;
    while(fgets(linie,1000,fis)!=NULL){
        n=split(componente,linie,',');
        if(n<=1)continue;       // sare peste liniile vide
        double suma=0;
        for(i=1;i<n;i++){
            suma+=atof(componente[i]);
            }
        printf("%s: %g\n",componente[0],suma/(n-1));
        }
    fclose(fis);
    return 0;
}
```

Programul de mai sus citește linie cu linie din fișier. Funcția *split* primește o linie și o împarte în componentele sale, după separatorul dat. Împărțirea în componente se face punând terminatoare de șir (‘\0’) în locul fiecărui separator (la sfârșit de șir există deja terminator). Pentru a se ști unde anume încep subșirurile nou formate în șirul inițial, se folosește variabila *componente*, care este un vector de șiruri de caractere, pentru a se stoca începuturile lor.

La citirea din fișier unele funcții (ex: *fgets*) returnează NULL atât în cazul în care s-a ajuns la sfârșit de fișier (EOF), cât și în caz de eroare. Dacă se dorește să se diferențieze între cele două situații, se poate folosi funcția *feof(fis)*. Aceasta returnează true (diferit de 0), doar dacă s-a ajuns la sfârșit de fișier. Funcția *feof* se poate folosi și la citirea din fișier, pentru a se testa dacă s-a ajuns la sfârșit (ex: *while(!feof(fis)){...citire..}*), dar în general este mai simplu să se folosească pentru aceasta direct valoarea returnată de funcțiile de citire.

Fișierele speciale stdin, stdout, stderr

Variabilele *stdin*, *stdout* și *stderr* sunt de fapt tot fișiere, astfel încât se pot folosi ca orice fișiere. Prin intermediul lor avem acces la tastatură (*stdin*) și la ecran (*stdout*), ca și când acestea ar fi fișiere de intrare, respectiv ieșire.

Fișierul *stderr* este inițial asociat tot ecranului, la fel ca *stdout*, dar se folosește pentru afișarea mesajelor de eroare. Dacă aplicația noastră va fi folosită de alte programe, acestea vor putea redirecționa *stdout* și *stderr* către destinații diferite, astfel încât să poată diferenția între mesajele normale ale aplicației și cele de eroare. Din acest motiv, este bine ca afișările mesajelor de eroare (ex: memorie insuficientă) să se facă folosind *stderr*:

```
fprintf(stderr,"memorie insuficienta");
```

Dacă se folosește *stdin* (tastatura) ca fișier de intrare și se dorește citirea datelor până la sfârșit de fișier (EOF), se poate introduce de la tastatură caracterul *sfârșit de fișier* (EOF) astfel: în Linux se apasă Ctrl-D, iar în Windows Ctrl-Z.

Aplicații propuse

Aplicația 4.2: Se citesc *m* și *n* de la tastatură, iar apoi o matrice *a[m][n]*. Matricea va fi alocată dinamic. Să scrie într-un fișier atât matricea originală cât și transpusa ei, separate printr-o linie goală.

Aplicația 4.3: Să se scrie un program care primește în linia de comandă un nume de fișier, urmat de unul sau mai multe cuvinte. Programul va afișa liniile din fișier care conțin simultan toate cuvintele din linia de comandă, în orice ordine.
Exemplu: ./cautare 1.txt ana ion - va afișa toate liniile din fișierul 1.txt care conțin simultan "ana" și "ion"

Aplicația 4.4: Citind caracter cu caracter dintr-un fișier, să se contorizeze de câte ori apare fiecare literă din alfabet și să se afișeze. Nu se va face distincție între literele mari și mici.

Aplicația 4.5: Să se contorizeze de câte ori apare într-un fișier fiecare cuvânt și să se afișeze. Nu se va face distincție între literele mari și mici.

Aplicația 4.6: Un fișier conține pe fiecare linie un produs dat prin numele său (un cuvânt) și preț (număr real), separate prin spațiu. Să se încarce fișierul în memorie și să se implementeze într-o buclă infinită un meniu cu următoarele opțiuni:

1. Caută un produs după nume și, dacă e găsit, îi afișează prețul
2. Afișează toate produsele
3. Adaugă un nou produs
4. Șterge un produs, după numele său
5. Salvează în fișierul originar baza de date din memorie
6. Ieșire din program

Aplicația 4.7: Să se modifice exemplul de sortare al liniilor din fișier, astfel încât o linie să poată avea orice lungime.

Aplicația 4.8: Un fișier conține pe fiecare linie numele unui angajat (posibil mai multe cuvinte) și salariul său (număr real), separate prin virgulă. Să se scrie o aplicație care primește în linia de comandă un fișier sursă, unul

destinație, un coeficient și două numere reprezentând un interval închis. Aplicația înmulțește toate salariile angajaților din intervalul dat cu coeficientul respectiv. Noua situație va fi scrisă în fișierul destinație.
Exemplu de apel: ./actualizare angajati.txt nou.txt 1.25 0 1500 - se vor citi toți angajații din *angajati.txt*, iar la cei care au salariile în intervalul [0,1500] se vor înmulți acestea cu 1.25. Noua situație va fi scrisă în *nou.txt*.