

Fișiere

15 noiembrie 2005

Lucrul cu fișiere

La nivel de *utilizator*, ne referim la un fișier prin *nume*.

La nivelul *interfeței de programare*, bibliotecile limbajului C definesc un tip **FILE** cu elementele necesare accesului la fișier (poziția curentă în fișier, tamponul de date, indicatori de eroare și EOF) Structura internă a tipului FILE: invizibilă programatorului; folosim doar pointeri **FILE ***, prin intermediul funcțiilor de bibliotecă.

Din punct de vedere logic, un fișier e un flux (*stream*) de octeți

Fișiere (FILE *) standard predefinite (și deschise automat la rulare)

stdin: fișierul standard de intrare (implicit: tastatura)

stdout: fișierul standard de ieșire (implicit: ecranul)

stderr: fișierul standard de eroare (implicit: ecranul)

Obs: E bine ca mesajele de eroare să fie scrise la **stderr**, pentru a putea fi separate (prin redirectare) de mesajele normale de ieșire

Orice operație cu fișiere poate rezulta în eroare (din multe motive)

⇒ e *obligatorie* testarea valorii returnate (codului de eroare)

Deschiderea și închiderea fișierelor

`FILE *fopen (const char *path, const char *mode);`

– arg. 1: numele fișierului (absolut sau față de directorul curent)

– arg. 2: modul de deschidere; primul caracter semnifică:

r: deschidere pentru citire (fișierul trebuie să existe)

w, **a**: deschidere pt. scriere; dacă fișierul nu există, e creat;

dacă există, **w** trunchiază la zero; **a** (append) adaugă la sfârșit

În plus, șirul de caractere pt. modul de deschidere mai poate conține:

+ permite și celălalt mod (**r/w**) în plus față de cel din primul caracter

b deschide fișierul în mod binar (implicit: în mod text)

– returnează `NULL` în caz de eroare (trebuie testat !!!)

– altfel, valoarea returnată se folosește pt. lucrul în continuare

`int fclose(FILE *stream);`

– scrie orice a rămas în tampoanele de date, închide fișierul

– returnează `0` în caz de succes, `EOF` în caz de eroare

Deschiderea, închiderea și lucrul cu fișierele

Secvența tipică de lucru cu un fișier (ex. pt. citire)

```
FILE *fp; char *name = "f.txt"; /* sau din argv[], sau solicitat */
if (!(fp = fopen(name, "r"))) { /* tratează eroarea */ }
else { /* lucrează cu fișierul */ }
if (fclose(fp)) /* eroare la închidere */;
```

La intrarea-ieșirea în mod *text* se pot petrece diverse *conversii* în funcție de implementare (de exemplu traducere `\n` în `\r\n` pt. DOS)
Datele citite corespund celor scrise doar dacă: toate caracterele sunt tipăribile, `\t` sau `\n`; `\n` nu e precedat de spații; ultimul caracter e `\n`
⇒ pentru *orice alte situații*, deschideți fișierele în mod *binar* (asigură corespondența exactă între conținutul scris și citit)

Citirea și scrierea într-un fișier folosesc același indicator de poziție ⇒ Pentru un fișier deschis în mod dual (cu +), nu se va citi direct după scriere fără a goli tampoanele (`fflush`) sau a re poziționa indicatorul; nu se scrie direct după citire fără re poziționarea indicatorului sau EOF

Citire/scriere (d)in fișiere

Cu funcții echivalente celor folosite până acum:

```
int fputc(int c, FILE *stream); /* scrie caracter în fișier */
int fgetc(FILE *stream);      /* citește caracter din fișier */
/* getc, putc: la fel ca și fgetc, fputc, dar sunt macrouri */
int ungetc(int c, FILE *stream); /* pune caracterul c înapoi */
```

```
int fscanf (FILE *stream, const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
```

```
int fputs(const char *s, FILE *stream); /* scrie un șir */
int puts(const char *s); /* scrie șirul și apoi \n la ieșire */
char *fgets(char *s, int size, FILE *stream);
```

– citește până la (inclusiv) linie nouă, sau max. size - 1 caractere, adaugă '\0' la sfârșit ⇒ citirea sigură a unei linii, fără depășire

NU FOLOSIȚI niciodată funcția gets(), nu e protejată la depășire!

Exemplu: afişarea unor fişiere

```
#include <stdio.h>
void cat(FILE *fi)
{ int c; while ((c = fgetc(fi)) != EOF) putchar(c); }

int main(int argc, char *argv[]) {
    FILE *fp;
    if (argc == 1) cat(stdin); /* citeşte de la intrare */
    else while (--argc > 0) { /* pt. fiecare argument */
        if (!(fp = fopen(*++argv, "r")))
            fprintf(stderr, "can't open %s", *argv);
        else { cat(fp); fclose(fp); }
    }
    return 0;
}
```

Funcții de eroare

```
void clearerr(FILE *stream);
```

resetează indicatorii de sfârșit de fișier și eroare pentru fișierul dat

```
int feof(FILE *stream); /* != 0: ajuns la sfârșit de fișier */
```

```
int ferror(FILE *stream); /* != 0 la eroare pt. acel fișier */
```

Coduri de eroare

Dacă un apel de sistem a rezultat în eroare, se poate citi codul erorii din variabila globală extern `int errno`; declarată în `errno.h`

Se poate folosi împreună cu funcția `char *strerror(int errnum)`; din `string.h` care returnează un șir de caractere cu descrierea erorii

Se poate folosi direct funcția `void perror(const char *s); /*stdio.h*/` care tipărește mesajul `s` dat de utilizator, un `:` și apoi descrierea erorii

Citire și scrierea directă

Până acum: funcții orientate pe caractere, linii, formatare (fișiere text)

Pentru a citi/scrie direct un număr dat de octeți, neinterpretați:

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE *stream);
/* citesc/scriu nmemb obiecte de câte size octeți */
```

Funcțiile întorc numărul obiectelor *complete* citite/scrise corect.

Dacă e mai mic decât cel dat, cauza se află din feof și ferrror

În C, nu există fișiere tipizate (file of din PASCAL);

putem însă defini astfel de funcții pentru fiecare tip în parte:

```
size_t readint(int *pn, FILE *stream) /* în format binar */
{ return fread(pn, sizeof(int), 1, stream); }
size_t writedbl(double x, FILE *stream) /* în format binar */
{ return fwrite(&x, sizeof(double), 1, stream); }
```


Exemplu: copierea a două fișiere

```
#include <errno.h>
#include <stdio.h>
#define MAX 512
int filecopy(FILE *fi, FILE *fo) {
    char buf[MAX];
    int size; /* nr. octeți citiți */
    while (!feof(fi)) {
        size = fread(buf, 1, MAX, fi);
        fwrite(buf, 1, size, fo); /* scrie doar size */
        if (ferror(fi) || ferror(fo)) return errno;
    }
    return 0;
}
```

Exemplu: copierea a două fișiere (cont.)

```
int main(int argc, char *argv[]) {
    FILE *fi, *fo;
    if (argc != 3) {
        fprintf(stderr, "usage: copy source destination\n"); return -1;
    } else {
        if (!(fi = fopen(argv[1], "r"))) {
            fprintf(stderr, "%s: can't open %s: ", argv[0], argv[1]);
            perror(NULL); /* am scris deja mesajul */; return errno;
        }
        if (!(fo = fopen(argv[2], "w"))) {
            fprintf(stderr, "%s: can't open %s: ", argv[0], argv[2]);
            perror(NULL); return errno;
        }
        if (filecopy(fi, fo)) perror("Eroare la copiere");
        if (fclose(fi) | fclose(fo)) perror("Eroare la închidere");
    }
    return errno;
}
```

Funcții de poziționare, etc.

Pe lângă citire/scriere secvențială, e posibilă poziționarea în fișier:

```
long ftell(FILE *stream); /* pozitia de la începutul fișierului */
int fseek(FILE *stream, long offset, int whence); /* poziționare */
Al treilea parametru: punctul de referință pt. poziționarea cu offset:
SEEK_SET (început), SEEK_CUR (punctul curent), SEEK_END (sfârșit)

void rewind(FILE *stream); /* repoziționează indicatorul la început */
(echivalent cu (void)fseek(stream, 0L, SEEK_SET), plus clearerr
```

Repoziționarea trebuie efectuată:

- când dorim sa “sărim” peste o anumită porțiune din fișier
- când fișierul a fost scris, și apoi dorim să revenim să citim din el

```
int fflush(FILE *stream);
scrie în fișier tampoanele de date nescrise pt. fluxul de ieșire stream
```

Funcții sistem (în `stdlib.h`)

Ieșirea din program

`void abort();` cauzează terminarea *anormală* a programului; efectul pt. tamponanele de date și fișierele deschise depinde de implementare

`int atexit(void (*func)(void));`

Înregistrează funcții pt. apelare la terminarea normală a programului (acțiuni specificate de programator, ex. “doriți să salvați fișierul ?”)

`void exit(int status);` termină *normal* execuția programului,

– întâi se apelează funcțiile înregistrate cu `atexit` în ordine inversă

– se scriu tamponanele, se închid fișierele, se șterg cele temporare

– se returnează sistemului de operare codul întreg dat (v. `int main()`)

Execuția unor comenzi din program

`int system(const char *string);`

se execută comanda `string` (cu argumente) de procesorul de comenzi;

dacă `string` e `NULL`, returnează `!= 0` dacă există procesor de comenzi

Alte funcții de lucru cu fișiere

`int remove(const char *filename);` șterge un fișier

`int rename(const char *old, const char *new);` redenumeste un fișier

ambele funcții returnează 0 la succes și != 0 la eroare

`FILE *tmpfile(void);` creează fișier temporar, deschis în mod `wb+`
șters automat la sfârșit de program; de ex. pt. date temporare mari

`char *tmpnam(char *s);` generează/returnează un nume nou de fișier
numele e copiat și în `s` dacă `s` e nenul (necesar: minim `L_tmpnam` octeți)

`FILE *freopen(const char * filename, const char * mode,
FILE * restrict stream);`

deschide fișierul `filename` și îl asociază cu fluxul `stream`

(redirectează fluxul logic `stream` în fișierul fizic `filename`;

returnează `NULL` în caz de eroare, `stream` la succes

închide un eventual fișier asociat anterior cu `stream`)

poate fi folosit pentru redirectarea intrării/ieșirii din program

Exemplu cu fișiere temporare, comenzi și redirectare

```
#include <stdio.h>
#include <stdlib.h>
int fatal(char *msg) { perror(msg); exit(1); }
int main(int argc, char **argv)
{
    char s[80];
    if (argc != 2) fatal("lipseste nume fișier");
    if (!freopen(argv[1], "r", stdin))
        fatal("eroare la deschidere/redirectare intrare");
    if (!freopen(tmpnam(NULL), "w+", stdout))
        fatal("eroare la creare/redirectare ieșire");
    if (system("sort")) fatal("eroare la sortare");
    if (fseek(stdout, 0, SEEK_SET)) fatal("eroare re poziționare");
    /* prelucram fisierul de iesire; aici doar tiparim */
    while (fgets(s, 80, stdout)) fprintf(stderr, "%s", s);
    fclose(stdin); fclose(stdout);
    return 0;
}
```

Alte funcții de intrare/ieșire

Funcțiile de tipul printf/scanf pot avea ca sursă/dest. și șiruri de char.

```
int sprintf(char *s, const char *format, ...);
```

```
int sscanf(const char *s, const char *format, ...);
```

Pentru sprintf, poate apărea problema depășirii tabloului în care se scrie, dacă acesta nu e dimensionat corect (suficient). Se recomandă:

```
int snprintf(char *str, size_t size, const char *format, ...);
```

în care scrierea e limitată la size caractere \Rightarrow variantă sigură

Între funcții similare, trebuie alese cele corespunzătoare situației. Ex:

```
int n, r; char *s, *end;
```

```
n = atoi(s); /* dacă suntem siguri; nu semnaleză erori */
```

```
n = strtol(s, &end, 10); /* se pot testa erori (s == end) și  
prelucra mai departe de la end */
```

```
r = sscanf(s, "%d", &n); /* se pot testa erori (r != 1)
```

```
dar punctul de oprire în s nu e explicit (eventual cu %n) */
```

Funcții cu număr variabil de argumente

Funcțiile de tipul `printf/scanf` au număr variabil de argumente (...)

Pentru a implementa o astfel de funcție, trebuie un mod de acces la argumentele cu număr variabil, pornind de la ultimul arg. numit.

⇒ limbajul C definește o serie de macro-uri în `stdarg.h`

– tipul `va_list` pentru a reține informații despre lista de argumente

```
void va_start(va_list ap, ultimarg);
```

– inițializează `ap` pornind de la adresa ultimului argument

```
tip va_arg(va_list ap, tip);
```

– returnează următorul argument din listă, presupus a fi de tipul `tip` apelată repetat pentru fiecare argument; tipul argumentelor și numărul lor trebuie deduse din argumentele fixe (ex. formatul la `printf/scanf`)

```
void va_copy(va_list dest, va_list src);
```

– copiază un `va_list`, inclusiv punctul curent de prelucrare atins

```
void va_end(va_list ap);
```

– apelat pentru încheierea corectă a prelucrării argumentelor

- extensii (macro-uri) pentru scrierea mai concisă a programelor
- preprocesorul efectuează transformarea într-un program C propriu-zis
- directivele de preprocesare au caracterul # la început de linie

```
#include <numefisier>    sau    #include "numefisier"
```

- include textual fișierul numit (în mod tipic definiții)

(a doua variantă: caută întâi în directorul curent apoi în cele standard)

```
#define LEN 20 /* substituție textuală simplă */
```

```
int tab[LEN]; /* pentru definirea de constante simbolice */
```

```
for (i = 0; i < LEN; i++) { ... } /* mai robust la modificări */
```

forma generală: #define nume(arg1,...,argn) substituție

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

```
#define swapint(a, b) { int tmp; tmp = a; a = b; b = tmp; }
```

Obs: substituția se face textual \Rightarrow pot apărea probleme subtile

- folosiți paranteze în jurul argumentelor (evită erori de precedență)
 - argumentele: evaluate la fiecare apariție textuală (ex. de 2x în max)
- \Rightarrow rezultat incorect la evaluarea repetată a expresiilor cu efect lateral

Compilarea condițională

pt. a compila selectiv porțiuni de cod din program în funcție de opțiuni (caracteristici arhitecturale; pt. depanare; funcționalitate în plus; etc)

Sintaxa: *grup-cod ::= test-if grup-cod grupuri-elif_{opt} grup-else_{opt} #endif*

test-if ::= #if expr-const | #ifdef identificador | #ifndef identificador

grup-elif ::= #elif expr-const grup-cod (toate # apar

grup-else ::= #else grup-cod pe linie nouă)

expr-const ::= cele obișnuite | defined identificador

```
#define DEBUG /* dacă depanăm */          #if defined __GNUC__ /* compiler GNU */
/* cod obișnuit */                          #if __GNUC__ == 2    /* versiunea 2 */
#ifdef DEBUG                                  /* cod specific pt. versiune */
printf("am ajuns aici, x = ...");          #else                /* altă versiune */
#endif                                        /* cod specific pt. altă versiune */
/* alt cod obișnuit */                      #endif
                                           #endif
```

Identificatori predefiniți: `__FILE__` `__LINE__` `__DATE__` `__TIME__`

Ex: `fprintf(stderr, "eroare în %s linia %d\n", __FILE__, __LINE__);`