Fişiere

15 noiembrie 2005

Lucrul cu fişiere

La nivel de *utilizator*, ne referim la un fişier prin *nume*.
La nivelul *interfeţei de programare*, bibliotecile limbajului C definesc un tip `FILE` cu elementele necesare accesului la fişier (poziţia curentă în fişier, tamponul de date, indicatori de eroare şi `EOF`)
Structura internă a tipului `FILE`: invizibilă programatorului;
folosim doar pointeri `FILE *`, prin intermediul funcţiilor de bibliotecă.

Din punct de vedere logic, un fişier e un flux (*stream*) de octeţi

Fişiere (`FILE *`) standard predefinite (şi deschise automat la rulare)
`stdin`: fişierul standard de intrare (implicit: tastatura)
`stdout`: fişierul standard de ieşire (implicit: ecranul)
`stderr`: fişierul 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 ieşire

Orice operaţie cu fişiere poate rezulta în eroare (din multe motive)
⇒ e *obligatorie* testarea valoarii returnate (codului de eroare)

Deschiderea şi închiderea fişierelor

`FILE *fopen (const char *path, const char *mode);`
– arg. 1: numele fişierului (absolut sau faţă de directorul curent)
– arg. 2: modul de deschidere; primul caracter semnifică:
`r`: deschidere pentru citire (fişierul trebuie să existe)
`w`, `a`: deschidere pt. scriere; dacă fişierul nu există, e creat;
dacă există, `w` trunchiază la zero; `a` (append) adaugă la sfârşit
În plus, şirul de caractere pt. modul de deschidere mai poate conţine:
`+` permite şi celălalt mod (`r`/`w`) în plus faţă de cel din primul caracter
`b` deschide fişierul în mod binar (implicit: în mod text)
– returnează `NULL` în caz de eroare (trebuie testat !!!)
– altfel, valoarea returnată se foloseşte pt. lucrul în continuare

`int fclose(FILE *stream);`
– scrie orice a rămas în tampoanele de date, închide fişierul
– returnează 0 în caz de succes, `EOF` în caz de eroare

Deschiderea, închiderea şi lucrul cu fişierele

Secvenţa tipică de lucru cu un fişier (ex. pt. citire)

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

La intrarea-ieşirea în mod *text* se pot petrece diverse *conversii* în funcţie 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 spaţii; ultimul caracter e `\n`
⇒ pentru *orice alte situaţii*, deschideţi fişierele în mod *binar*
(asigură corespondenţa exactă între conţinutul scris şi citit)

Citirea şi scrierea într-un fişier folosesc acelaşi indicator de poziţie ⇒
Pentru un fişier deschis în mod dual (cu +), nu se va citi direct după scriere fără a goli tampoanele (`fflush`) sau a repoziţiona indicatorul; nu se scrie direct după citire fără repoziţionarea indicatorului sau `EOF`

Citire/scriere (d)in fişiere

Cu funcţii echivalente celor folosite până acum:

```
int fputc(int c, FILE *stream);  /* scrie caracter în fişier */
int fgetc(FILE *stream);      /* citeşte caracter din fişier */
 /* getc, putc: la fel ca şi 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 şir */
int puts(const char *s); /* scrie şirul şi apoi \n la ieşire */
char *fgets(char *s, int size, FILE *stream);
```

– citeşte până la (inclusiv) linie nouă, sau max. `size - 1` caractere, adaugă `'\0'` la sfârşit ⇒ citirea sigură a unei linii, fără depăşire

NU FOLOSIŢI niciodată funcţia `gets()`, nu e protejată la depăşire!

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;
}
```

Funcţii de eroare

`void clearerr(FILE *stream);`
reesetează indicatorii de sfârşit de fişier şi eroare pentru fişierul dat

`int feof(FILE *stream); /* != 0: ajuns la sfârşit de fişier */`
`int ferror(FILE *stream); /* != 0 la eroare pt. acel fişier */`

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 funcţia `char *strerror(int errnum);` din `string.h` care returnează un şir de caractere cu descrierea erorii

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

Citire şi scrierea directă

Până acum: funcţii orientate pe caractere, linii, formatare (fişiere text)
Pentru a citi/scrie direct un număr dat de octeţi, neinterpretaţi:
`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 octeţi */`

Funcţiile întorc numărul obiectelor *complete* citite/scrise corect.
Dacă e mai mic decât cel dat, cauza se află din `feof` şi `ferror`

În C, nu există fişiere tipizate (`file of` din PASCAL);
putem însă defini astfel de funcţii 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 fişiere

```
#include <errno.h>
#include <stdio.h>
#define MAX 512
int filecopy(FILE *fi, FILE *fo) {
  char buf[MAX];
  int size; /* nr. octeţi citiţi */
  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 fişiere (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;
}
```


Funcţii de poziţionare, etc.

Pe lângă citire/scriere secvenţială, e posibilă poziţionarea în fişier:
`long ftell(FILE *stream); /* pozitia de la începutul fişierului */`
`int fseek(FILE *stream, long offset, int whence); /* poziţionare */`
Al treilea parametru: punctul de referinţă pt. poziţionarea cu `offset`:
`SEEK_SET` (început), `SEEK_CUR` (punctul curent), `SEEK_END` (sfârşit)

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

Repoziţionarea trebuie efectuată:
– când dorim sa "sărim" peste o anumită porţiune din fişier
– când fişierul a fost scris, şi apoi dorim să revenim să citim din el

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

Funcţii sistem (în `stdlib.h`)

Ieşirea din program

`void abort();` cauzează terminarea *anormală* a programului; efectul pt. tampoanele de date şi fişierele deschise depinde de implementare

`int atexit(void (*func)(void));`
înregistrează funcţii pt. apelare la terminarea normală a programului (acţiuni specificate de programator, ex. "doriţi să salvaţi fişierul ?")
`void exit(int status);` termină *normal* execuţia programului,
– întâi se apelează funcţiile înregistrate cu `atexit` în ordine inversă
– se scriu tampoanele, se închid fişierele, se şterg cele temporare
– se returnează sistemului de operare codul întreg dat (v. `int main()`

Execuţia 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 funcţii de lucru cu fişiere

`int remove(const char *filename);` şterge un fişier
`int rename(const char *old, const char *new);` redenumeşte un fişier
ambele funcţii returnează 0 la succes şi != 0 la eroare

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

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

```
FILE *freopen(const char * filename, const char * mode,
              FILE * restrict stream);
```
deschide fişierul `filename` şi îl asociază cu fluxul `stream`
(redirectează fluxul logic `stream` în fişierul fizic `filename`;
returnează `NULL` în caz de eroare, `stream` la succes
închide un eventual fişier asociat anterior cu `stream`)
poate fi folosit pentru redirectarea intrării/ieşirii din program

Exemplu cu fişiere temporare, comenzi şi 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 fişier");
  if (!freopen(argv[1], "r", stdin))
    fatal("eroare la deschidere/redirectare intrare");
  if (!freopen(tmpnam(NULL), "w+", stdout))
    fatal("eroare la creare/redirectare ieşire");
  if (system("sort")) fatal("eroare la sortare");
  if (fseek(stdout, 0, SEEK_SET)) fatal("eroare repoziţionare");
  /* prelucram fisierul de iesire; aici doar tiparim */
  while (fgets(s, 80, stdout)) fprintf(stderr, "%s", s);
  fclose(stdin); fclose(stdout);
  return 0;
}
```


Alte funcţii de intrare/ieşire

Funcţiile de tipul `printf/scanf` pot avea ca sursă/dest. şi şiruri de char.
`int sprintf(char *s, const char *format, ...);`
`int sscanf(const char *s, const char *format, ...);`

Pentru `sprintf`, poate apărea problema depăşirii 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 ⇒ variantă sigură

Între funcţii similare, trebuie alese cele corespunzătoare situaţiei. Ex:

```
int n, r; char *s, *end;
n = atoi(s); /* dacă suntem siguri; nu semnalează erori */
n = strtol(s, &end, 10); /* se pot testa erori (s == end) şi
                            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) */
```


Funcţii cu număr variabil de argumente

Funcţiile de tipul `printf/scanf` au număr variabil de argumente (...)
Pentru a implementa o astfel de funcţie, trebuie un mod de acces la argumentele cu număr variabil, pornind de la ultimul arg. numit.
⇒ limbajul C defineşte o serie de macro-uri în `stdarg.h`
– tipul `va_list` pentru a reţine informaţii despre lista de argumente
`void va_start(va_list ap,`*ultimarg*`);`
– iniţializează `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 şi numărul lor trebuie deduse din argumentele fixe (ex. formatul la `print/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

Preprocesorul C

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

`#include <numefisier>` sau `#include "numefisier"`
– include textual fişierul numit (în mod tipic definiţii)
(a doua variantă: caută întâi în directorul curent apoi în cele standard)

```
#define LEN 20 /* substituţie 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)` substituţie

```
#define max(A, B) ((A) > (B) ? (A) : (B))
#define swapint(a, b) { int tmp; tmp = a; a = b; b = tmp; }
```

Obs: substituţia se face textual ⇒ pot apărea probleme subtile
– folosiţi paranteze în jurul argumentelor (evită erori de precedenţă)
– argumentele: evaluate la fiecare apariţie textuală (ex. de 2x în max)
⇒ rezultat incorect la evaluarea repetată a expresiilor cu efect lateral

Compilarea condiţională

pt. a compila selectiv porţiuni de cod din program în funcţie de opţiuni (caracteristici arhitecturale; pt. depanare; funcţionalitate în plus; etc)

Sintaxa: *grup-cod* ::= *test-if grup-cod grupuri-elif*$_{opt}$ *grup-else*$_{opt}$ `#endif`
test-if ::= `#if` *expr-const* | `#ifdef` *identificator* | `#ifndef` *identificator*
grup-elif ::= `#elif` *expr-const grup-cod* (toate # apar
grup-else ::= `#else` *grup-cod* pe linie nouă)
expr-const ::= cele obişnuite | `defined` *identificator*

```
#define DEBUG /* dacă depanăm */
/* cod obişnuit */
#ifdef DEBUG
printf("am ajuns aici, x = ...");
#endif
/* alt cod obişnuit */
```

```
#if defined __GNUC__ /* compilator GNU */
#if __GNUC__ == 2    /* versiunea 2 */
  /* cod specific pt. versiune */
#else      /* altă versiune */
  /* cod specific pt. altă versiune */
#endif
#endif
```

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