Limbaje de programare

Fișiere. Structuri

3 decembrie 2012

Fișiere text și binare

Un *fișier* e o secvență de date (octeți) stocată pe un mediu extern.

Fișiere *text*: conțin caractere grupate în *linii* terminate cu `\n`
fișiere `.txt`, programe `.c`, `.c++`, pagini `.html`, etc.

La prelucrare pot fi transformate după convenții date de sistem:
de ex. `\r\n` (Windows) în loc de `\n` (Unix)
$\Rightarrow$ ceea ce se citește poate diferi de ce s-a scris

(e la fel doar dacă textul conține doar caractere tipăribile, tab și `\n` și nu conține spații înainte de `\n`)

Fișiere *binare*: secvențe arbitrare de octeți (nu neapărat text)
fișiere executabile, `.doc`, `PDF`, imagini, video/audio, `.zip`, etc.

Prelucrarea în format binar se face *fără conversii* de caractere
$\Rightarrow$ octeții sunt citiți exact la fel cum au fost scriși

Orice fișier (inclusiv text) poate fi deschis / prelucrat în mod binar

Lucrul cu fișiere în trei pași

1. Deschiderea fișierului: funcția `fopen`
2. Prelucrarea: citire, scriere, poziționare
   `(f)getc, (f)putc, fgets, fputs, fscanf, fprintf, fread, fwrite, fseek, ftell`
3. Închiderea fișierului: funcția `fclose`

`fopen`: se referă la fișier prin *nume* (șir, `char *`)
produce un pointer de tip `FILE *` spre o structură de date internă (fișierul e gata de lucru)

Restul funcțiilor *folosesc pointerul* de tip `FILE *` returnat de `fopen`
NU se mai referă la fișier prin nume

Moduri de deschidere a fișierelor

`r` (read): deschide pentru citire (trebuie să existe)
`w` (write): deschide pentru scriere (șters dacă există, creat altfel)
`a` (append): deschidere pentru adăugare (creat dacă nu există; altfel poziționare la sfârșit, datele existente rămân)

După primul caracter (`r`, `w`, `a`) pot urma:
`+` (ex. `r+`, `w+`, `a+`): fișierul se deschide în modul indicat, dar poate fi folosit și în modul complementar (scriere / citire)

Modul implicit e *text* (folosit pentru caractere tipăribile, tab, `\n`)
`b`: deschidere în mod binar (pentru orice alt format)

`x`: (eXclusiv) poate fi ultimul caracter din modul de deschidere fișierul nu trebuie să existe, și nu se permite acces partajat

Exemple: `rb+` (citire și scriere, binar), `wx`, `wb+x`, `a+x`, etc.

Deschiderea si închiderea fișierelor

`FILE *fopen (const char *pathname, const char *mode)`

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

arg. 2: *șir* cu *modul de deschidere*: `r`, `w`, sau `a`; opțional `+`, `b`, `x`

```
FILE *f1 = fopen("/home/student/t.txt", "r");   // nume fix
FILE *f2 = fopen(argv[2], "w"); // numele: al doilea arg. la cmd

char nume[32];                          // exemplu cu numele citit
if (scanf("%31s", nume) == 1) {
  FILE *f = fopen(nume, "ab+");
} // deschide in mod binar, pentru adaugare+citire
```

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

Altfel, pointerul returnat (un `FILE *`) e folosit în alte funcții

`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

Fișiere standard. Redirectare

În `stdio.h` sunt definite:
`stdin`: fișierul standard de intrare (normal: tastatura)
  de aici citesc `getchar, scanf`
`stdout`: fișierul standard de ieșire (normal: ecranul)
  aici scriu `putchar, printf, puts`
`stderr`: fișierul standard de eroare (normal: ecranul)

Aceste fișiere sunt deschise automat la rularea programului

E bine ca mesajele de eroare să fie scrise la `stderr`, pentru a le putea separa la nevoie de scrierea rezultatelor la `stdout`

Putem *redirecta* fișierele standard spre/dinspre alte fișiere:
  intrarea: `program < in.txt` (va citi din `in.txt`)
  ieșirea: `program > out.txt` (va scrie în `out.txt`)
  ambele: `program < in.txt > out.txt`

Citirea/scrierea fișierelor

Câte un *caracter*

```
int fputc(int c, FILE *stream) // scrie caracter în fişier
int fgetc(FILE *stream)   // citeşte caracter din fişier
 // getc, putc: ca si fgetc, fputc
int ungetc(int c, FILE *stream) // pune caracter c înapoi
```

Citire/scriere *formatată* (la fel ca `printf/scanf`, din fișierul indicat)

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

Câte o *linie de text*

```
int fputs(const char *s, FILE *stream)  // scrie un şir
int puts(const char *s)  // scrie şirul + \n la stdout
char *fgets(char *s, int size, FILE *stream)
//citeşte din fişier în tabloul s, max. size caract. + '\0'
```

Lucrul cu fișierele

Secvența tipică de lucru cu un fișier (exemplu pentru citire)

```
FILE *f = fopen(nume, "r"); // sau "rb", "w", "a", etc.
if (f) {    // f != NULL, s-a deschis
  // foloseşte fişierul f
  if (fclose(f)) { /* eroare la închidere */ }
} else { /* tratează eroare la deschidere */ }
```

Exemplu cu numele fișierului dat pe linia de comandă:

```
#include <errno.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
  if (argc != 2) {
    fprintf(stderr, "folosire corectă: program numefisier\n");
    return 1; // sau alt cod de eroare
  }
  FILE *fp = fopen(argv[1], "r");
  if (!fp) { perror("eroare la deschidere"); return errno; }
  // foloseste fisierul, apoi inchide
  return 0;
}
```

Funcții de eroare

`int feof(FILE *stream)` != 0: ajuns la sfârșit de fișier
`int ferror(FILE *stream)` != 0 dacă fișierul a avut erori

`void clearerr(FILE *stream)`
resetează indicatorii de sfârșit de fișier și eroare pentru fișierul dat

Coduri de eroare
variabila globală `int errno` declarată în `errno.h` conține codul ultimei erori într-o funcție de bibliotecă
(operație nepermisă, fișier inexistent, memorie unsuficientă, etc.)

Funcția `void perror(const char *s)` din `stdio.h` tipărește mesajul `s` dat de utilizator, un `:` și apoi descrierea erorii
(aceeași ca și `char *strerror(int errnum)` din `string.h`)

Citire și scrierea în format binar

Până acum am folosit funcții pentru citirea în mod text
Pentru a citi/scrie direct un număr dat de octeți, neinterpretați:

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *strm)
size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE *strm)
```

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 `ferror`

Putem defini funcții pentru a scrie/citi numere în format binar

```
int32_t readint32(FILE *stream) // intreg pe 32 de biţi
{ int32_t n; fread(&n, sizeof(int32_t), 1, stream); return n; }
```

Poziționarea în fișier

`long ftell(FILE *stream)` returnează poziția relativ la început

Pe lângă citire/scriere secvențială, e posibilă poziționarea în fișier:
`int fseek(FILE *stream, long offset, int whence)`
Poziționează indicatorul de citire/scriere la deplasamentul `offset` față de punctul de referință precizat prin parametrul 3:
`SEEK_SET` (început), `SEEK_CUR` (punctul curent), `SEEK_END` (sfârșit)

`void rewind(FILE *stream)` repoziționează indicatorul la început
la fel ca `fseek(stream, 0L, SEEK_SET); clearerr(stream);`

Repoziționarea e utilă pentru a "sări" peste o parte din fișier
Trebuie folosit `fseek`/`fflush` când comutăm între citire și scriere.
NU e posibilă poziționarea în orice fișier! (ex.`stdin`/`stdout`)

`int fflush(FILE *stream)`
scrie tampoanele de date nescrise pt. fișierul de ieșire indicat

Tipuri definite de utilizator
(structuri, uniuni, enumerări)

Tipuri structură

Grupează elemente de tipuri diferite, legate logic între ele

```
struct lung {   // defineste tipul 'struct lung'
  double val;
  char unit[3];
};
struct lung d1 = { 60, "km" }; // declară variabilă inițializată

struct vect {   // definește tipul 'struct vect'
  double x, y;
} v1, v2;       // și în același timp declară două variabile
```

Elementele unei structuri se numesc *câmpuri* (engl. fields)
pot fi de orice tip, dar *NU* de *același* tip structură (nu recursiv)

Folosirea câmpurilor: cu sintaxa *nume_variabila*.*nume_câmp*
punctul . e *operatorul de selecție* (e un operator postfix)

```
struct vect p1; p1.x=2; p1.y=3; printf("%f %f\n", p1.x, p1.y);
```

Exemplu de structură

```
struct student {      // numele complet de tip (incl. "struct")
  char nume[32];      // tablou alocat cu numar fix de caractere
  char *domiciliu;    // doar ADRESA, nu aloca si memoria pt. sir
} s;                  // declara var. s de tip struct student
```

Într-un câmp tablou de caractere poate fi copiată direct o valoare

`strcpy(s.nume, "Stefanovici"); // NU atribuire, e tablou`

Un câmp și poate fi *atribuit* cu alt șir:

`s.domiciliu = "str. Linistei nr. 2";`

Dacă sursa e un tablou care își schimbă valoarea, facem o *copie* a șirului, alocată dinamic:

`s.domiciliu = strdup(buf);` (dacă apoi în buf se citește altceva)

Numele câmpurilor se văd doar în interiorul structurii

$\Rightarrow$ nu putem folosi doar numele cămpului, doar `numevar.câmp`

$\Rightarrow$ tipuri structuri diferite pot avea câmpuri numite la fel

Folosirea structurilor (cont.)

Structurile *pot* fi atribuite în totalitatea lor.

```
struct vect v1 = {2, 3}, v2;  v2 = v1;
```

Structurile *pot* fi transmise către / returnate de funcții.
(când sunt mari, se preferă transmiterea / returnarea de pointeri)

```
struct vect add(struct vect v1, struct vect v2)
{
  struct vect v;
  v.x = v1.x + v2.x; v.y = v1.y + v2.y;
  return v;
}
```

Putem scrie *valori compuse* de tip structură indicând tipul între ( )

```
struct vect v1; v1 = (struct vect){-4, 5};
```

NU putem compara structuri cu operatori logici
⇒ trebuie comparate câmp cu câmp:

```
if (v1.x==v2.x && v1.y==v2.y) ...
```

Declararea de tipuri

Putem da noi nume la tipuri existente (mai expresive; mai scurte)

Forma generală: `typedef nume-tip-existent nume-tip-nou;`

Ex. `typedef double real;` `typedef struct vect vect_t;`

```
typedef int (*cmpfun)(const void *, const void *);
```

(ca declarația de variabile + `typedef` în față $\Rightarrow$ declară un *tip*)

Numele nou se poate da direct în definirea tipului:

```
typedef struct student { /* ceva campuri */ } student_t;
```

putem omite eticheta după `struct` (folosim apoi doar numele nou)

```
typedef struct { /* ceva campuri */ } student_t;
```

sau definim întâi tipul structură și apoi sinonimul pentru el

```
struct student { /* ceva campuri */ }; // defineşte tipul
typedef struct student student_t;  // defineşte sinonimul
```

Pointeri la structuri

Putem accesa câmpurile cu ajutorul unui pointer la structură:
`struct student *p, s;  p = &s; (*p).nota_dipl = 9.50;`
Operatorul `->` e echivalent cu indirectarea urmată de selecție:
`pointer->numecâmp` echivalent cu `(*pointer).numecâmp`
Operatorii `.` și `->` au *precedența cea mai ridicată*, ca și `()` și `[]`

Atenție la ordinea de evaluare !

`p->x++`	înseamnă	`(p->x)++`	(`->` e prioritar)
`++p->x`	înseamnă	`++(p->x)`	(`->` e prioritar)
`*p->x`	înseamnă	`*(p->x)`	(`->` e prioritar)
`*p->s++`	înseamnă	`*((p->s)++)`	(`++` e prioritar lui `*`)

Structuri și tablouri

În C, tipurile agregat (compuse) pot fi combinate arbitrar
(tablouri de structuri, structuri cu câmpuri de tip tablou, etc.)

Tipurile trebuie definite în așa fel încât să grupeze logic datele.

Ex.: două tablouri de aceiași indici, folosite împreună
$\Rightarrow$ înlocuim cu un tablou de element structură:

```
char* nume_luna[12] = { "ianuarie", /* ... , */ "decembrie" };
char zile_luna[12] = { 31, 28, 31, 30, /* ... , */ 30, 31 };
// e preferabilă varianta următoare
struct luna {
  char *nume;
  int zile;
};
struct luna luni[12] = {{"ianuarie",31}, ..., {"decembrie",31}};
```

Structuri de date recursive

Un câmp al unei structuri nu poate fi o structură de același tip (s-ar obține o structură de dimensiune infinită/nedefinită!).
Poate fi însă *adresa* unei structuri de același tip (un pointer)!
$\Rightarrow$ structuri de date recursive, înlănțuite (liste, arbori, etc.)

```
struct wl {          // struct wl e un tip, incomplet definit
  char *word;        // cuvantul: informaţia propriu-zisă
  struct wl *next;   // pointer la structura de acelaşi tip
};                   // acum definiţia tipului e completă
```

Un arbore binar, având în noduri numere întregi:

```
typedef struct t tree; // def. tipul incomplet tree = struct t
struct t {
  int val;
  tree *left, *right;  // foloseşte numele din typedef
};           // aici tipul struct t e complet şi echivalent cu t
```

Structuri cu câmpuri pe biți

Vrem să reprezentăm mai multe informații cât mai compact pe biți.
Ex. dată=întreg pe 32 de biți: sec, min (0-59): 6 biți, ora (0-23), ziua (1-31): 5 biți, luna (1-12): 4 biți), an (1970 + 0-63): 6 biți.

```
struct date_t {    // structură cu câmpuri pe biţi
  unsigned sec, min : 6;  // indică numărul de biţi
  unsigned hour, day: 5;  // se permit tipuri întregi
  unsigned month: 4;
  unsigned year: 6;
} data = {0, 0, 17, 19, 5, 39 };    // 17:00:00, 19.05.(1970+39)
```

Putem scrie direct: `printf("%u.%u\n", data.day, data.month);`

Putem avea câmpuri fără nume: `int:  2; // pe 2 biţi`
sau forța trecerea la memorarea în octetul următor `int:  0;`

Tipul enumerare

Tipul enumerare: dă nume unui șir de valori numerice.
⇒ folosit când e mai sugestiv de scris un nume decât un număr

```
enum luni_sc {ian=1, feb, mar, apr, mai, iun, oct=10, nov, dec};
```

definește tipul `enum luni_sc` (`enum` *e parte din nume* !)
Implicit, șirul valorilor e crescător începând cu 0
Putem specifica și explicit valori (și o valoare se poate repeta)
Un tip enumerare e un tip întreg
⇒ variabilele/valorile enumerare se folosesc ca și întregi

```
enum {D, L, Ma, Mc, J, V, S} zi; // tip anonim+variabila zi
int nr_ore_lucru[7];             // număr de ore pe zi
for (zi = L; zi <= V; zi++) nr_ore_lucru[zi] = 8;
```

Un nume de constantă nu poate fi folosit în mai multe enumerări

Uniuni

Folosite pentru a reține valori care pot avea tipuri *diferite*.

Sintaxa: ca la structuri, dar cu cuvântul cheie `union`

Lista de câmpuri reprezintă o listă de variante, pentru fiecare tip:

- o variabilă structură conține *toate* câmpurile declarate
- o variabilă uniune conține exact *una* din variantele date
  (dimensiunea tipului e dată de cel mai mare câmp)

```
union {          // tip uniune, fără nume
  int i;
  double r;
  char *s;
} v;           // trei variante pentru fiecare tip de valoare
enum { INT, REAL, SIR } tip;   // ţine minte varianta găsită
char s[32]; if (scanf("%31s", s) == 1) {
  if (isdigit(*s))  // începe cu cifră ? conţine şi punct ?
    if (strchr(s, '.')) { sscanf(s, "%lf", &v.r); tip = REAL; }
    else { sscanf(s, "%d", &v.i); tip = INT; }
  else { v.s = strdup(s); tip = SIR; }
}
```