

Caractere și întregi

`signed char` e un *tip întreg* (ca și `short`, `int`, `long`, `long long`)
`char` e `signed char` (-128..127) sau `unsigned char` (0..255) (neprecizat)
 ⇒ poate fi folosit (și e convertit) ca un întreg în expresii

Conversii cifră ↔ întreg: '5' == '0' + 5; 7 == '7' - '0' etc.

(cifre, litere mari, mici: trei blocuri de caractere în tabela ASCII)

Funcțiile din `ctype.h` `isalpha()` etc. returnează != 0 sau 0, NU 1 sau 0
 ⇒ scrieți: `if (isdigit(c))` și nu `if (isdigit(c) == 1)`

Funcțiile de clasificare: definite și pentru EOF == -1 (toate false)

Atenție! >> la numere cu semn poate introduce bitul de semn, nu 0
 ⇒ folosiți `unsigned` pentru efect bine definit (introduce 0)

Un caracter ('a', valoare întregă) NU e un șir ("a", valoare adresă).
 ⇒ NU putem scrie `atoi('9')`; `strcat(s, 'b')`; etc.

Prelucrare de texte și testare de EOF

Funcțiile standard au nevoie de \0 pentru a detecta sfârșitul unui șir

La validarea datelor, testați valoarea returnată de `scanf`

La corectare, goliți tamponul de intrare: `while (getchar() != '\n');`

Testați corect sfârșitul de fișier!

Declarați caracterul ca `int` pentru `while ((c = getchar()) != EOF)`

Testați de EOF *LA* citire, *NU* înainte sau după. Corect.

`while (scanf("%d", &n) == 1) ...` (nu doar != 0)

`while (fgets(s, 80, stdin)) ...`

Evitați *blocarea* la sfârșit de fișier:

`while (isspace(c = getchar())) ...` iese pentru `c == EOF` (false)

`while (!isspace(c = getchar())) ...` se blochează la `c == EOF` (true)

Limitări de memorie

Orice tablou în C are dimensiune *finită* și *precizată*

⇒ nu există tablouri de dimensiune necunoscută

`int tab[]`; are 1 element!

Când accesăm (ex. `u[1]`) un tablou NU avem voie să depășim dimensiunea alocată

– la `scanf`. NU: `%s` sau `%[A-Z]` ci de ex. `%19s`. NU: `gets`. DA: `fgets`

– la `%s`: permitem 1 mai puțin decât tabloul (loc pentru \0)

– `fgets` citește automat cu 1 mai puțin decât parametrul

(atenție: `%s` citește *cuvânt*, `fgets` citește *linie*)

– la parcurgere. NU: `while ((c = getchar()) != EOF) tab[i++] = c;`
 (trebuie verificată depășirea indicelui `i`)

Pointeri

O declarație de pointer: `tip *ptr`; spune: *voi avea* un obiect

(sau tablou) de tipul `tip`, dar încă *nu există*, *n-am memorie* pentru el

⇒ nu-l putem folosi înainte de a-i atribui o zonă de memorie!

(adresa unei variabile existente, sau zonă alocată dinamic)

– *Alocăm static*: când cunoaștem dinainte dimensiunea.

`char s[80]`; NU ne complicăm: `char *s`; `s = malloc(80)`; `if (!s) ...`

– *Folosim malloc*: când știm dimensiunea în momentul apelului.

`printf("Câte numere"); scanf("%d", &n); tab=malloc(n*sizeof(int));`

`l=strlen(s); if (p=malloc(l+1)) strcpy(p, s); else ...`

– *Folosim realloc*: când inițial nu am alocat cât trebuie

întodeauna folosim pointerul *nou* returnat (poate muta memoria)

Pointeri și tablouri

Numele unui tablou e *adresa* sa de început (o *constantă* !)

⇒ numele unui tablou (incl. șir de caractere) e un *pointer* (constant)

⇒ `tablou[indice]` sau `pointer[indice]` e același lucru

⇒ `char a[10], b[10]; a = b`; NU copiază tablouri, ci atribuie adrese!

(și dă eroare de compilare, pentru că `a` e constantă !)

`s1==s2` compară pointerii (se suprapun?), nu conținutul: `strcmp(s1, s2)`

⇒ NU are sens să scriem `void f(char s[20])`.

scriem: `void f(char tab[])` sau `void f(char *tab)`

(NU se transmit 20 de caractere, se transmite adresa tabloului)

Tablouri de șiruri de caractere:

`char tab[NUM][LEN]`; (dacă cunoaștem lungimea maximă a șirului)

`char *tab[NUM]`; fiecare element (adresă) trebuie atribuit (*alocat*) !

Pointeri ca parametri și rezultate

Orice parametru transmis trebuie să aibă o valoare validă, utilizabilă !

⇒ un pointer transmis trebuie să indice o zonă de memorie validă!

– zona respectivă e folosită la citire sau scriere, depinzând de funcție

NU: `char *p`; `strcpy(p, "un sir");` `p` neinițializat/nealocat !

NU: `char **endptr`; `l=strtol(sir, endptr, 10)`; `endptr` e nealocat!

DA: `char *endptr`; `l=strtol(sir, &endptr, 10)`; scrie valoare la `&endptr`

O funcție nu poate întoarce adresa unei variabile *locale* (ex. tablou).

– e alocată pe stivă ⇒ va *dispare* odată cu ieșirea din corpul funcției

⇒ un pointer returnat de o funcție provine din a) un parametru;

b) o variabilă globală (problematic: suprascriere); c) alocare dinamică

Un pointer returnat de o funcție trebuie să fie *valid* sau *NULL*.