

Funcții pentru manipularea timpului (`time.h`)

Tipuri definite pentru reprezentarea timpului: `clock_t` și `time_t`
(sunt de fapt tipuri aritmetice, de ex. `unsigned` sau `unsigned long`)

```
clock_t clock(void);
```

returnează timpul scurs de la lansarea programului, în unități de ceas
date de constanta `CLOCKS_PER_SEC` (în standardul POSIX, 1 milion)
e o aproximație dependentă de granularitatea ceasului de timp real
poate interveni depășire (pe sistem de 32 de biți, după cca 72 min.)

```
time_t time(time_t *timer);
```

returnează o valoare aritmetică reprezentând data/ora curentă
(în UNIX, numărul de secunde trecute de la 1 ian. 1970 UTC)
dacă argumentul pointer e nenul, valoarea e stocată și la acea adresă

```
double difftime(time_t time1, time_t time0);
```

returnează diferența exprimată în secunde, ca `double`

Reprezentarea timpului (cont.)

`time.h` definește și un tip cu componentele unei date/ore:

```
struct tm {
    int    tm_sec;        /* seconds */
    int    tm_min;        /* minutes */
    int    tm_hour;       /* hours */
    int    tm_mday;       /* day of the month */
    int    tm_mon;        /* month */
    int    tm_year;       /* year */
    int    tm_wday;       /* day of the week */
    int    tm_yday;       /* day in the year */
    int    tm_isdst;      /* daylight saving time */
};
```

`time_t mktime(struct tm *tm);` calculează valoarea reprezentată de o astfel de structură; completează `tm_wday` și `tm_yday`.

```
struct tm *gmtime(const time_t *timep);
```

```
struct tm *localtime(const time_t *timep);
```

transformă timpul dat în format structură, considerat timp universal (UTC) sau relativ la zona locală de timp; returnează pointer spre o zonă statică ce va fi suprascrisă cu o nouă valoare la următorul apel

Alte funcții pentru conversii în șir: `asctime()`, `ctime()` – v. manualul

Funcții pentru numere pseudoaleatoare (`stdlib.h`)

Numerele generate sunt *pseudo*aleatoare (de fapt deterministe, bazate pe un algoritm, dar cu distribuție cât mai uniformă)
(numere cu adevărat aleatoare ar trebui să fie bazate pe fenomene fizice, ex. aruncarea unei monede sau descompunerea unor particule)

```
int rand(void);
```

returnează un număr pseudoaleator între 0 și `RAND_MAX` (min. 32767)
pt. un număr aleator între 1 și `N` putem folosi `1 + rand() % N`

```
void srand(unsigned int seed);
```

reinițializează generatorul de numere pseudoaleatoare cu valoarea dată
următorul număr va fi generat de `rand()` pornind de la această valoare
fără apelarea ei, două rulări generează același șir de valori cu `rand()`
se poate folosi de ex. cu `srand(unsigned)time(NULL)`;

Considerații despre scrierea programelor

- Programați *modular*!
- Scrieți funcții nu doar pentru cod repetat (evitați duplicarea de cod!), ci pentru orice unitate logică cu un scop bine definit
- Programele sunt mult mai lizibile când sunt structurate în funcții!
- Compilați, rulați și testați programele pe măsură ce le dezvoltați!
Scrieți fragmente care implementează incremental funcționalitatea cerută.
Nu scrieți programe mari dintr-o dată, fără pași intermediari!
- Documentați (măcar minimal) variabilele esențiale și funcțiile

Recapitulare

Reprezentarea valorilor

- lucrați cu `sizeof` (octeți!), nu dimensiuni presupuse (2, 4 octeți, etc.)
- atenție la semn (`>>` la dreapta; reprezentat diferit la int și reali)
- atenție la depășire pt. întregi și precizie limitată pentru reali

Prelucrări de texte

- citiți *doar pe lungime limitată*: nu `%s`, `%[]`, `gets()`
- tratați în orice punct cazul de sfârșit de fișier
- nu limitați inutil capacitatea de a trata dimensiuni mari
- prelucrați fragmente cât mai mici (caracter/cuvânt/linie), în funcție de problemă

Recapitulare

Șiruri și conversii numerice

- un șir e definit prin început și terminatorul nul `'\0'`
- nu copiați fragmente inutile; e suficient adesea avansul unui pointer
- folosiți funcțiile standard pentru conversiile șir → număr

Pointeri

- declarând un pointer NU se alocă loc pentru obiect, doar pt. adresa!
- nu returnați dintr-o funcție adresa unei variabile locale (ex. tablou)

Fișiere

- tratați cazurile de eroare (și pt. orice interacțiune cu exteriorul)
- diferențiați între numerele stocate în format text (șir de cifre) și în format binar (ca în memorie; citiți cu `fread` în tip de dimensiune fixă corespunzătoare (ex. `uint32_t`)).

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 întreagă) 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. `umplem`) 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

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

Pointeri si tablouri

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

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

⇒ tablou[indice] sau pointer[indice] e acelaşi lucru

⇒ `char a[10], b[10]; a = b;` NU copiază tablouri, ci atribuie adrese !
(şi dă eroare de compilare, pentru că a e constantă !)

`s1==s2` compară pointerii (se suprapun?), nu conţinutul: `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 şiruri de caractere:

`char tab[NUM][LEN];` (dacă cunoaştem lungimea maximă a şirului)

`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**.