

# Pointeri

3 noiembrie 2003

## Variabile și adrese

---

În limbajul C:

- o variabilă are: nume, valoare, adresa unde e memorată valoarea
- oricare două variabile ocupă spațiu de memorie distinct
- memoria pentru variabile e alocată implicit:
  - pe toată durata programului, pentru variabilele globale și statice
  - la fiecare activare a unui bloc, pentru variabilele locale blocului

Cu elementele de limbaj studiate până acum:

- o variabilă poate fi referită *doar* prin numele ei (nu există *alias*-uri)
- nu ne putem referi la adresa unei variabile
- valoarea unei variabile e modificată doar prin *atribuire explicită*
  - nu și ca parametru al unei funcții (transmiterea se face prin valoare)
- Obs: la *citire* (ex. `scanf`) se transmite *adresa* ! (vom discuta)
- nu putem aloca explicit, la rulare, memorie pentru noi variabile

## Declararea pointerilor. Adrese. Dereferențiere

---

Pointer = o variabilă care conține *adresa* altei variabile

### Declararea pointerilor

```
tip *nume_var; /* nume_var e pointer la o valoare de tip */
```

### Operatorul adresă &

operator prefix

- operand: o variabilă (ex. x); rezultat: adresa variabilei &x
- se poate folosi numai pt. *variabile*, nu pt. constante, expresii, etc.
- se poate atribui unei variabile pointer la tipul respectiv:

```
int x; int *p; p = &x;
```

### Operatorul de dereferențiere (indirectare) \*

operator prefix

- operand: pointer; rezultat: referință la obiectul indicat de pointer
- dacă  $p = \&x$ , atunci  $*p$  e efectiv sinonim cu  $x$
- referința  $*p$  poate fi folosită la stânga sau la dreapta unei atribuiri:

```
int x, y, z, *p; p = &x; /* *p înseamnă x */  
z = *p; /* ca și z = x */ *p = y; /* ca și x = y */
```

## Declarații și referințe: observații

---

În C, putem declara/folosi oricâte nivele de indirectare: `char **s;`

O **declarație** `nume_tip declarator ;` se interpretează:

un obiect de aceeași formă ca și declaratorul are tipul `nume_tip`

```
int *p; /* *p este int, deci p e pointer la int */
```

```
char **s; /* **s este char; s e adresa unei adrese de char */
```

```
int m[5][3]; /* m[i][j] e int; m e tablou de 5 tablouri de 3 int */
```

```
char *t[10]; /* *t[i] e char; t e tablou de 10 adrese de char */
```

Noțiune: obiect care poate apărea în stânga atribuirii (*lvalue, referință*)

= variabilă simplă, element de tablou, sau referință prin indirectare `*p;`

– ceilalți operatori produc *expresii* care pot sta doar în dreapta atribuirii:

`a+b, &x, i++, (a > b) ? a : b`

`*` și `&` au **precedența** mai ridicată decât operatorii aritmetici:

```
y = *px + 1; /* cu 1 mai mult decât valoarea indicată de px */
```

dar `*px++` dă valoarea indicată de `px`, și incrementează pointerul `px`

(nu valoarea), pentru că `++` și `*` se evaluează de la dreapta la stânga !

Utilizarea *oricărei* variabile neinițializate e o eroare logică în program !

```
{ int sum; for (i=0; i++ < 10; ) sum += a[i]; /* dar inițial? */ }
```

⇒ în cel mai bun caz, o comportare aleatoare

Pointerii, ca orice variabile trebuie inițializați !

- cu adresa unei variabile (sau cu alt pointer inițializat deja)
- cu o adresă de memorie alocată dinamic (vom discuta ulterior)

EROARE: *tip \*p; \*p = valoare;*

- p este neinițializat (eventual nul, dacă e variabilă globală)
- ⇒ valoarea va fi scrisă la o adresă de memorie necunoscută (evtl. nulă)
- ⇒ coruperea memoriei, rezultate eronate sau imprevizibile, terminarea forțată a programului (sub sisteme de operare cu memorie protejată)

**ATENȚIE!:** un pointer nu este un întreg. Nu se recomandă conversia între pointer și int (presupune că `sizeof(void *) == sizeof(int)`)

## Pointeri ca argumente/rezultate de funcții

---

**Modificarea valorii unei variabile** prin transmiterea adresei ei

- o variabilă poate modificată prin indirectarea unui pointer către ea
- nu constituie excepție de la transmiterea parametrilor prin valoare (parametrul transmis e *adresa*, care nu se modifică)

```
void swap (int *pa, int *pb)
{
    int tmp;
    tmp = *pa; *pa = *pb; *pb = tmp;
}
```

Ex.: `int x = 3, y = 5; swap(&x, &y); /* acum x = 5 și y = 3 */`

Când **limbajul nu permite transmiterea prin valoare** (tablouri)  
sau ea ar fi ineficientă (structuri)

- funcțiile se scriu utilizând *adresa* variabilelor de tipul respectiv

*Tipărirea* valorii unui pointer: cu specificatorul `%p` în `printf`

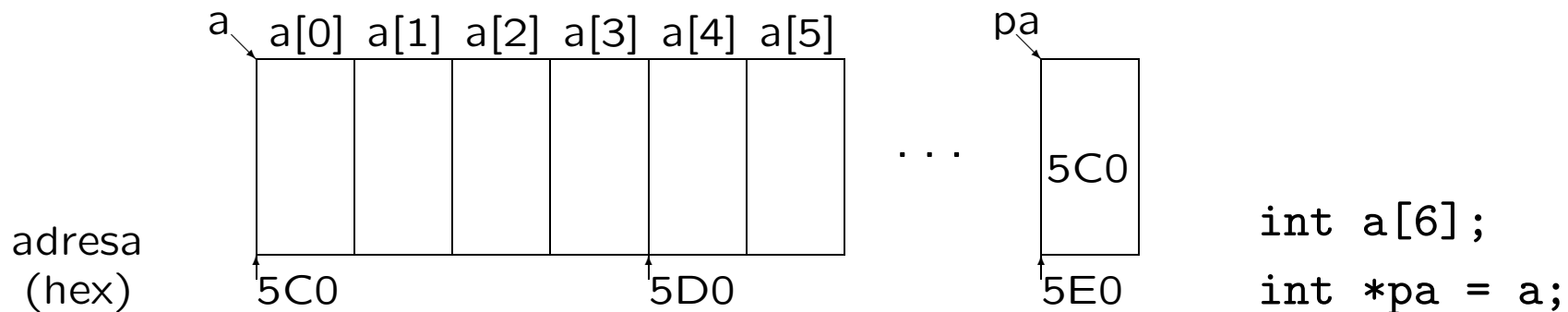
## Tablouri și pointeri

În limbajul C noțiunile de *pointer* și *nume de tablou* sunt asemănătoare.

- declararea unui tablou alocă un bloc de memorie pt. elementele sale
- *numele* tabloului e adresa blocului respectiv (= a primului element)

declarând `tip a[LEN], *pa;` putem atribui `pa = a;`  
`&a[0]` e echivalent cu `a` iar `a[0]` e echivalent cu `*a`

Diferența: adresa `a` e o *constantă* (tabloul e alocat la o adresă fixă)  
 $\Rightarrow$  nu putem atribui `a = adresă`, dar putem atribui `pa = adresă`  
`pa` e o *variabilă*  $\Rightarrow$  ocupă spațiu de memorie și are o adresă `&pa`



## Tablouri și pointeri (continuare)

---

În declarații de funcții, se pot folosi oricare din variante:

```
size_t strlen(char s[]);   sau   size_t strlen(char *s);
```

(de fapt, compilatorul convertește prima variantă în a doua)

⇒ nu se transmit tablouri (bloc de memorie) la funcții, ci adresele lor

Fie `char t[21];`    Compilatorul consideră `&t` ca fiind `t`

⇒ s-ar putea scrie și `scanf("%20s", &t)` în loc de `scanf("%20s", t)`

se recomandă totuși prima variantă, pentru uniformitate cu cazul:

```
char *p; p = t + 4; scanf("%16s", p)  /* e incorect &p ! */
```

Diferență între tablouri și pointeri:

```
sizeof t == 21*(sizeof char)   diferit de   sizeof p == sizeof(char *)
```

Atenție la tipuri!

Fie `char m[5][80]; char *p;` `p` și `m` nu au același tip, dar `p` și `m[2]` au !



O variabilă  $v$  de un anumit tip ocupă  $\text{sizeof}(\text{tip})$  octeți  
 $\Rightarrow \&v + 1$  reprezintă adresa la care s-ar putea memora următoarea variabilă de același tip (adresa cu  $\text{sizeof}(\text{tip})$  mai mare decât  $\&v$ ).

1. *Adunarea* unui întreg la un pointer: poate fi parcurs un tablou  
 $a + i$  e echivalent cu  $\&a[i]$  iar  $*(a + i)$  e echivalent cu  $a[i]$

```
char *endptr(char *s) { /* returnează pointer la sfârșitul lui s */
    char *p = s;        /* sau: char *p; p = s; */
    while (*p) p++;     /* adică la poziția marcată cu '\0' */
    return p;
}
```

2. *Diferența*: doar între doi pointeri *de același tip*  $\text{tip } *p, *q$ ;  
= numărul (trunchiat) de obiecte de tip care încap între cele 2 adrese  
– diferența numerică în octeți: se convertesc ambii pointeri la  $\text{char } *$   
$$p - q == ((\text{char } *)p - (\text{char } *)q) / \text{sizeof}(\text{tip})$$

Nu sunt definite nici un fel de alte operații aritmetice pentru pointeri !  
Se pot însă efectua operații logice de comparație ( $==, !=, <$ , etc.)

## Aplicații: funcții cu șiruri de caractere

---

– declarate în `string.h`; mai jos, exemple de implementări posibile

```
size_t strlen(const char *s) { /* lungimea șirului s */
    char *p = s;
    while (*p) p++;          /* până întâlnește '\0' */
    return p - s;           /* '\0' nu e numărat */
}

char *strcpy(char *dest, char *src) { /* copiază src în dest */
    char *p = dest;
    while (*p++ = *src++); /* copiază până întâlnește '\0' */
    return dest;          /* returnează dest prin convenție */
}

int strcmp (char *s1, char *s2) { /* compară caracter cu caract. */
    while (*s1 == *s2 && *s1) { s1++; s2++; } /* egale dar nu '\0' */
    return *s1 - *s2; /* < 0 pt. s1<s2, > 0 pt. s1>s2, 0 pt. egal */
}
```

## Alte funcții cu șiruri de caractere

---

```
char *strncpy(char *dest, char *src, size_t n) {
    char *p = dest; /* copiază cel mult n caractere */
    while (n-- && *p++ = *src++);
    return dest;
}

int strncmp (char *s1, char *s2, size_t n) { /*compară cel mult n*/
    if (n == 0) return 0;
    while (--n && *s1 == *s2 && *s1) { s1++; s2++; }
    return *s1 - *s2; /* < 0 pt. s1<s2, > 0 pt. s1>s2, 0 pt. egal */
}

char *strchr(char *s, int c) { /* prima poziție a lui c în s */
    do if (*s == c) return s; while (*s++);
    return NULL;          /* dacă nu a fost găsit */
}
```

NULL (0 cf. `stddef.h`) se folosește convențional ca adresă invalidă

## Pointeri și tablouri multidimensionale

---

Fie declarația `tip a[DIM1][DIM2];` Elementul `a[i][j]` este al `j`-lea element din tabloul de `DIM2` elemente `a[i]` și are adresa

$\&a[i][j] == (\text{tip } *) (a + i) + j == (\text{tip } *) a + \text{DIM2} * i + j$

⇒ pentru compilarea expresiei `a[i][j]` e necesară cunoașterea lui `DIM2`

⇒ în declarația unei funcții cu parametri tablou trebuie precizate toate dimensiunile în afară de prima (irelevantă): `void f(int m[][5]);`

### Pointeri și șiruri

Declarațiile `char s[] = "sir";` și `char *s = "sir";` sunt diferite!

– prima rezervă spațiu doar pt. șirul "sir", iar adresa `s` e o constantă

– a doua rezervă spațiu și pentru pointerul `s`, care poate fi reatribuit

`char s[12][4]={"ian",...,"dec"};` și `char *s[12]={"ian",...,"dec"};`

primul e un tablou 2-D de caractere, al doilea e un tablou de pointeri

## Argumentele liniei de comandă

---

Limbajul C permite accesul la parametrii argumentele) cu care programul e rulat din linia de comandă (ex. opțiuni, nume de fișiere) De asemenea, permite returnarea de program a unui cod întreg (folosit uzual pentru a semnala succes sau o condiție de eroare)

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    int i;

    printf("Numele programului: %s\n", argv[0]);
    if (argc == 1) printf("Program apelat fără parametri\n");
    else for (i = 1; i < argc; i++)
        printf("Parametrul %d: %s\n", i, argv[i]);
    return 0; /* codul returnat de program */
}
```

- argv[0] e numele programului, deci întotdeauna argc >= 1
- argv[1], etc.: parametrii, așa cum au fost separați de spații

## Pointeri la funcții

---

Adresa unei funcții se poate obține, memora, și utiliza pentru a o apela.

pentru o funcție `tip_rez fct (tip1, ..., tipn);`

adresa are tipul `tip_rez (*pfct) (tip1, ..., tipn);`

se poate atribui `pfct = fct;` (numele funcției reprezintă adresa ei)

Atenție la sintaxă:

`int *fct(void);` declară o funcție ce returnează pointer la întreg

`int (*fct)(void);` declară un pointer la o funcție ce returnează întreg

Exemplu de utilizare: parametrizarea unei alte funcții

Algoritmul *quicksort*, declarat (în `stdio.h`) ca funcție cu parametrii:

– adresa tabloului de sortat, numărul și dimensiunea elementelor

– adresa funcției care compară 2 elemente (returnează `<`, `=` sau `>` 0)

efectuarea comparării depinde de tip: întreg, șir, definit de utilizator

`void qsort(void *base, size_t num, size_t size, int (*compar)(void *, void *));`

– folosește argumente `void *` fiind compatibile cu pointeri la orice tip

## Utilizarea pointerilor la funcții (cont.)

---

- pentru tabele de rutine, apelate în funcție de un indice
- exemplu: meniu cu apelare de funcții în funcție de tasta apăsată

```
void help(void); void menu(void); /*...*/ void quit(void);  
void (*funtab)[10](void) = { help, menu, ...., quit };  
int getkey(void); /* citește tasta apăsată de utilizator */
```

```
void do_cmd(void)  
{  
    int k = getkey();  
    if (k >= 0 && k <= 9) funtab[k]();  
}
```

Sintaxa pointerilor de funcții e complicată  $\Rightarrow$  e util să declarăm un tip:

```
typedef void (*funptr)(void); /* pointer la funcție void */  
funptr funtab[10]; /* tabloul de pointeri de funcție */
```

## Alocarea dinamică

---

Până acum am atribuit la pointeri doar adrese de variabile *existente* și am declarat *static* doar variabile de dimensiuni cunoscute la compilare. Discutăm: funcții de gestiune *dinamică* a memoriei (`stdlib.h`): alocarea memoriei după necesități stabilite la *rularea* programului

```
void *malloc(size_t size); /* alocă size octeți */
void *calloc(size_t num, size_t size); /* num*size oct. init. 0 */
/* m/calloc returnează NULL la eroare (ex. mem. insuficientă) */
void *realloc(void *ptr, size_t size); /* modifică dimensiunea,
    poate muta blocul, dar păstrează conținutul memoriei */
void free(void *ptr); /* eliberează mem. alocată cu c/malloc */
```

```
int i, n, *t;
printf("Nr. de elemente ?"); scanf("%d", &n);
if ((t = malloc(n * sizeof(int))) != NULL)
    for (i = 0; i < n; i++) scanf("%d", &t[i]);
```



## Exemplu: citirea unei linii de dimensiune nelimitată

---

```
#include <stdio.h>
#include <stdlib.h>
const int BLOCK = 16;
char *getline(void) {
    char *p, *s = NULL;
    int c, lim = -1, size = 0; /* 1 loc pentru \0 */
    while ((c = getchar()) != EOF) {
        if (size >= lim) /* alocă memorie, testează de eroare */
            if (p = s, (s = realloc(s, (lim+=BLOCK)+1)) == NULL)
                return free(p), NULL;
        if ((s[size++] = c) == '\n') break;
    }
    s[size] = '\0'; return s;
}
```

## Exemple de alocare dinamică (cont.)

---

Să se citească un șir de numere, terminat cu zero și să se sorteze.

```
#include <stdio.h>
#include <stdlib.h>
#define NUM 100 /* alocăm pt. 100 de numere odată */
typedef int (*cmpptr)(const void *, const void *);
int cmp(int *p, int *q) { return *p - *q; } /* pt. sortare */
void main(void) {
    int i = 0, n = 0, *t = NULL; /* contor, total, tablou */
    do { /* alocă câte NUM întregi, inițial și când e nevoie */
        if (i == n) { n += NUM; /* realloc(NULL,sz) e ca malloc(sz) */
            if (!(t = realloc(t, n*(sizeof int)))) return 1; }
        if (scanf("%d", &t[i]) != 1) return 1; /* iese la eroare */
    } while (t[i++]); /* până când introducem zero */
    qsort(t, i, sizeof(int), (cmpptr)cmp); /* sortează */
    for (n = 0; n < i; n++) printf("%d ", t[n]);
    free(t);
}
```