# Pointeri

30 noiembrie 2004





## Variabile şi adrese

În limbajul C, orice variabilă are o *adresă*: o valoare numerică; indică locul din memorie unde e memorată valoarea variabilei

*Operatorul prefix* `&` dă adresa operandului: `&x` e adresa variabilei `x`
Operandul: orice poate folosit pe partea stângă a unei atribuiri (variabilă, element de tablou, funcţie; NU pentru expresii oarecare)

O adresă poate fi tipărită (în hexazecimal) cu formatul `%p` în `printf`

```
#include <stdio.h>
double d; int a[10]; /* variabile globale */
int main(void)
{
  int k;             /* variabilă locală */
  printf("Adresa lui d: %p\n", &d); /* de ex. 0x80496c0 */
  printf("Adresa lui a[0]: %p\n", &a[0]);  /* 0x80496e0 */
  printf("Adresa lui a[5]: %p\n", &a[5]);  /* 0x80496f4 */
  printf("Adresa lui k: %p\n", &k);        /* 0xbffff8e4 */
} /* Obs &a[5] - &a[0] == 5 * sizeof(int)  (poziţii consecutive) */
```





## Tipuri pointer. Declarare. Indirectare

Orice expresie în C are un tip ⇒ la fel şi expresiile adresă.
*OBS*: Dacă variabila `x` are tipul `tip`, `&x` are tipul `tip *`
`int x;` ⇒ `&x` are tipul `int *`, adică pointer la `int` (adresă de `int`)
`char c;` ⇒ `&c` are tipul `char *`, (pointer la `char`, adresă de `char`)
⇒ există tipuri de adresă diferite pentru fiecare tip de date
⇒ putem *declara* variabile de aceste tipuri (pointeri):

*tip* `*` *nume_var*; *nume_var* e pointer la (adresă pt.) o valoare de *tip*
*pointer* = o variabilă care conţine *adresa* altei variabile

*Operatorul prefix* `*` dă obiectul `*p` de la adresa dată de operandul `p`
Operand: pointer. Rezultat: *referinţă* la obiectul indicat de pointer
⇒ operator de *indirectare* (dereferenţiere, referire indirectă prin adresă)
*OBS*: Dacă pointerul `p` are tipul `tip *`, `*p` are tipul `tip`

Sintaxa declaraţiei (aceeaşi dar citită în două feluri) sugerează folosirea:
`char* p;` `p` e o variabilă de tipul `char *` (adresă de `char`)
`char *p;` `*p` (obiectul de la adresa `p`) are tipul `char`





## Operatorii de adresă şi dereferenţiere

Pointerii au adrese, ca orice variabile:
pt. `int *p;` adresa `&p` are tipul `int **`
`int ** pp = &p;` ⇒ `pp` are tipul `int **`, adică adresa unei adrese de `int`
dar putem citi `int* *pp` sau `int **pp`
deci `*pp` are tipul `int *` (adresă de `int`)
şi `**pp` are tipul `int` (val. de la adr. `*pp`)

| Variabilă | Valoare | Adresă |
|---|---|---|
| `int x=5;` | 5 | 0x408 |
| | ... | |
| `int *p=&x;` | 0x408 | 0x51C |
| | ... | |
| `int **pp=&p;` | 0x51C | 0x9D0 |

Înainte de folosire, un pointer trebuie *iniţializat*, de ex. cu adresa unei variabile de tipul potrivit: `int x, *p, **pp; p = &x; pp = &p;`

O *referinţă* `*p` poate fi folosită *la stânga sau la dreapta unei atribuiri* (în cazul de mai sus, `*p` se foloseşte absolut la fel (sinonim) cu `x`)
`int x, y, z, *p; p = &x; z = *p; /* z = x */ *p = y; /* x = y */`

*OBS*: Operatorii adresă `&` şi de indirectare `*` sunt *unul inversul celuilalt*:
`*&x` este chiar `x`, pentru orice obiect (variabilă) `x`
`&*p` are valoarea `p`, pentru orice variabilă pointer `p`
(dar `p` e o *variabilă* şi poate fi atribuită; `&*p` e o *expresie* şi nu poate)





## Eroarea cea mai frecventă: absenţa iniţializării

Utilizarea *oricărei* variabile neiniţializate e o eroare logică în program !
`{ int x; printf("%d", x); } /* cât e x ?? valoare la întâmplare! */`

*Pointerii trebuie iniţializaţi*, ca orice variabile !
– cu adresa unei variabile (sau cu alt pointer iniţializat deja)
– cu o adresă de memorie alocată dinamic (vom discuta ulterior)

*EROARE*: *tip* `*p; *p =` *valoare*; p este *neiniţializat*!! (eventual nul)
⇒ valoarea va fi scrisă la o adresă de memorie necunoscută (evtl. nulă)
⇒ coruperea memoriei, rezultare eronate sau imprevizibile, terminarea forţată a programului (sub sisteme de operare cu memorie protejată)

`NULL` definit în `stddef.h` ca `(void *)0`: nu e o adresă validă
⇒ folosit (la iniţializări, sau returnat) ca valoare de pointer invalid

*OBS*: pointerii au valori numerice, dar nu sunt acelaşi lucru ca întregii.
⇒ Nu convertiţi între pointer şi `int` (e dependent de implementare).

*OBS*: Un prim test al corectitudinii programului: *verificarea de tipuri*
Verificaţi că expresiile au tipuri corespunzătoare (ex. la atribuire)
⇒ valabil şi pentru pointeri (nu confundaţi `p` cu `*p`, etc.)





## Pointeri ca argumente/rezultate de funcţii

Permit modificarea valorii unei variabile prin transmiterea adresei ei
– o variabilă se poate modifica prin indirectarea unui pointer către ea
– nu se modifică *adresa* (transmisă tot prin valoare) ci *conţinutul* ei

```
void swap (int *pa, int *pb) /* schimbă val. de la adr. pa şi pb */
{
  int tmp; /* variabilă auxiliară necesară pentru interschimbare */
  tmp = *pa; *pa = *pb; *pb = tmp;  /* trei atribuiri de întregi */
} /* în funcţie s-a lucrat cu conţinutul de la adresele pa şi pb */
```

Ex.: `int x = 3, y = 5; swap(&x, &y); /* acum x = 5 şi y = 3 */`
*OBS*: Nu se poate obţine efectul cu `void swap(int m, int n);`
(ar schimba valorile transmise în corpul funcţiei, fără efect în afară)

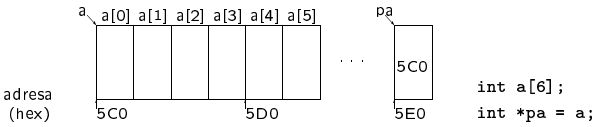
Folosire: când limbajul nu permite transmiterea prin valoare (*tablouri*) sau ar fi ineficientă (structuri mari) ⇒ transmitem *adresa* variabilei

## 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)
⇒ pentru tabloul `tip a[LEN];` numele `a` e o *constantă* de tipul `tip *`
`&a[0]` e echivalent cu `a` (adresa tabloului e adresa primului element)
`a[0]` e echivalent cu `*a` (obiectul de la adresa `a` e primul element)

Dacă declarăm `tip *pa;` putem atribui `pa = a;`

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




```
int a[6];
int *pa = a;
```





## 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)
`size_t`: tip pt. dimensiuni pozitive din `stddef.h` (ca și `unsigned` sau `unsigned long`)
⇒ 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 = s; scanf("%20s", p) /* aici e incorect &p ! */`

Diferență între tablouri și pointeri:
`sizeof t == 21*(sizeof char)` diferit de `sizeof p == sizeof(char *)`

Atenție! Verificați corespondența tipurilor în expresii.
Ex. `char m[5][80]; char *p;` `p` și `m` nu au același tip, dar `p[2]` și `m[2]` au !





## Aritmetica cu pointeri

O variabilă `v` de un anumit `tip` ocupă `sizeof(tip)` octeți
⇒ `&v + 1` reprezintă adresa la care s-ar putea memora următoarea variabilă de același tip (adresa cu `sizeof(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 sfârşitul lui s */
  char *p = s; /* sau: char *p; p = s; */
  while (*p) p++;       /* adică la poziţia marcată cu '\0' */
  return p;
}
```

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

```
p - q == ((char *)p - (char *)q) / sizeof(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 (`string.h`

```
size_t strlen(const char *s) { /* lungimea şirului s */
  char *p = s;
  while (*p) p++;     /* până întâlneşte '\0' */
  return p - s;       /* '\0' nu e numărat */
}
char *strcpy(char *dest, const char *src) { /* dest <- src */
  char *p = dest;
  while (*p++ = *src++);  /* copiază până întâlneşte '\0' */
  return dest;            /* returnează dest prin convenţie */
}
char *strcat(char *dest, const char *src) /* concat. src la dest */
{
  return strcpy(dest + strlen(dest), src);
}
int strcmp (const char *s1, const char *s2) { /* compară */
  while (*s1 == *s2 && *s1) { s1++; s2++; } /* egale dar nu '\0' */
  return *s1 - *s2; /* < 0 pt. s1<s2, > 0 pt. s1>s2, 0 pt. egal */
}
```





## Funcții cu șiruri de caractere (cont.)

```
char *strncpy(char *dest, const char *src, size_t n) {
  char *p = dest; /* copiază cel mult n caractere */
  while (n-- && *p++ = *src++);
  return dest;
}
int strncmp (const char *s1, const char *s2, size_t n) {
  if (n == 0) return 0;        /* compară pe lungime cel mult n */
  while (--n && *s1 == *s2 && *s1) { s1++; s2++; }
  return *s1 - *s2; /* < 0 pt. s1<s2, > 0 pt. s1>s2, 0 pt. egal */
}
char *strchr(const char *s, int c) { /* caută primul c în s */
  do if (*s == c) return s; while (*s++);
  return NULL;                 /* dacă nu a fost găsit */
}
void *memset(void *s, int c, size_t n); /* setează n octeţi cu c */
void *memcpy(void *dest, const void *src, size_t n);
void *memmove(void *dest, const void *src, size_t n);
/* copiază n octeţi; ultima variantă şi pentru zone suprapuse */
```





## 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] == (tip *)(a + i) + j == (tip *)a + 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 };

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

Sintaxa pointerilor de funcții e complicată ⇒ e util să declarăm un tip:

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





## 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 octeţi */
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 conţinutul 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]);
```





## 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, iniţial şi când e nevoie */
    if (i == n) { n += NUM; t = realloc(t, n*sizeof(int)); }
    scanf("%d", &t[i]); /* realloc(NULL,sz) e ca şi malloc(sz) */
  } while (t[i++]);
  qsort(t, i, sizeof(int), (cmpptr)cmp); /* sortează */
  for (n = 0; n < i; n++) printf("%d ", t[n]);
  free(t);
}
```