Programarea calculatoarelor

# Pointeri

Marius Minea

22 aprilie 2008





## Pointeri: recapitulare

O variabilă `x` de tipul *tip* are o *adresă* `&x` de tipul *tip* `*`
Variabila `x` ocupă `sizeof(x)` (sau: `sizeof(`*tip*`)`) octeți pornind de la `&x`
Adresele sunt nenule. Valoarea `NULL` (adresă 0) indică o adresă invalidă.

În *tip* `t[5];` numele `t` e *adresa* tabloului (elem. `[0]`) și are tipul *tip* `*`

Funcțiile au ca parametri nu conținutul tabloului, ci *adresa* tabloului.
`void f(`*tip* `t[8]);` e la fel ca `void f(`*tip* `t[])` și ca `void f(`*tip* `*t);`

Funcția care primește adresa unei variabile o poate *modifica* (și citi).
Ex: `scanf` (atribuie valori citite de la intrare), funcții cu tablouri
(modifică *conținutul* tabloului, dar nu *adresa*, transmisă prin *valoare*!)

O *constantă șir de caractere* `"sir"` are tipul `char *`
Valoarea constantei `"sir"` este *adresa* de memorie unde se află șirul.

*ATENȚIE* Nu putem compara un `char` (`'a'`) cu un șir (adresă) `"a"` !
Comparăm șiruri cu `str(n)cmp`, nu cu `==` (compară *adrese*, nu conținut)

*Pointerii sunt variabile normale*: au tip, valoare, loc în memorie, adresă, pot fi declarați, atribuiți, tipăriți, dați parametri, au operații specifice.





## 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` *tip*

Operatorul adresă `&` operator prefix
– operand: o variabilă (ex. `x`); rezultat: *adresa* variabilei `&x`
– folosit doar pt. *variabile* (și elem. tablou), nu constante, expresii, etc.
– se poate atribui unui pointer la acel tip: `int x; int *p; p = &x;`

Operatorul de dereferențiere (indirectare) `*` operator prefix
– operand: pointer; rezultat: *obiectul* (variabila) indicat de pointer
– `*p` e un *lvalue*, poate fi folosit la stânga unei atribuiri, ca și variabilele sau elem. tablou; (orice *expresie* poate fi la dreapta lui `=`)

– dacă `p` e `&x`, atunci `*p` e obiectul de la adresa `p` (a lui `x`), deci `x`
`int x, y, *p; p = &x; y = *p; /* y = x */ *p = y; // x = y`

Operatorul `*` e *inversul* lui `&`: `*&x` e chiar `x` (obiectul de la adresa lui x)
`&*p` e `p` (p: pointer cu valoare validă): adresa obiectului de la adresa `p`





## Declarații și referințe: observații

Putem citi declarația *tip* `* p;`
*tip* `*` `p;` `p` are tipul *tip* `*`
*tip* `*p;` `*p` e un caracter
`char **s; // adresă de adr.de char`
`char *t[8]; // tab.de 8 adr.de char`

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

*ATENȚIE* O *declarație* cu *inițializare* NU este o *atribuire* !
`int x = 5;` e la fel cu `int x; x = 5;`
`int t[2] = { 3, 5 };` dar ~~`t[2] = {3, 5 }`~~ NU are sens!
`int x, *p = &x;` este `int x; int *p = &x;` sau `int x; int *p; p = &x;`
(e inițializat/atribuit `p`, NU `*p`). ~~`p = x`~~ e incorect ca tip!
`char *p = "sir";` e `char *p; p = "sir";` dar ~~`*p = "sir";`~~ e greșit!

`*` 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 !





## Eroarea cea mai frecventă: absența inițializării

Folosirea *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 =` *ceva*`;` *EROARE*: `char *p; scanf("%s", p);`
– p este *neinițializat* (eventual nul, dacă e variabilă globală)
⇒ valoarea va fi scrisă la o *adresă de memorie necunoscută* (evtl. nulă)
⇒ memorie coruptă, vulnerabilități de securitate, rulare abandonată

ATENȚIE: un pointer nu este un întreg. Greșit: ~~`int *p = 640;`~~ !
Doar compilatorul/sistemul de operare poate alege adresele, nu noi!





## Pointeri ca argumente/rezultate de funcții

Având adresa `p` a unei variabile îi putem *modifica valoarea*: `*p = ...`
funcția care primește adresa unei variabile poate modifica valoarea ei
ex. `scanf` primește *adrese*, completează *conținutul* cu valorile citite
dar parametrii sunt transmiși *tot prin valoare*: adresa nu se modifică

```
void swap (int *pa, int *pb) {  // schimba valorile de la 2 adrese
  int tmp;  // variabila temporara pentru valoarea schimbata prima
  tmp = *pa; *pa = *pb; *pb = tmp;   // trei atribuiri de intregi
}
```

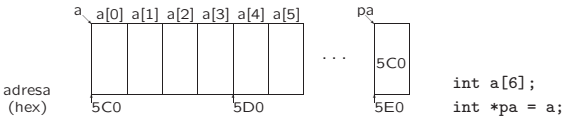
Ex.: `int x = 3, y = 5; swap(&x, &y); // acum x = 5 și y = 3`

Folosim:
– când limbajul ne obligă (tablouri ca parametri la funcții)
– pentru a întoarce mai multe rezultate (funcția permite doar unul)
ex. minimul *și* maximul unui tablou; rezultat *și* cod de eroare

## Tablouri şi pointeri

În limbajul C noţiunile de *pointer* şi *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`

Diferenţa: 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 spaţiu de memorie şi are o adresă `&pa`




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





## Tablouri şi pointeri (continuare)

În declaraţii de funcţii, se pot folosi oricare din variante:
`size_t strlen(char s[]);` sau `size_t strlen(char *s);`

*ATENŢIE* la diferenţe!
`char s[] = "test";` `s[0]` e `'t'`, `s[4]` e `'\0'` etc.
`s` e o *adresă constantă* de tip `char *`, nu variabilă cu loc în memorie
NU se poate atribui `s = ...`, se poate atribui `s[0] = 'f'`
`sizeof(s)` e 5 `* sizeof(char)` `&s` e chiar `s`
(dar are alt tip, adresă de tablou de 5 char: `char (*)[5]`)

`char *p = "test";` la fel: `p[0]` e `'t'`, `p[4]` e `'\0'` etc.
`p` e o *variabilă de tip adresă* (`char *`), ocupă loc în memorie
NU se poate atribui `p[0] = 'f'` ("test" e o constantă şir),
se poate atribui `p = "ana";` sau `p = s;` şi apoi `p[0] = 'f'`
`sizeof(p)` e `sizeof(char *)` `&p` NU e `p`
⇒ e GREŞIT: `scanf("%4s", &p);` CORECT: `scanf("%4s", p);`





## Aritmetica cu pointeri

O variabilă `v` de un anumit `tip` ocupă `sizeof(tip)` octeţi
⇒ `&v + 1` reprezintă adresa la care s-ar putea memora următoarea variabilă de acelaşi 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. *Diferenţa*: doar între doi pointeri *de acelaşi tip* `tip *p, *q;`
= numărul (trunchiat) de obiecte de `tip` care încap între cele 2 adrese
– diferenţa numerică în octeţi: se convertesc ambii pointeri la `char *`
`p - q == ((char *)p - (char *)q) / sizeof(tip)`

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





## Pointeri şi indici

Termenul "pointer" provine de la "to point (to)" (a indica)
CĂnd identificăm un element de tablou `a[i]` folosim doua variabile: tabloul şi indicele, şi implicit o adunare (indicele la adresa de bază)
Mai simplu: folosind direct un pointer la adresa elementului `&a[i]==a+i`
⇒ la parcurgere, în loc să avansăm indicele, incrementăm pointerul

```
char *strchr_i(const char *s, int c) {  // caută caracter în şir
  for (int i = 0; s[i]; ++i) // parcurge s cu indice i până la '\0'
    if (s[i] == c) return &s[i]; // s-a găsit: returnează adreasa
  return NULL; // nu s-a găsit: returnează NULL (adresă invalidă)
}

char *strchr_p(const char *s, int c) {  // scrisă folosind pointer
  for ( ;*s; ++s)     // folosim chiar parametrul pentru parcurgere
    if (*s == c) return s;        // s indică caracterul curent
  return NULL;        // nu s-a găsit
}
```





## Pointeri şi tablouri multidimensionale

Fie un tablou bidimensional (matrice) declarat *tip* `a[DIM1][DIM2];`
`a[i]` e adresa (constantă *tip* `*`) a unui tablou (linii) de `DIM2` elemente
`a[i][j]` e al j-lea element din tabloul de `DIM2` elemente `a[i]` ; adresa `&a[i][j] == a[i]+j` e cu `DIM2*i+j` elemente după adresa tabloului `a`
⇒ o funcţie cu parametri tablou trebuie să cunoască toate dimensiunile în afară de prima ⇒ trebuie declarată *tip-f* `f(`*tip-t* `t[][DIM2]);`

`char t[12][4]={"ian",...,"dec"};` şi `char *p[12]={"ian",...,"dec"};`

`t` e un tablou 2-D de caractere

| | | | |
|---|---|---|---|
| i | a | n | \0 |
| f | e | b | \0 |
| ... | | | |
| d | e | c | \0 |

`t` ocupă 12 * 4 octeţi

`t[6] = ...` e GREŞIT
(`t[6]` e adresa constantă a liniei 7)

`p` e un tablou de pointeri

| | | | | | |
|---|---|---|---|---|---|
| 0x460 | ⟶ | i | a | n | \0 |
| 0x5C4 | ⟶ | f | e | b | \0 |
| ... | | | | | |
| 0x9FC | ⟶ | d | e | c | \0 |

`p` ocupă 12*sizeof(char *) octeţi
(+ 12*4 octeţi pt. *constantele* şir)

`p[6]="iulie"` modifică o adresă
(elementul 7 din tabloul de adrese `p`)





## Argumentele liniei de comandă

Pe linia de comandă, după numele programului rulat, pot urma argumente (parametri): opţiuni, nume de fişiere ... Exemple:
`gcc -Wall -o prog prog.c` `ls` *director* `cp` *fisier1 fisier2*
În C, avem acces la linia de comandă declarând `main` cu 2 parametri:
`int argc` : nr. de cuvinte din linia de comandă (nr. argumente + 1)
`char *argv[]` : tablou cu adresele argumentelor (şiruri de caractere)

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

`argv[0]` (primul cuvânt) e numele programului, deci sigur `argc >= 1`
tabloul `argv[]` e încheiat cu un element `NULL` (`argv[argc]`)