

Limbaje de programare

## Pointeri. Alocare dinamică

21 noiembrie 2011

## Un pointer e o adresă (recapitulare)

O variabilă `x` de tipul `tip` are o *adresă* `&x` de tipul `tip *`

Adresele sunt nenule. Valoarea `NULL` (0) indică o adresă invalidă.

Variabila `x` ocupă `sizeof(x)` octeți pornind de la `&x`.

### *Adrese și tablouri*

Numele `t` al tabloului `tip t[5]`; e *adresa* tabloului (a primului element, `&t[0]`). Adresa `t` are tipul `tip *`

Funcțiile au ca parametri *adresa* tabloului, NU conținutul său  
`void f(typ t[8]);` e ca `void f(typ t[]);` și ca `void f(typ *t);`

Primind adresa unei variabile, funcția o poate citi și *modifica*.

Ex: `scanf` (atribuie valori citite de la intrare), funcții cu tablouri (pot modifica *conținutul*, dar nu *adresa*, transmisă prin *valoare* !)

# Folosirea pointerilor (recapitulare)

## *Șiruri de caractere*

O *constantă șir de caractere* "sir" are tipul `char *`

Valoarea constantei "sir" este *adresa* unde se află șirul.

**ATENȚIE!** un `char ('a')` e diferit de un șir (adresă) "a" !

Comparăm șiruri cu `strcmp`, `strncmp`, NU cu `==`

Operatorul `==` compară *adrese* (UNDE se află șirul),

dar putem avea șiruri egale chiar dacă sunt la adrese diferite!

## *Variabile pointer*

*Pointerii se folosesc ca și orice alte variabile*

au tip, valoare, loc în memorie, adresă

pot fi declarați, atribuiți, tipăriți, dați parametri

au operații specifice (dereferențiere `*`, aritmetică `+` `-` `++` `--`)



## Ce valoare se află la o adresă?

Operatorul de dereferențiere (indirectare) \* operator prefix  
operand: pointer; rezultat: *obiectul* (variabila) indicat de pointer

\*p poate fi folosit la stânga unei atribuirii (engl. *lvalue*),  
ca și variabilele sau elementele de tablou;

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 (dacă p e pointer valid): adresa obiectului de la adresa p

## Declarație și dereferențiere

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
<code>int x = 5;</code>	5	0x408
	...	
<code>int *p=&amp;x;</code>	0x408	0x51C
	...	
<code>int **pp=&amp;p;</code>	0x51C	0x9D0

**ATENȚIE** O **declarație** cu **inițializare** NU este o **atribuire** !

`int t[2] = { 3, 5 };` inițializează t. Incorect: ~~`t[2] = { 3, 5 };`~~

`int x, *p = &x;` e la fel ca `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";` Greșit: ~~`*p = "sir;"`~~

## EROARE: lipsa inițializării

E o *EROARE* să folosim o *variabilă neinițializată*

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

⇒ programul începe calculul cu o valoare *la întâmplare !!*

*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:* `int *p; *p = 0;` *EROARE:* `char *p; scanf("%s", p);`

p este *neinițializat* (eventual nul, dacă e variabilă globală)

⇒ valoarea e scrisă la o *adresă de memorie necunoscută*

⇒ memorie coruptă, vulnerabilități de securitate, terminare forțată

ATENȚIE: un pointer nu este un întreg. Greșit: `int *p = 640; !`

NU putem alege adresa unei variabile (unde să e pusă în memorie)

⇒ se determină la încărcarea programului / când se alocă memoria

## Pointeri ca argumente/rezultate de funcții

Având adresa unei variabile îi putem *modifica valoarea*: \*p = ...  
primind adresa unei variabile, o funcție poate modifica valoarea ei  
scanf primește *adrese*, completează *conținutul* cu valori citite  
parametrii sunt transmiși *tot prin valoare*: adresa nu se modifică

```
void swap (int *pa, int *pb) { // schimba valori de la 2 adrese  
    int tmp; // var. temporara pentru valoarea schimbata prima  
    tmp = *pa; *pa = *pb; *pb = tmp; // atribuirii de intregi  
}
```

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

Folosim adrese ca parametri de funcții:

ca să transmitem tablouri (altfel nu se poate)

pentru a întoarce mai multe rezultate (funcția permite doar unul)

ex. minimul și maximul unui tablou; rezultat și cod de eroare



# Tablouri și pointeri

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

declararea unui tablou alocă un bloc de memorie pt. elemente

*numele* tabloului e *adresa* blocului de mem. (a primului element)

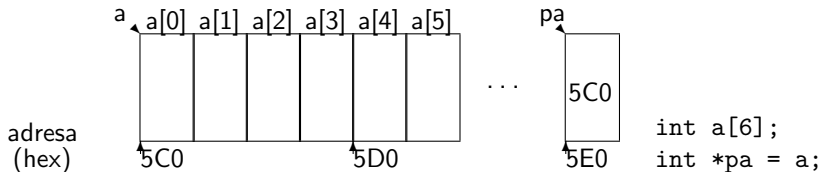
Dacă declarăm `tip a[LEN], *pa;` putem atribui `pa = a;`

`&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)

⇒ *nu putem atribui* `a = adresă`, dar putem atribui `pa = adresă`

`pa` e o *variabilă* ⇒ ocupă spațiu de memorie și are o adresă `&pa`



## Tablouri și pointeri (continuare)

Ca parametri la funcții, cele două scrieri înseamnă *același lucru*  
`size_t strlen(char s[]);` sau `size_t strlen(char *s);`

Ca declarații, de tablou / pointer, *sunt diferite!*

*Tablou:* `char s[] = "test";`      `s[0]` e 't', `s[4]` e '\0' etc.  
`s` e *adresă constantă* (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]`)

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

## Aritmetica cu pointeri

O variabilă  $v$  de un anumit tip ocupă  $\text{sizeof}(\text{tip})$  octeți  
 $\Rightarrow \&v + 1$  reprezintă adresa la care s-ar găsi 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  
 $a + i$  înseamnă  $\&a[i]$  iar  $*(a + i)$  înseamnă  $a[i]$

```
char *endptr(char *s) { // ret. 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* tip \*p, \*q;  
= numărul întreg de obiecte de tip care încap între cele 2 adrese

Diferența numerică în octeți: se convertesc pointerii la char \*

$$p - q == ((\text{char } *)p - (\text{char } *)q) / \text{sizeof}(\text{tip})$$

Nu sunt definite alte operații aritmetice pentru pointeri !

Se pot efectua operații logice de comparație (==, !=, <, etc.)

## Pointeri și indici

Termenul “pointer” provine de la “to point (to)” (a indica)

Când identificăm un element de tablou `a[i]` folosim doua variabile: tabloul și indicele, și implicit o adunare (indicele la adresa de bază)

Mai simplu: direct cu 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 șir
    for (int i = 0; s[i]; ++i) // parcurge cu indice i până la '\0'
        if (s[i] == c) return &s[i]; // s-a găsit: returnează adr.
    return NULL; // nu s-a găsit: returnează NULL
}
```

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

## Pointeri și tablouri multidimensionale

Un tablou bidimensional (matrice) se declară `tip a[DIM1][DIM2];`  
`a[i]` e adresa (constantă `tip *`) a unui tablou (linii) de DIM2 elem.  
`a[i][j]` e al j-lea element din tabloul de DIM2 elemente `a[i]`

`&a[i][j]` sau `a[i]+j` e cu DIM2\*i+j elem. după adresa a

⇒ o funcție cu tablou are nevoie de toate dimensiunile

în afară de prima ⇒ trebuie declarată `tip-f f(tip-t t[][DIM2]);`

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

`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 \*) octeți

(+ 12\*4 octeți pt. *constantele șir*)

`t[6] = ...` e GREȘIT

`p[6]="iulie"` modifică o adresă

(`t[6]` e adresa constantă a liniei 7) (elementul 7 din tabloul de adrese `p`)

## Argumentele liniei de comandă

Linia de comandă conține *numele programului* urmat de eventuale *argumente* (parametri): opțiuni, nume de fișiere ... Exemple:

```
gcc -Wall -o prog prog.c      ls director      cp fisier1 fisier2
```

În C, accesăm linia de comandă declarând `main` cu 2 parametri:

`int argc` nr. cuvinte din linia de comandă (nr. argumente + 1)  
`char *argv[]` tablou, adresele argumentelor (șiruri 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 `argc >= 1`  
tabloul `argv[]` e încheiat cu un element `NULL` (`argv[argc]`)

## Funcții de citire/scriere formatată în șiruri

Variante de printf/scanf cu sursă/destinație și șiruri de char.

```
int sprintf(char *s, const char *format, ...);
```

```
int sscanf(const char *s, const char *format, ...);
```

sprintf nu are limitare  $\Rightarrow$  poate depăși mărimea tabloului. Folosiți

```
int snprintf(char *str, size_t size, const char *format, ...);
```

în care scrierea e limitată la size caractere  $\Rightarrow$  variantă sigură

*Conversia din șir în număr* int n; char \*s, \*end;

```
if (sscanf(s, "%d", &n) == 1) ... (citire corectă)
```

(folosim când nu trebuie prelucrat restul șirului după număr)

strtol: atribuie adresa primului caracter rămas, putem prelucra restul

```
n = strtol(s, &end, 10);
```

 (în baza 10, sau în altă bază)

```
n = atoi(s);
```

 (returnează 0 la eroare, dar și la șirul "0")

## De ce e necesară alocarea dinamică

Până acum am indicat prin pointeri doar variabile deja declarate:

```
int x; int *p; p = &x; char a[20]; char *s; s = a+5;
```

Am declarat *static* doar tablouri de dimensiuni cunoscute și fixe  
(în C99 se permit dimensiuni variabile, evaluate la rulare)

Nu putem returna un tablou declarat într-o funcție. L-am declarat în afara funcției, am transmis adresa la funcția care-l completează.  
(scanf, strcpy, strcat, exemplele de lucru cu vectori/matrici)

Funcțiile de *alocare dinamică* (stdlib.h) permit să creem variabile noi de dimensiuni necesare apărute la *rularea* programului

**ATENȚIE!** *tip* \*p declară doar o variabilă *adresă*,  
NU și loc pentru un *obiect* (variabilă) de acel *tip*.

Declarația char \*s; NU alocă memorie pentru conținutul șirului!



## Funcții de alocare dinamică (stdlib.h)

`void *malloc(size_t size);`                      alocă size octeți  
`void *calloc(size_t n, size_t size);`      n\*size octeți init. cu 0

Returnează adresa blocului de memorie alocată sau NULL la eroare  
(mem. insuficientă) ⇒ *trebuie testat rezultatul!*

*modificarea dimensiunii* unei zone alocate cu malloc/calloc:

`void *realloc(void *ptr, size_t size);` modifică mărimea la size  
Poate muta conținutul existent și returna altă adresă decât ptr  
`if (p1 = realloc(p, size)) { p = p1; /* apoi folosim p */ }`

Memoria alocată dinamic *trebuie eliberată* când nu mai e necesară  
`void free(void *ptr);`                      eliberează memoria alocată cu c/malloc

```
int *t; unsigned i, n;
printf("Nr. de elemente ?");
if (scanf("%u", &n) == 1)
    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>
#define BLOCK 16
char *getline(void) {
    char *p, *s = NULL; // s initializat pentru realloc
    int c, lim = -1, size = 0; // pastram un loc pentru \0
    while ((c = getchar()) != EOF) {
        if (size >= lim) // s-a umplut zona alocata
            if (!(p = realloc(s, (lim+=BLOCK)+1))) { // mai aloca 16
                ungetc(c, stdin); break; // termina daca nu mai e loc
            } else s = p; // tine minte noua adresa alocata
        s[size++] = c; // adauga ultimul caracter
        if (c == '\n') break; // iese la linie noua
    } // termina cu \0, realoca doar cat e nevoie
    if (s) { s[size++] = '\0'; s = realloc(s, size); }
    return s;
}
```

## Când și cum folosim alocarea dinamică

**NU** e necesară când știm dinainte de câtă memorie e nevoie

```
NU: int *px; px = malloc(sizeof(int)); scanf("%d", px);
```

```
Mai simplu: int x; scanf("%d", &x);
```

**DA**, când nu știm de la compilare câtă memorie e necesară (tablouri cu dimensiuni aflate la rulare, liste, arbori, etc.)

**DA**, când trebuie să returnăm un obiect nou creat dintr-o funcție (NU putem returna adresa var. locale, memoria dispare la revenire!)

```
char *strdup(const char *s) {          // creeaza copie a lui s
    char *d = malloc(strlen(s) + 1); // loc pentru sir si '\0'
    return d ? strcpy(d, s) : NULL;    // fa copia, returneaza d
}
```

**DA**, când trebuie păstrat un obiect citit într-un loc temporar

```
char *tab[10], buf[81];
while (i < 10 && fgets(buf, 81, stdin))
    tab[i++] = strdup(buf); // salveaza adresa copiei
```

## Pointeri la funcții

*Parametrii, variabilele* permit calcule flexibile (nu doar cu date fixe)

Uneori dorim să variem *funcția* apelată într-un punct de program.

Exemplu: parcurgerea unui tablou pentru diverse prelucrări:

```
for (int i = 0; i < len; ++i) f(tab[i]); (cu diverse funcții f)
```

⇒ folosim variabile *pointeri la funcții*

*Numele* unei funcții reprezintă *adresa* funcției. Vezi declarațiile:

de *funcție*: `tip_rez fct (tip1, ..., tipn);`

de *pointer la funcție*: `tip_rez (*pfct) (tip1, ..., tipn);`

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

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

Putem atribui `pfct = fct;` (numele funcției e adresa ei)

*DAR*: `int *fct(void);` e o funcție ce returnează *pointer la întreg*

Declarând un tip pointer, e ușor să declarăm variabile de acest tip:

`typedef void (*funptr)(void);` (tip pointer la funcție void)

`funptr funtab[10];` (tablou de pointeri de funcție void)

## Folosirea pointerilor la funcții

```
void add3(int *p) { *p += 3; }
void tip(int *p) { printf("%d ", *p); }
void prel(int tab[], int len, void (*fp)(int *p)) {
    for (int i = 0; i < len; ++i) fp(&tab[i]);
} // apoi in main putem scrie:
int t[LEN] = { 2, 3, 5, 7, 11 }; // tabloul de prelucrat
prel(t, LEN, add3); prel(t, LEN, tip); // adună, scrie
```

Exemplu: funcția standard de sortare `qsort` (`stdlib.h`)

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

adresa tabloului de sortat, numărul și dimensiunea elementelor  
adresa funcției de comparat elemente (returnează <, = sau > 0)

⇒ are argumente void \*, compatibile cu pointeri la orice tip

```
typedef int (*comp_t)(const void *, const void *); //tip ptr.fct
int intcmp(int *p1, int *p2) { return *p1 - *p2; }
int tab[5] = { -6, 3, 2, -4, 0 }; // tabloul de sortat
qsort(tab, 5, sizeof(int), (comp_t)intcmp); // sort.crescator
```