

# Limbaje de Programare

## Curs 7 – Pointeri. Alocare dinamică de memorie. Argumentele liniei de comandă

Dr. Casandra Holotescu

Universitatea Politehnica Timișoara

# Ce discutăm azi...

- ① Argumentele liniei de comandă
- ② Tipul pointer
- ③ Erori în lucru cu pointeri
- ④ Pointeri ca argumente/rezultate la funcții
- ⑤ Tablouri și pointeri
- ⑥ Aritmetica pointerilor
- ⑦ Alocare dinamică de memorie

## Argumentele funcției *main*

Un program poate fi executat **cu 0 sau mai mulți parametri**.

În UNIX, acești parametri se dău ca argumente în linia de comandă:

`./program arg1 arg2 arg3 .. argn`

Argumentele liniei de comandă pot fi accesate în funcția main, ca **șiruri de caractere**.

Astfel, *main* va avea 2 parametri:

`int argc – numărul de parametri` cu care s-a apelat prog. +1  
`char *argv[ ] – tabloul de șiruri` care conține parametrii

`int main(int argc, char *argv []) { .. }`

Obs: `argv[0]` conține întotdeauna **numele executabilului**.

Parametrii propriu-zisi încep de la `argv[1]` încolo.

## Accesarea argumentelor liniei de comandă

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i;
    printf(" Numele programului: %s\n", argv[0]);

    if (argc==1)
        printf(" Nu are parametri\n");
    else
        for (i = 1; i < argc; i++)
            printf(" Parametrul %d: %s\n", i, argv[i]);

    return 0; /* codul returnat de program */
}
```

## Ne amintim: Adrese

Orice variabilă are o **adresă**, la care se memorează valoarea ei.

Operatorul prefix **&** dă adresa operandului:

**&x** e adresa variabilei x.

& se poate folosi cu orice **lvalue** (=destinație validă de atribuiri):  
**variabile, elemente de tablou.**

Nu se poate folosi cu: expresii, constante (nu au adresă!).

În cazul tablourilor, numele unui tablou este chiar adresa sa.

Exemplu: int z[4];

Numele z reprezintă **adresa de început** a tabloului (nu toate elementele sale, împreună, doar adresa sa în memorie!).

## Ne amintim: Adrese

O adresă poate fi tipărită (în hexazecimal – baza 16) folosind formatul `%p` în apelul funcției `printf`.

```
#include <stdio.h>
int main(void) {
    double d; int a[6];

    printf("Adresa lui d: %p\n", &d );
    // folosim operatorul &

    printf("Adresa lui a: %p\n", a );
    // a e deja adresa, nu e nevoie de &

    return 0;
}
```

## Ne amintim: Tipul pointer

Orice variabilă x are o **adresă**, &x, unde se memorează valoarea ei.

- fiind o expresie, &x are un tip: tipul **pointer** (adresă)

Pentru o variabilă declarată:

**nume-tip x;**

adresa & x are tipul **nume-tip \***

= pointer la nume-tip, adresa unui obiect de tip nume-tip

Funcția care primește **adresa** unei variabile îi poate **citi și modifica valoarea memorată**.

## Ne amintim: Tipul pointer

Numele unui tablou are tipul pointer la tipul elementului:

*int a[4];*  $\Rightarrow$  *a* are tipul *int \**

*char s[8];*  $\Rightarrow$  *s* are tipul *char \**

La declararea parametrilor funcției:

*void f(tip a[ ])* înseamnă de fapt

*void f(tip \*a)*

(de aceea dimensiunea: *void f(tip a[6])* nu contează)

Tipul unei constante sir de caractere "sir" este *char \**  
= adresa unde se găsește sirul în memorie

Valoarea specială **NULL** (0 de tip void \* = adresă de tip neprecizat) e folosit pentru a indica o **adresă invalidă**.

## Tipul pointer

Pointer = o variabilă care conține **adresa** altei variabile

Declararea unei variabile de tip pointer:

**nume-tip \*nume-variabila**

= *nume-variabila* e **pointer la o valoare** de tip *nume-tip*  
= *nume-variabila* **conține adresa unei valori** de tip *nume-tip*

```
int x;      // variabila de tip int
int *p;      // variabila pointer la int

// lui p i se atribuie adresa lui x
p = &x;
```

## Dereferențierea unui pointer

### Operatorul de derefențiere: \*

- operator prefix: **\*nume-pointer**
- operand: pointer
- rezultat: **valoarea** de la adresa conținută de pointer

Obs: dacă p e un pointer, **\*p** e un **lvalue** – i se pot atribui valori

Operatorul \* e inversul lui &:

**\*&x** e chiar x (valoarea de la adresa lui x)

**&\*p** e p (p: pointer valid): adresa valorii de la adresa p

```
int x, y, z, *p;  
p = &x; // p contine adresa lui x  
y = *p; // echivalent cu y = x  
*p = z; // echivalent x = z
```

## Erori frecvente

Eroarea clasică: **lipsa inițializării**

Folosirea unei **variabile neinitializate** este în totdeauna o **eroare!**

**Pointerii trebuie inițializați!** (ca orice variabilă)

cu adresa unei variabile

cu alt pointer, deja inițializat/care conține o adresă validă

cu adresa unei zone de memorie alocate dinamic

```
int *p; //p neinitializat, adresa invalida  
*p = 10; //GRESIT!!!
```

```
char *s; //s neinitializat, adresa invalida  
scanf("%20s", s); //GRESIT!!!  
//=> memorie corupta, vulnerabilitati
```

# Pointeri și funcții

Având adresa unei variabile îi putem modifica valoarea.

⇒ o funcție care primește ca parametru **adresa** unei variabile poate să îi **modifice** valoarea

Obs: În C toți **parametrii se transmit prin valoare**, în cazul parametrilor de tip **pointer** valoarea transmisă este o **adresă** (adresa nu se modifică!)

Folosim parametri de tip pointer:

când ne obligă limbajul (tablouri ca parametri la funcții) pentru a întoarce mai multe rezultate (funcția – un rez.)

## Pointeri și funcții – exemplu

```
// schimba valorile de la 2 adrese
void swap (int *pa, int *pb) {
    int tmp; // variabila temporara

    tmp = *pa;
    *pa = *pb;
    *pb = tmp;
    // trei atribuirile de intregi
}
... //in main
int x = 3, y = 5;
swap(&x, &y); //apelul functiei
// acum x = 5 si y = 3
```

## Tablouri și pointeri

În C, noțiunile de pointer și tablou sunt asemănătoare:

declararea unui tablou alocă o zonă de memorie pt elementele tabloului ⇒ **numele tabloului e adresa de început** a zonei

Diferența: un tablou e alocat la o **adresă fixă**, constantă.

un pointer e o **variabilă**, ocupă spațiu în memorie, are adresă

**tip a[LEN], \*pa;**

**pa = a;** //atribuim o adresa constantă unei variabile **tip\***

**&a[0]** e echivalent cu **a**

**a[0]** e echivalent cu **\*a**

Nu putem atribui o adresă lui **a** (e constantă), dar putem atribui o adresă lui **pa** (e variabilă).

## Tablouri și pointeri

În declarații de funcții se poate folosi orice variantă:

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

Atenție la diferențe:

```
char s [] = "test"; // adresa constanta (char *)  
char *p = "test"; // variabila de tip char *
```

s e *tablou*: &s == s, sizeof(s) == 5 \* sizeof(char)  
putem avea s[0]='a', s[2]='b', etc.

p e *pointer*: &p != p, sizeof(p) == sizeof(char \*)  
nu putem avea p[0]='a' ("test" e sir constant)  
putem avea p = "altceva" (tot un sir constant)

## Aritmetica pointerilor: adunarea

O variabilă  $v$  de un anumit tip ocupă  $\text{sizeof}(tip)$  octeți/bytes.

$\&v$  – adresa lui  $v$

$\&v + 1$  – adresa la care s-ar putea memora **următoarea** variabilă **de același tip** (adresa cu  $\text{sizeof}(tip)$  mai mare decât  $\&v$ )

Adunarea unui întreg la un pointer:

- se obține tot o adresă de același tip.

Se poate folosi pentru parcurgerea de tablouri:

$a + i$  e echivalent cu  $\&a[i]$

$*(a + i)$  e echivalent cu  $a[i]$

## Aritmetica pointerilor: adunarea

```
char *endptr(char *s) {
    // returneaza pointer la sfarsitul sirului s

    char *p = s;
    // echivalent cu: char *p; p = s;

    while (*p)
        p++;

    // adica pana ajunge la pozitia
    // din sir marcata cu '\0' (*p == 0)
    // adresa memorata in p creste cu 1

    return p;
}
```

## Aritmetica pointerilor: diferență

Se poate calcula doar **diferența între doi pointeri de același tip**  
**tip `*p, *q;`**

$p - q$  = nr. de valori de tip care încap între cele 2 adrese

Pentru a obține diferența numerică **în octeți**:

se **convertesc** ambii pointeri la **char \***

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

**Nu sunt definite alte operații aritmetice pentru pointeri !**

Dar se pot efectua operații logice de **comparație**  
(cu operatorii relaționali `==`, `!=`, `<`, etc.)

## Exemplu: parcurgere cu pointeri vs. indici

```
char *strchr_i(const char *s, int c) {
    // cauta un caracter in sir

    for (int i = 0; s[i]; ++i)
        // parcurge s cu indice i pana la '\0'
        if (s[i] == c)
            return &s[i];
        // s-a gasit: returneaza adresa

    return NULL;
    // nu s-a gasit: returneaza NULL
}
```

## Exemple: parcurgere cu pointeri vs. indici

Folosim direct un pointer la adresa elementului  $\&a[i] == a+i$   
⇒ în loc să avansăm indicele, incrementăm pointerul.

```
char *strchr_p(char *s, int c) {
    // cauta un caracter in sir
    // scrisa folosind pointeri

    for ( ;*s; ++s)
        // folosim chiar parametrul pentru parcurgere
        if (*s == c)
            return s;
        // s indica elem. curent

    return NULL; // nu s-a gasit
}
```

## Alocare dinamică de memorie

Până acum: adrese ale unor variabile existente, alocate static.

Funcțiile de **alocare dinamică** a memoriei (declarate în **stdlib.h**) ne permit să creem **variabile noi** de dimensiunile necesare apărute la rularea programului:

**void \*malloc(size\_t size);**

– alocă *size* octeți

**void \*calloc(size\_t num, size\_t size);**

– alocă *num\*size* octeți

**void \*realloc(void \*ptr, size\_t size);**

– modică dimensiunea unei zone alocate cu *c/malloc* la *size*, mută blocul de memorie dacă e necesar.

## Alocare dinamică de memorie

Funcțiile *m/c/realloc* returnează:

**adresa de început** a blocului unde a fost alocat numărul de octeți cerut, dacă alocarea de memorie a fost posibilă

**NULL** în caz de eroare (memorie insuficientă!)

⇒ **trebuie testat rezultatul returnat !=NULL**

Memoria alocată dinamic **trebuie eliberată** când nu mai e necesară; folosim tot o funcție din **stdlib.h**:

**void free(void \*ptr);**

## Exemplu – tablou de int alocat dinamic

```
int i, n, *t;
printf("Nr. de elemente ?");
scanf("%d", &n);

t = malloc(n * sizeof(int));
//alocam memorie pentru n elemente de tip int

if (t != NULL)
//daca alocarea memoriei nu a dat eroare
//citim elementele tabloului
    for (i = 0; i < n; i++)
        scanf("%d", &t[i]);
```

## Când folosim alocarea dinamică?

**NU** e necesară dacă **știm de dinainte** (la compilare) de câtă memorie avem nevoie (putem aloca memorie static, e mai simplu!).

**DA** dacă **nu știm la compilare** câtă memorie ne trebuie (de ex, tablouri cu dimensiuni aflate la rulare, etc.).

**DA** dacă trebuie să returnăm **adresa unui obiect creat într-o funcție** (**NU putem returna adresa de variabilă locală**, memoria alocată e alocată doar temporar, pe stivă, și dispare la revenirea din funcție!)

**DA** dacă trebuie să păstrăm un obiect citit într-un loc temporar

## Exemplu – creare obiect într-o funcție

```
char *strup(const char *s) {  
    // creeaza copie a lui s  
  
    char *d = malloc(strlen(s) + 1);  
    // loc pentru sir si '\0'  
  
    if (d!=NULL){ // alocare ok  
        // fa copia, returneaza d  
        strcpy(d, s);  
        return d;  
  
    } else // eroare la alocare  
        return NULL;  
}
```