

# PROGRAMAREA CALCULATOARELOR

Andrei Patrascu

[andrei.patrascu@fmi.unibuc.ro](mailto:andrei.patrascu@fmi.unibuc.ro)

Secția Calculatoare si Tehnologia  
Informatiei, anul I, 2018-2019

Cursul 8

# PROGRAMA CURSULUI

## □ Introducere

- Algoritmi.
- Limbaje de programare.
- Introducere în limbajul C. Structura unui program C.
- Complexitatea algoritmilor.

## □ Fundamentele limbajului C

- Tipuri de date fundamentale. Variabile. Constante. Operatori. Expresii. Conversii.
- Instrucțiuni de control
- Directive de preprocesare. Macrodefiniții.
- Funcții de citire/scriere.
- Etapele realizării unui program C.

## □ Tipuri derivate de date

- Tablouri. Șiruri de caractere.
- Structuri, uniuni, câmpuri de biți, enumerări.
- Pointeri.

## □ Funcții (1)

- Declarație și definire. Apel. Metode de transmitere a parametrelor.
- Pointeri la funcții.

## □ Tablouri și pointeri



- Legătura dintre tablouri și pointeri
- Aritmetica pointerilor
- Alocarea dinamică a memoriei
- Clase de memorare

## □ Șiruri de caractere

- Funcții specifice de manipulare.

## □ Fișiere text și fișiere binare

- Funcții specifice de manipulare.

## □ Structuri de date complexe și autoreferite

- Definire și utilizare

## □ Funcții (2)

- Funcții cu număr variabil de argumente.
- Preluarea argumentelor funcției main din linia de comandă.

# CURSUL DE AZI

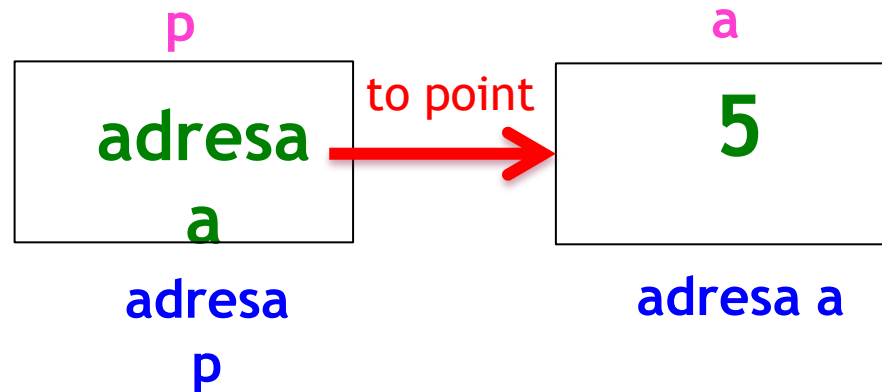
1. Legătura dintre tablouri și pointeri
2. Aritmetica pointerilor
3. Alocare dinamica

# LEGĂTURA DINTRE POINTERI ȘI TABLOURI 1D

- un pointer: variabilă care poate stoca adrese de memorie

- exemple: 

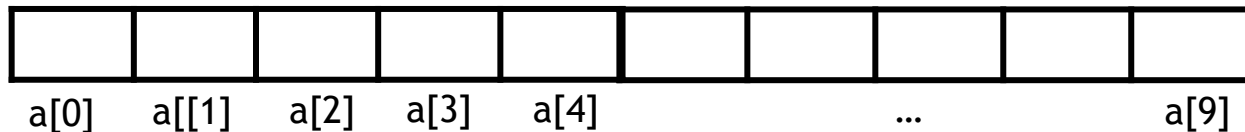
```
int a=5  
int *p;  
p = &a;
```



- un tablou 1D: set de valori de același tip memorat la adrese succesive de memorie

- exemplu: 

```
int a[10];
```

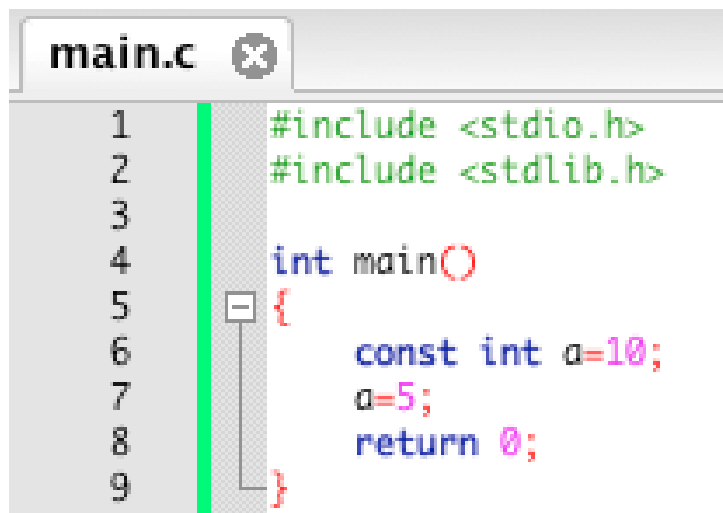


# LEGĂTURA DINTRE POINTERI ȘI TABLOURI 1D

- ❑ adresarea unui element dintr-un tablou cu ajutorul pointerilor
- ❑ inițializarea pointerului p cu adresa primului element al unui tablou
  - ❑ **`int *p = v;`**
  - ❑ **`p = &v[0];`**
  - ❑ **numele unui tablou este un pointer (constant) spre primul său element**
- ❑ cum pot să găsesc adresa/valoarea celui de-al i-lea element din vectorul v pe baza pointerului p (p pointează către adresa de început a tabloului)?

# MODELATORUL CONST

- ❑ modelatorul **const** precizează pentru o variabilă inițializată că nu este posibilă modificarea variabilei respectivă. Dacă se încearcă acest lucru se returnează eroare la compilarea programului.



```
main.c [X]
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      const int a=10;
7      a=5;
8      return 0;
9  }
```

In function 'main':

error: assignment of read-only variable 'a'

== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ==

# MODELATORUL CONST

- ❑ modelatorul **const** precizează pentru o variabilă inițializată că nu este posibilă modificarea variabilei respective. Dacă se încearcă acest lucru se returnează eroare la compilarea programului.
- ❑ putem modifica valoarea unei variabile însoțite de modelatorul **const** prin intermediul unui pointer (în mod indirect):

```
main.c [X]
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      const int a=10;
7      int *p=&a;
8      *p=5;
9      printf("a = %d \n",a);
10     return 0;
11 }
12
```

```
-----
a = 5
```

```
Process returned 0 (0x0)   execution time : 0.005 s
Press ENTER to continue.
```

# POINTERI LA VALORI COSTANTE

- ❑ modelatorul **const** poate preciza pentru un pointer că valoarea variabilei aflate la adresa conținută de pointer nu se poate modifica.

```
main.c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int a=10;
7      const int *p=&a;
8      *p=5;
9      printf("a = %d \n",a);
10     return 0;
11 }
```

In function 'main':

error: assignment of read-only location

```
== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) .
```

- putem modifica valoarea pointerului:

```

4 int main()
5 {
6     int a=10,b=7;
7     const int *p=&a;
8     p = &b;
9     printf("p=%d \n",*p);
10    return 0;
11 }

```

\*p=7

```
Process returned 0 (0x0)    execution time : 0.004 s
Press ENTER to continue.
```



# POINTERI CONSTANȚI

- modelatorul **const** poate preciza pentru un pointer că nu poate referi o altă adresă decât cea pe care o conține la inițializare.

```
4 int main()
5 {
6     int a=10;
7     int* const p=&a;
8     printf("p=%d \n",*p);
9     int b;
10    p = &b;
11    printf("p=%d \n",*p);
12    return 0;
13 }
```

10

error: assignment of read-only variable 'p'

== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s))

- putem modifica valoarea variabilei aflate la adresa conținută de pointer:

```
4 int main()
5 {
6     int a=10;
7     int* const p=&a;
8     printf("p=%d \n",*p);
9     *p = 5;
10    printf("p=%d \n",*p);
11    return 0;
12 }
```

\*p=10

\*p=5

Process returned 0 (0x0)  
Press ENTER to continue.

execution time : 0.004 s

# POINTERI CONSTANȚI LA VALORI CONSTANTE

- modelatorul **const** poate preciza pentru un pointer că nu poate referi o altă adresă decât cea pe care o conține la inițializare și de asemenea că nu poate schimba valoarea variabilei aflate la adresa pe care o conține.

```
main.c x
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int a=10;
7      const int* const p=&a;
8      printf("*p=%d \n",*p);
9      *p = 5;
10     int b = 5;
11     p = &b;
12     printf("*p=%d \n",*p);
13     return 0;
14 }
15
```

9 error: assignment of read-only location

11 error: assignment of read-only variable 'p'

== Build failed: 2 error(s), 0 warning(s) (0 minute(s), 0

# POINTERI CONSTANȚI VS. VALORI CONSTANTE

- diferențele constau în poziționarea modelatorului **const** înainte sau după caracterul \*:
  - pointer constant: `int* const p;`
  - pointer la o constantă: `const int* p;`
  - pointer constant la o constantă: `const int* const p;`
- dacă declarăm o funcție astfel:

```
void f(const int* p)
```

atunci valorile din zona de memoria referită de `p` nu pot fi modificate.

# LEGĂTURA DINTRE POINTERI ȘI TABLOURI 1D

- ❑ adresarea unui element dintr-un tablou cu ajutorul pointerilor
- ❑ inițializarea pointerului p cu adresa primului element al unui tablou
  - ❑ **`int *p = v;`**
  - ❑ **`p = &v[0];`**
  - ❑ **numele unui tablou este un pointer (constant) spre primul său element**
- ❑ cum pot să găsesc adresa/valoarea celui de-al i-lea element din vectorul v pe baza pointerului p (p pointează către adresa de început a tabloului)?

# LEGĂTURA DINTRE POINTERI ȘI TABLOURI 1D

- ❑ adresarea unui element dintr-un tablou cu ajutorul pointerilor

main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main(){
6
7      int v[5] = {0,2,4,10,20};
8      int *p = v;
9      int i;
10     for (i=0;i<5;i++)
11     {
12         printf("Accesam elementul %d din vector v prin intermediul lui p.\n",i);
13         printf("Valoarea acestui element este = %d \n",*(p+i));
14     }
15
16     p = &v[0];
17     for (i=0;i<5;i++)
18     {
19         printf("Accesam elementul %d din vector v prin intermediul lui p.\n",i);
20         printf("Valoarea acestui element este = %d \n",*(p+i));
21     }
22     return 0;
23 }
24
```

# LEGĂTURA DINTRE POINTERI ȘI TABLOURI 1D

- ❑ adresarea unui element dintr-un tablou cu ajutorul pointerilor

main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main() {
6
7      int v[5] = {0,2,4,10,20};
8      int *p = v;
9      int i;
10     for (i=0; i<5; i++)
11     {
12         printf("Accesam elementul ");
13         printf("Valoarea acestui element este ");
14     }
15
16     p = &v[0];
17     for (i=0; i<5; i++)
18     {
19         printf("Accesam elementul ");
20         printf("Valoarea acestui element este ");
21     }
22     return 0;
23 }
24
```

Accesam elementul 0 din vector v prin intermediul lui p.  
Valoarea acestui element este = 0  
Accesam elementul 1 din vector v prin intermediul lui p.  
Valoarea acestui element este = 2  
Accesam elementul 2 din vector v prin intermediul lui p.  
Valoarea acestui element este = 4  
Accesam elementul 3 din vector v prin intermediul lui p.  
Valoarea acestui element este = 10  
Accesam elementul 4 din vector v prin intermediul lui p.  
Valoarea acestui element este = 20  
Accesam elementul 0 din vector v prin intermediul lui p.  
Valoarea acestui element este = 0  
Accesam elementul 1 din vector v prin intermediul lui p.  
Valoarea acestui element este = 2  
Accesam elementul 2 din vector v prin intermediul lui p.  
Valoarea acestui element este = 4  
Accesam elementul 3 din vector v prin intermediul lui p.  
Valoarea acestui element este = 10  
Accesam elementul 4 din vector v prin intermediul lui p.  
Valoarea acestui element este = 20



# LEGĂTURA DINTRE POINTERI ȘI TABLOURI 1D

- ❑ adresarea unui element dintr-un tablou cu ajutorul pointerilor
- ❑ adresa lui  $v[i]$ :  $\&v[i] = p+i$
- ❑ valoarea lui  $v[i]$ :  $v[i] = *(p+i)$
- ❑ comutativitate:  $v[i] = *(p+i) = *(i+p) = i[v] ?!$

# LEGĂTURA DINTRE POINTERI ȘI TABLOURI 1D

- ❑ adresarea unui element dintr-un tablou cu ajutorul pointerilor

```
main.c [X]
1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main() {
6
7      int v[5] = {0,2,4,10,20};
8      int i;
9
10     printf("Afisare v[i] \n");
11     for(i=0;i<5;i++)
12         printf("v[%d]=%d \n",i,v[i]);
13
14     printf("Afisare i[v] \n");
15     for(i=0;i<5;i++)
16         printf("%d[v]=%d \n",i,i[v]);
17
18
19     return 0;
20 }
21
```

```
Afisare v[i]
v[0]=0
v[1]=2
v[2]=4
v[3]=10
v[4]=20
Afisare i[v]
0[v]=0
1[v]=2
2[v]=4
3[v]=10
4[v]=20
```

```
Process returned 0 (0x0)    execution ti
Press ENTER to continue.
```

Concluzie?



# LEGĂTURA DINTRE POINTERI ȘI TABLOURI 1D

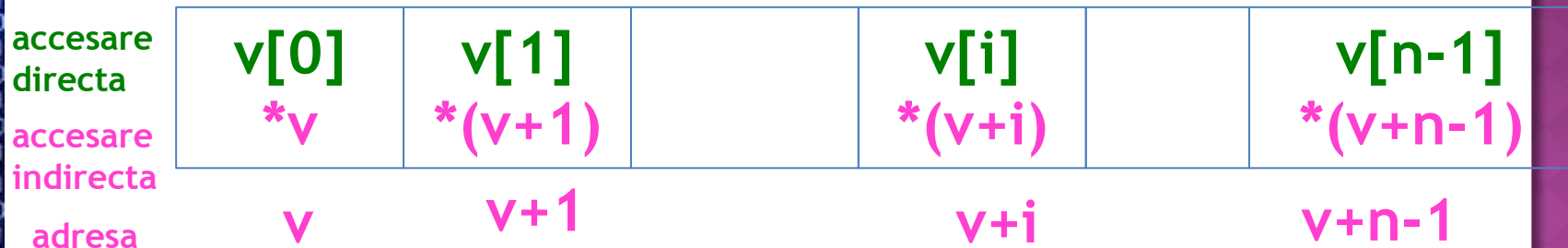
- ❑ adresarea unui element dintr-un tablou cu ajutorul pointerilor
- ❑ *conceptul de tablou nu există în limbajul C. Numele unui tablou este un pointer (**constant**) spre primul său element.*

# LEGĂTURA DINTRE POINTERI ȘI TABLOURI 1D

- numele unui tablou este un pointer constant spre primul său element.

`int v[100];`  `v = &v[0];`

- elementele unui tablou pot fi accesate prin pointeri:



- $*(v+1) = v[1]$ , insa  $*v+1 = v[0] + 1$

# LEGĂTURA DINTRE POINTERI ȘI TABLOURI 1D

n-1

Numele unui tablou = valoarea adresei primului element  
= un pointer constant (nu poate fi schimbat)


```
int a[100], b[100];  
int *p;  
int x;  
...  
p=a; // p reține adresa lui a[0]  
b=a; // gresit! -> b este pointer constant  
x=a[0] este echivalentă cu x=*p;  
x=a[10] este echivalentă cu x=*(p+10);  
    // al unsprezece-lea element din tablou  
    (elementul de pe pozitia numarul 10)
```

# LEGĂTURA DINTRE POINTERI ȘI TABLOURI 2D

```
int a[3][5];
```

```
a[1][4] = 41;
```

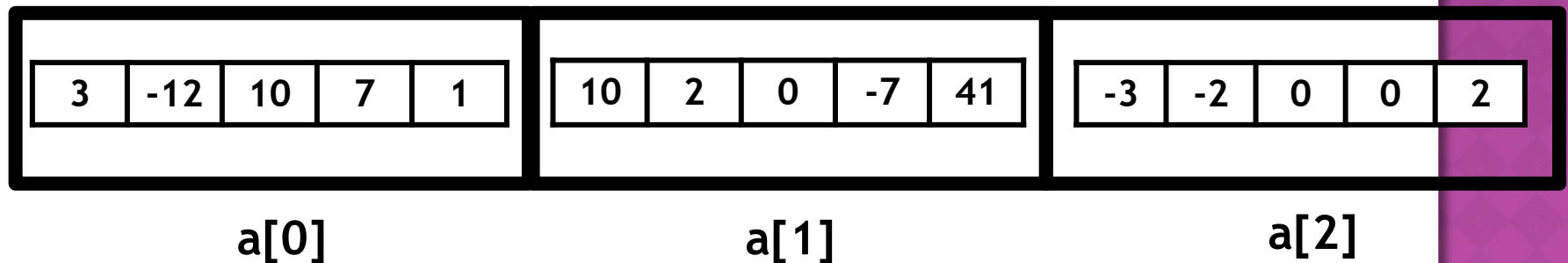
	0	1	2	3	4
0	3	-12	10	7	1
1	10	2	0	-7	41
2	-3	-2	0	0	2



3	-12	10	7	1	10	2	0	-7	41	-3	-2	0	0	2
a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]	a[1][0]	...								a[2][4]

Reprezentarea în memoria calculatorului a unui tablou bidimensional

❑ **tablou bidimensional = tablou de tablouri**



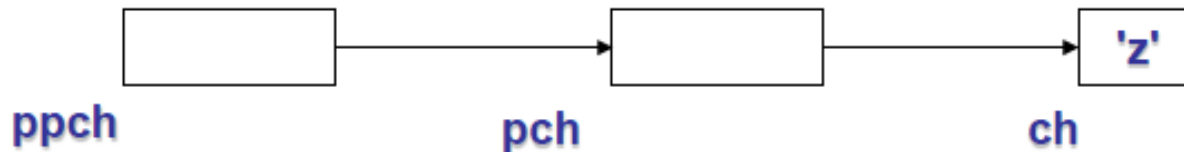
# POINTERI LA POINTERI (POINTERI DUBLI)

## □ sintaxa

**tip      \*\*nume\_variabilă;**

**tip** = tipul de bază al variabilei de tip pointer dublu nume\_variabilă;  
**nume\_variabila** = variabila de tip pointer dublu care poate lua ca valori adrese de memorie ale unor variabile de tip pointer.

```
char ch = 'z'; // un caracter  
char *pch; // un pointer la caracter  
char **ppch; // un pointer la un pointer la caracter  
pch = &ch; ppch = &pch;
```



```
printf("%p %p %c",ppch,pch,ch); // 0028FF04 0028FF0B z
```

# POINTERI LA POINTERI

## ❑ exemplu:

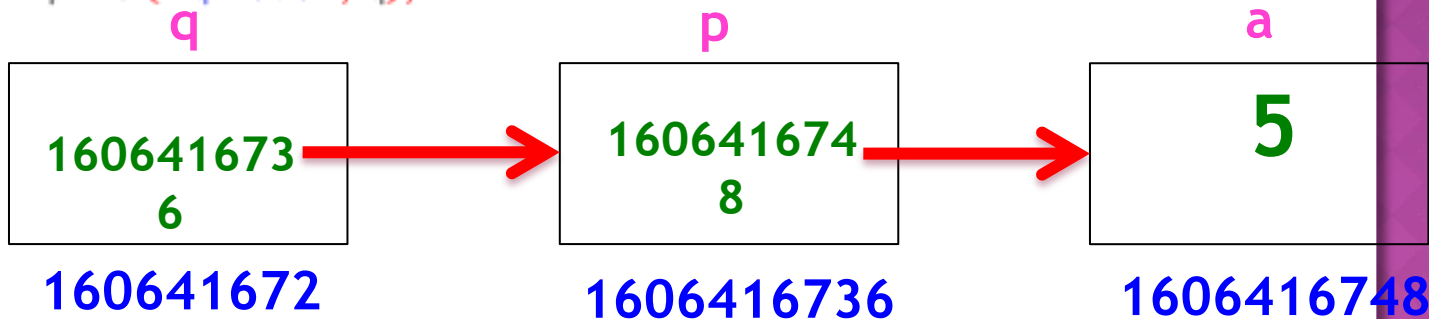
```
main.c x
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main()
6  {
7      int a = 5, *p = &a, **q = &p;
8
9      printf("Valoarea lui a poate fi obtinuta astfel:\n");
10     printf("Direct: a = %d \n", a);
11     printf("Prin p: *p = %d \n", *p);
12     printf("Prin q : **q = %d \n", **q);
13
14     printf("Adresa lui a este: %d\n", &a);
15     printf("Adresa lui p este: %d\n", &p);
16     printf("Adresa lui q este: %d\n", &q);
17     printf("Adresa spre care pointeaza p este %d\n", p);
18     printf("Adresa spre care pointeaza q este %d\n", q);
19     printf("**q = %d\n", *q);
20 }
```

# POINTERI LA POINTERI

## ❑ exemplu:

```
main.c x
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main()
6  {
7      int a = 5, *p = &a, **q = &p;
8
9      printf("Valoarea lui a poate fi obtinuta astfel:");
10     printf("Direct: a = %d \n",a);
11     printf("Prin p: *p = %d \n",*p);
12     printf("Prin q : **q = %d \n",**q);
13
14     printf("Adresa lui a este: %d\n",&a);
15     printf("Adresa lui p este: %d\n",&p);
16     printf("Adresa lui q este: %d\n",&q);
17     printf("Adresa spre care pointeaza p este %d\n",p);
18     printf("Adresa spre care pointeaza q este %d\n",q);
19     printf("**q = %d\n",*q);
20 }
```

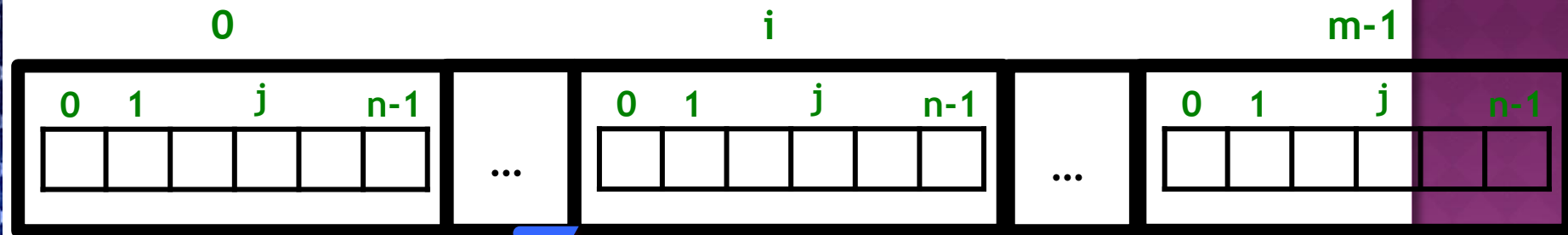
Valoarea lui a poate fi obtinuta astfel:  
Direct: a = 5  
Prin p: \*p = 5  
Prin q : \*\*q = 5  
Adresa lui a este: 1606416748  
Adresa lui p este: 1606416736  
Adresa lui q este: 1606416728  
Adresa spre care pointeaza p este 1606416748  
Adresa spre care pointeaza q este 1606416736  
Process returned 0 (0x0) execution time : 0.009 s  
Press ENTER to continue.





# LEGĂTURA DINTRE POINTERI ȘI TABLOURI 2D

- ❑ tablou bidimensional = tablou de tablouri
- ❑ cazul general: `int a[m][n];`



`a[i]` este un tablou unidimensional.



# LEGĂTURA DINTRE POINTERI ȘI TABLOURI 2D

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a[5][5], i, j;

    printf("Adresa inceput tablou a: %d \n", a);

    for (i = 0; i < 5 ; i++)
        printf("%d %d %d \n", a[i], a+i, *(a+i));

    return 0;
}
```

# LEGĂTURA DINTRE POINTERI ȘI TABLOURI 2D

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main()
```

```
{
```

```
    int a[5][5], i, j;
```

```
    printf("Adresa inceput tablou a: %d \n", a);
```

```
    for (i = 0; i < 5 ; i++)
```

```
        printf("%d %d %d \n", a[i], a+i, *(a+i));
```

```
    return 0;
```

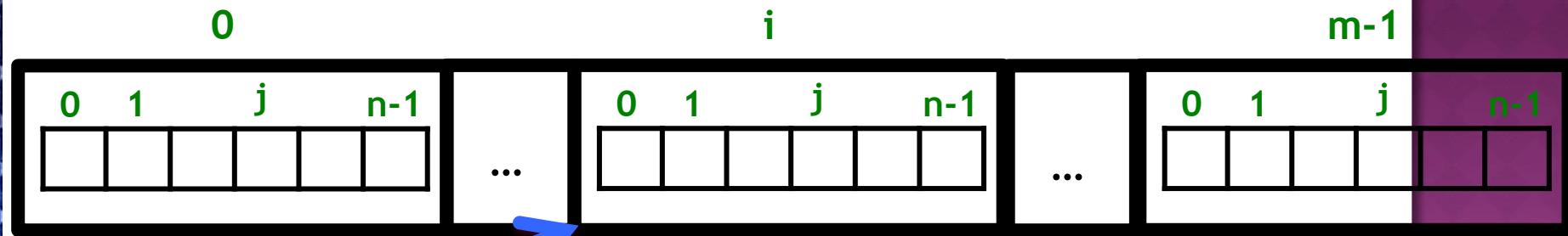
```
}
```

```
Adresa inceput tablou a: 2686680
2686680 2686680 2686680
2686700 2686700 2686700
2686720 2686720 2686720
2686740 2686740 2686740
2686760 2686760 2686760
```

```
Process returned 0 (0x0)   execution time : 0.010 s
Press any key to continue.
```

# LEGĂTURA DINTRE POINTERI ȘI TABLOURI 2D

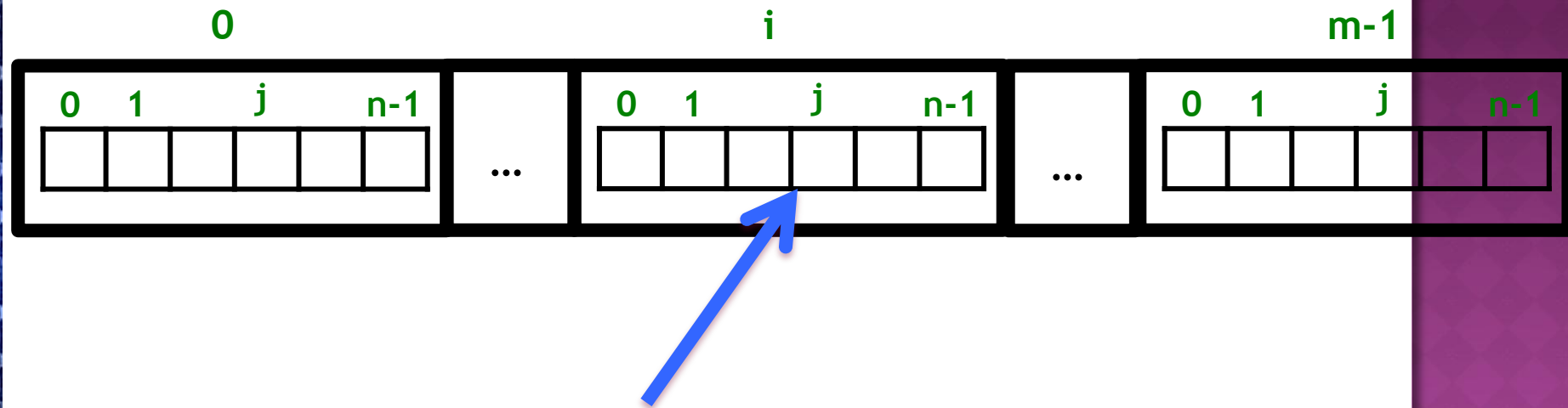
- cazul general: `int a[m][n];`



`a[i]` poate fi considerat nume de tablou =>  
`a[i]` pointer constant de inceput al tabloului

# LEGĂTURA DINTRE POINTERI ȘI TABLOURI 2D

□ cazul general: `int a[m][n];`



Care este adresa lui `a[i][j]`?  $\Rightarrow a[i] + j = *(a + i) + j$

# LEGĂTURA DINTRE POINTERI ȘI TABLOURI 2D

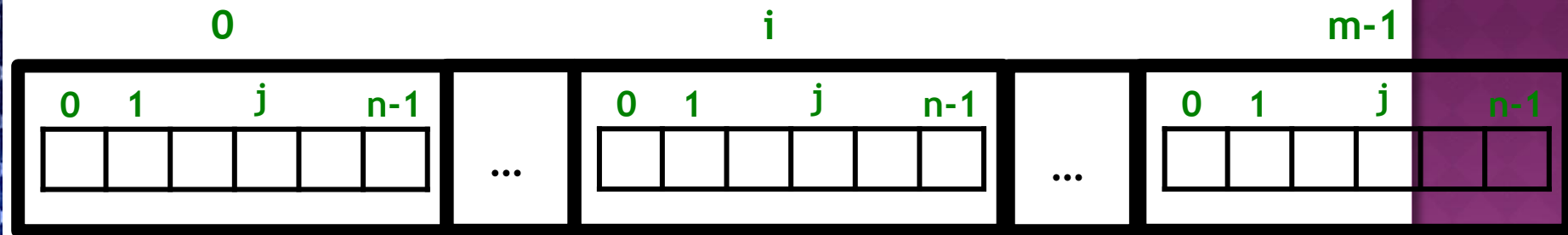
- ❑ tablou bidimensional = tablou de tablouri

```
tablou_pointer.c
1  #include <stdio.h>
2
3  int main()
4  {
5      int a[5][5], i, j;
6
7      i = 3;
8
9      for (j=0; j<5; j++)
10     {
11         printf("Adresa lui a[%d][%d] este %d \n", i, j, &a[i][j]);
12         printf("Adresa lui a[%d][%d] este %d \n", i, j, *(a+i)+j);
13     }
14
15     return 0;
16 }
17
```

```
Adresa lui a[3][0] este 1606416668
Adresa lui a[3][0] este 1606416668
Adresa lui a[3][1] este 1606416672
Adresa lui a[3][1] este 1606416672
Adresa lui a[3][2] este 1606416676
Adresa lui a[3][2] este 1606416676
Adresa lui a[3][3] este 1606416680
Adresa lui a[3][3] este 1606416680
Adresa lui a[3][4] este 1606416684
Adresa lui a[3][4] este 1606416684
```

# LEGĂTURA DINTRE POINTERI ȘI TABLOURI 2D

▣ cazul general: `int a[m][n];`



Care este adresa lui  $a[i][j]$ ?

$\&a[i][j] = *(a+i)+j = a[i] + j$  (a este pointer dublu).

Cum exprim valoarea lui  $a[i][j]$  în aritmetica pointerilor în funcție de a, i, j?

$*\&a[i][j] = * ( * (a + i) + j) = *(a[i] + j)$

# LEGĂTURA DINTRE POINTERI ȘI TABLOURI 2D

- ❑ tablou bidimensional = tablou de tablouri

```
tablou_pointer_2.c ×
1  #include <stdio.h>
2
3  int main()
4  {
5      int a[5][5], i, j;
6
7      for(i=0; i<5; i++)
8          for(j=0; j<5; j++)
9              a[i][j] = i*j;
10
11     i = 3;
12
13     for (j=0; j<5; j++)
14     {
15         printf("Valoarea lui a[%d][%d] este %d \n", i, j, a[i][j]);
16         printf("Valoarea lui a[%d][%d] este %d \n", i, j, (*(a+i)+j));
17     }
18
19     return 0;
20 }
```

```
Valoarea lui a[3][0] este 0
Valoarea lui a[3][0] este 0
Valoarea lui a[3][1] este 3
Valoarea lui a[3][1] este 3
Valoarea lui a[3][2] este 6
Valoarea lui a[3][2] este 6
Valoarea lui a[3][3] este 9
Valoarea lui a[3][3] este 9
Valoarea lui a[3][4] este 12
Valoarea lui a[3][4] este 12
```

# LEGĂTURA DINTRE POINTERI ȘI TABLOURI 2D

▣ tablou bidimensional = tablou de tablouri

## Recapitulam:

Adresa lui  $a[i][j] = *(a+i)+j$  (a este pointer dublu).

Valoarea lui  $a[i][j] = *(*a+i)+j$

Știu că  $a[i] = *(a+i) = i[a]$ .

Atunci  $a[i][j]$  se mai poate scrie ca:

1.  $*(a[i]+j)$
2.  $*(i[a] + j)$
3.  $(*(a+i))[j]$
4.  $i[a][j]$
5.  $j[i[a]]$
6.  $j[a[i]]$



# CURSUL DE AZI

1. Legătura dintre tablouri și pointeri
2. Aritmetica pointerilor
3. Alocare dinamica

# ARITMETICA POINTERILOR

- ❑ **asupra pointerilor pot fi realizate operații aritmetice:**
  - ❑ incrementare (++), decrementare (--);
  - ❑ adăugare (+ sau +=) sau scădere a unui întreg (- sau -=)
  - ❑ scădere a unui pointer din alt pointer;
  - ❑ asignări;
  - ❑ comparații.

# ARITMETICA POINTERILOR

- ❑ inițializarea unui pointer cu adresa primul element al unui tablou

```
main.c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main() {
6
7      int v[5];
8      int *p;
9
10     p = &v[0];
11     printf("Adresa lui v[0] este %x \n", p);
12
13     p = v;
14     printf("Adresa lui v este %x \n", p);
15
16
17     return 0;
18 }
19
```

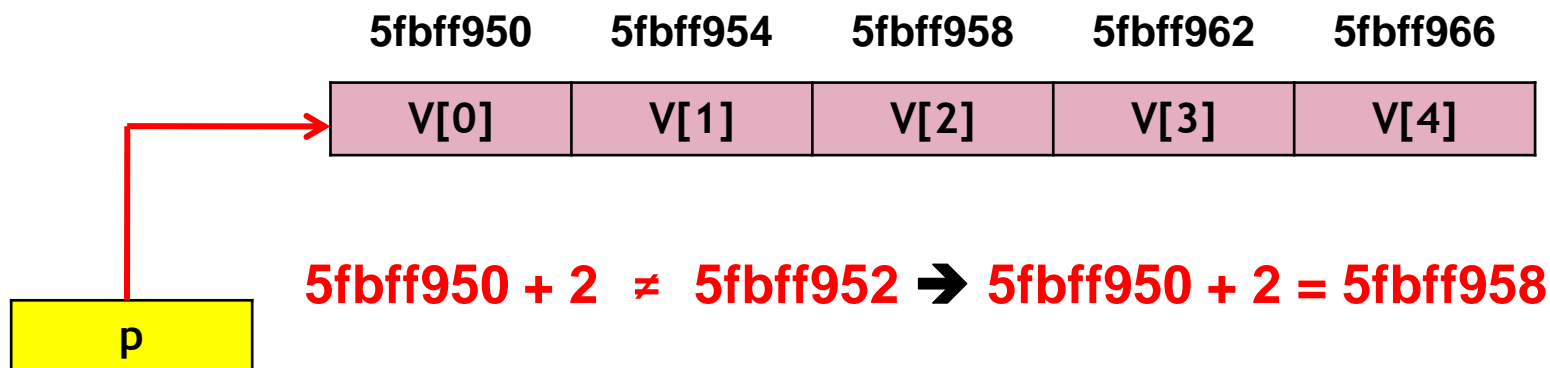
Adresa lui v[0] este 5fbff950  
Adresa lui v este 5fbff950

Process returned 0 (0x0) execution time : 0.  
Press ENTER to continue.

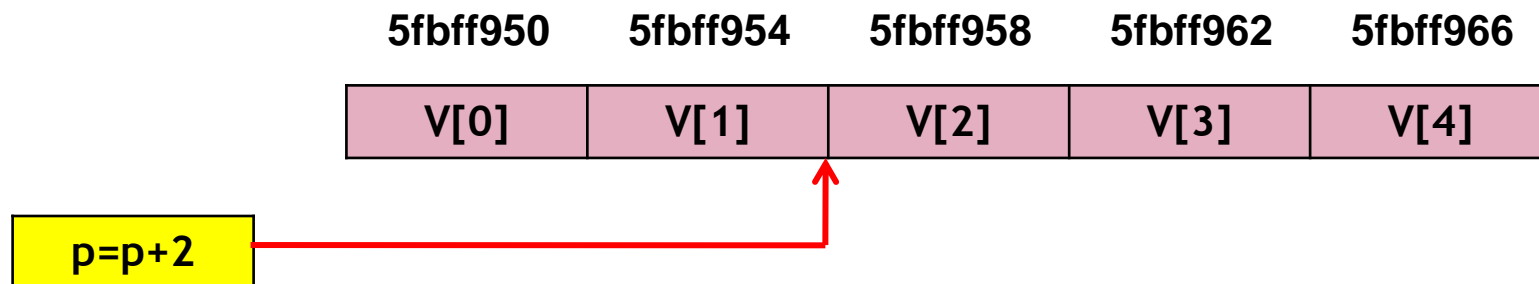
v este un pointer care  
pointeaza către v[0]

# ARITMETICA POINTERILOR

- ❑ adunarea/scăderea unui număr natural dintr-un pointer



- ❑ în aritmetica pointerilor adăugarea unui întreg la o adresă de memorie are ca rezultat o nouă adresă de memorie!



# ARITMETICA POINTERILOR

main.c



```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4
5 int main(){
6
7     int v[5];
8     int *p;
9
10    p = &v[0];
11    printf("Adresa spre care pointeaza acum p este %d \n", p);
12    p+=4;
13    printf("Adresa spre care pointeaza acum p este %d \n", p);
14    p-=2;
15    printf("Adresa spre care pointeaza acum p este %d \n", p);
16    p++;
17    printf("Adresa spre care pointeaza acum p este %d \n", p);
18    ++p;
19    printf("Adresa spre care pointeaza acum p este %d \n", p);
20    p--;
21    printf("Adresa spre care pointeaza acum p este %d \n", p);
22    --p;
23    printf("Adresa spre care pointeaza acum p este %d \n", p);
24
```

# ARITMETICA POINTERILOR

main.c



```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4
5 int main(){
6
7     int v[5];
8     int *p;
9
10    p = &v[0];
11    printf("Adresa spre care pointeaza acum p este %d \n", p);
12    p+=4;
13    printf("Adresa spre care pointeaza acum p este %d \n", p);
14    p-=2;
15    printf("Adresa spre care pointeaza acum p este %d \n", p);
16    p++;
17    printf("Adresa spre care pointeaza acum p este %d \n", p);
18    ++p;
19    printf("Adresa spre care pointeaza acum p este %d \n", p);
20    p--;
21    printf("Adresa spre care pointeaza acum p este %d \n", p);
22    --p;
23    printf("Adresa spre care pointeaza acum p este %d \n", p);
24
```

Adresa spre care pointeaza acum p este 1606416720  
Adresa spre care pointeaza acum p este 1606416736  
Adresa spre care pointeaza acum p este 1606416728  
Adresa spre care pointeaza acum p este 1606416732  
Adresa spre care pointeaza acum p este 1606416736  
Adresa spre care pointeaza acum p este 1606416732  
Adresa spre care pointeaza acum p este 1606416728

Process returned 0 (0x0) execution time : 0.007 s  
Press ENTER to continue.

# ARITMETICA POINTERILOR

- ❑ adunarea/scăderea unui număr natural dintr-un pointer
- ❑ **adunarea cu  $n$ :** adresa aflată peste  $n$  locații de memorie de adresa curentă stocată în pointer (“la dreapta”, se obține adăugând la adresa curentă  $n * \text{sizeof}(*p)$  octeți) de același tip cu tipul de bază al variabilei de tip pointer
- ❑ **scăderea cu  $n$ :** adresa aflată înainte cu  $n$  locații de memorie de adresa curentă stocată în pointer (“la stânga”, se obține scăzând la adresa curentă  $n * \text{sizeof}(*p)$  octeți) de același tip cu tipul de bază al variabilei de tip pointer



# ARITMETICA POINTERILOR

- ❑ scăderea a două variabile de tip pointer

```
main.c x
1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main(){
6
7      int v[5];
8      int *p,*q;
9
10     p = &v[0];
11     printf("Adresa spre care pointeaza acum p este %d \n", p);
12
13     q = &v[3];
14     printf("Adresa spre care pointeaza acum q este %d \n", q);
15
16     printf("Rezultatul diferentei dintre q si p este %d\n",q-p);
17     printf("Rezultatul diferentei dintre p si q este %d\n",p-q);
18
19
20     return 0;
21 }
22
```



# ARITMETICA POINTERILOR

- ❑ scăderea a două variabile de tip pointer

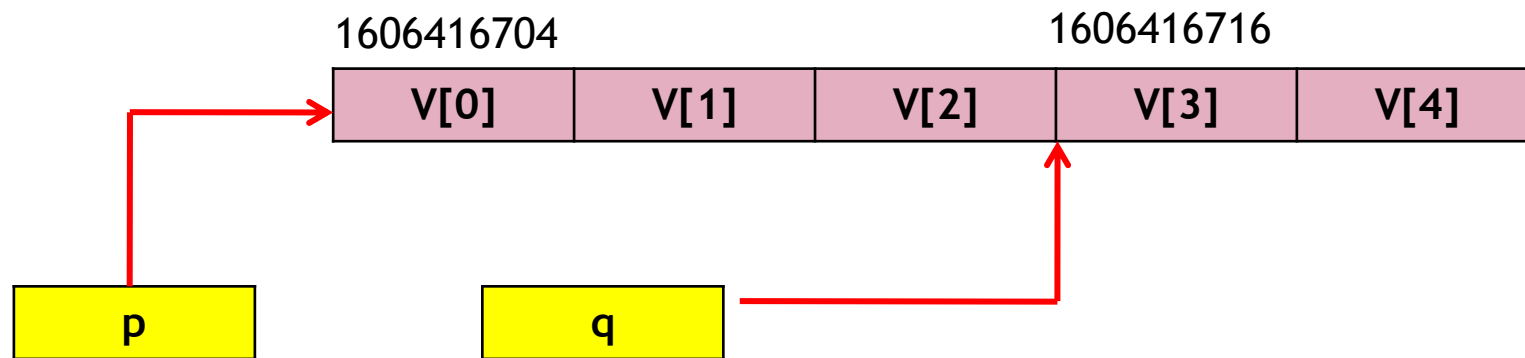
```
main.c x
1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main(){
6
7      int v[5];
8      int *p,*q;
9
10     p = &v[0];
11     printf("Adresa spre care pointeaza acum p este %d \n", p);
12
13     q = &v[3];
14     printf("Adresa spre care pointeaza acum q este %d \n", q);
15
16     printf("Rezultatul diferentei dintre q si p este %d\n",q-p);
17     printf("Rezultatul diferentei dintre p si q este %d\n",p-q);
18
19
20     return 0;
21 }
22
```

```
Adresa spre care pointeaza acum p este 1606416704
Adresa spre care pointeaza acum q este 1606416716
Rezultatul diferentei dintre q si p este 3
Rezultatul diferentei dintre p si q este -3
```

```
Process returned 0 (0x0)    execution time : 0.006 s
Press ENTER to continue.
```

# ARITMETICA POINTERILOR

- scăderea a două variabile de tip pointer



- În aritmetica pointerilor diferența dintre doi pointeri reprezintă numărul de obiecte de același tip care despart cele două adrese

$p - q > 0$  înseamnă că *p* e la dreapta lui *q*

$p - q < 0$  înseamnă că *p* e la stânga lui *q*

# ARITMETICA POINTERILOR

- compararea a două variabile de tip pointer

```
main.c x
1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main(){
6
7      int v[5];
8      int *p,*q;
9
10     p = &v[2];
11     printf("Adresa spre care pointeaza acum p este %d \n", p);
12     q = &v[4];
13     printf("Adresa spre care pointeaza acum q este %d \n", q);
14
15     p > q ? printf("p este la dreapta lui q\n"):printf("p este la stanga lui q\n");
16     q = &v[0];
17     printf("Adresa spre care pointeaza acum q este %d \n", q);
18     p > q ? printf("p este la dreapta lui q\n"):printf("p este la stanga lui q\n");
19
20     return 0;
21 }
```

# ARITMETICA POINTERILOR

- compararea a două variabile de tip pointer

main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main(){
6
7      int v[5];
8      int *p,*q;
9
10     p = &v[2];
11     printf("Adresa spre care pointeaza acum p este %d \n", p);
12     q = &v[4];
13     printf("Adresa spre care pointeaza acum q este %d \n", q);
14
15     p > q ? printf("p este la dreapta lui q\n"):printf("p este la stanga lui q\n");
16     q = &v[0];
17     printf("Adresa spre care pointeaza acum q este %d \n", q);
18     p > q ? printf("p este la dreapta lui q\n"):printf("p este la stanga lui q\n");
19
20     return 0;
21 }
```

```
Adresa spre care pointeaza acum p este 1606416712
Adresa spre care pointeaza acum q este 1606416720
p este la stanga lui q
Adresa spre care pointeaza acum q este 1606416704
p este la dreapta lui q
```

```
Process returned 0 (0x0)    execution time : 0.006 s
Press ENTER to continue.
```

# ARITMETICA POINTERILOR

- ❑ compararea a două variabile de tip pointer = compararea diferenței lor cu 0

main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main(){
6
7      int v[5];
8      int *p,*q;
9
10     p = &v[2];
11     printf("Adresa spre care pointeaza acum p este %d \n", p);
12     q = &v[4];
13     printf("Adresa spre care pointeaza acum q este %d \n", q);
14
15     p - q > 0? printf("p este la dreapta lui q\n"):printf("p este la stanga lui q\n");
16     q = &v[0];
17     printf("Adresa spre care pointeaza acum q este %d \n", q);
18     p - q > 0? printf("p este la dreapta lui q\n"):printf("p este la stanga lui q\n");
19
20     return 0;
21 }
```

```
Adresa spre care pointeaza acum p este 1606416712
Adresa spre care pointeaza acum q este 1606416720
p este la stanga lui q
Adresa spre care pointeaza acum q este 1606416704
p este la dreapta lui q
```

# ARITMETICA POINTERILOR

- observație: aritmetica pointerilor *are sens și este sigură* dacă adresele implicate sunt adrese ale elementelor unui tablou.

```
main.c x
1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main(){
6
7      double a=3.14,b=2*a;
8      int x=10,y=20,z=30,w=40;
9
10     printf(" a=%f \n b=%f \n x=%d \n y=%d \n z=%d \n w=%d \n",a,b,x,y,z,w);
11
12     double *p = &b;
13     *p = 5.2;
14     *(p+1) = 6.4;
15     *(p+2) = 100.54;
16     *(p+3) = 1000.971;
17
18     printf(" a=%f \n b=%f \n x=%d \n y=%d \n z=%d \n w=%d \n",a,b,x,y,z,w);
19
20     return 0;
```

```
a=3.140000
b=6.280000
x=10
y=20
z=30
w=40
a=6.400000
b=5.200000
x=1083131844
y=-1683627180
z=1079583375
w=1546188227
```

```
Process returned 0 (0x0)
Press ENTER to continue.
```



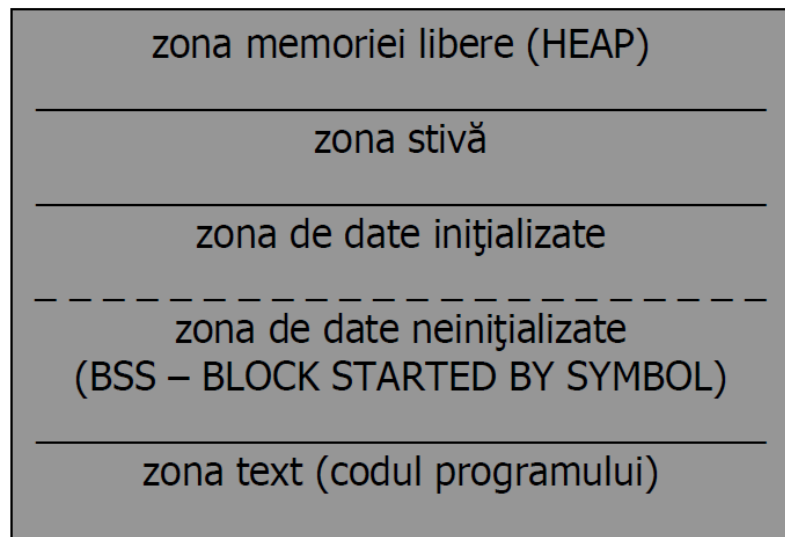
# ALOCAREA DINAMICĂ A MEMORIEI

- ❑ *heap*-ul este o zonă predefinită de memorie (de dimensiuni foarte mari) care poate fi accesată de program pentru a stoca date și variabile
- ❑ datele și variabilele pot fi alocate pe *heap* prin apeluri speciale de funcții din biblioteca *stdlib.h*: **malloc**, **calloc**, **realloc**
- ❑ zonele de memorie pot să fie dezalocate la cerere prin apelul funcției **free**
- ❑ este recomandat ca memoria să fie eliberată în momentul în care datele/variabilele respective nu mai sunt de interes!



# HARTA SIMPLIFICATĂ A MEMORIEI LA RULAREA UNUI PROGRAM

**zona de HEAP: pot aloca  
dinamic memorie aici și prin  
intermediul aritmeticii  
pointerilor reține informație**



**Registers**

# FUNCȚIA MALLOC

## □ prototipul funcției:

***void \* malloc( int dimensiune);***

unde:

- ***dimensiune*** = numărul de octeți ceruți a se alocă
- dacă există suficient spațiu liber în HEAP atunci un bloc de memorie continuu de dimensiunea specificată va fi marcat ca ocupat, iar funcția malloc va returna un pointer ce conține adresa de început a acelui bloc. Dacă nu există suficient spațiu liber funcția malloc întoarce NULL.
- accesarea blocului alocat se realizează printr-un pointer (din STACK) către adresa de început a blocului (din HEAP).

# FUNCȚIA MALLOC

## □ prototipul funcției:

***void \* malloc( int dimensiune);***

unde:

- ***dimensiune*** = numărul de octeți ceruți a se alocă
- tipul generic void \* returnat de funcția malloc face obligatorie utilizarea unei conversii de tip atunci când respectivul pointer este asignat unui pointer de tip obișnuit.
- pointerul în care păstrăm adresa returnată de malloc va fi plasat în zona de memorie statică.

# FUNCTIA MALLOC

## ▣ exemplu:

```
main.c ×
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5
6      int a=0;
7      int *p=&a;
8      printf("Adresa lui a este = %d \n",&a);
9      printf("Adresa lui p este = %d \n",&p);
10     printf("Cerere alocare memorie in HEAP \n");
11     p = (int*) malloc(5*sizeof(int));
12     if (p==NULL)
13     {
14         printf("Nu exista spatiu liber in HEAP \n");
15         exit(0);
16     }
17     else
18         printf("Pointerul p pointeaza catre adresa = %d din HEAP\n",p);
19     int i;
20     for (i=0;i<5;i++)
21         p[i] = i;
22     free(p);
23
24     return 0;
25 }
```

# FUNCȚIA MALLOC

## ▣ exemplu:

main.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5
6     int a=0;
7     int *p=&a;
8     printf("Adresa lui a este = %d \n",&a);
9     printf("Adresa lui p este = %d \n",&p);
10    printf("Cerere alocare memorie in HEAP \n");
11    p = (int*) malloc(5*sizeof(int));
12    if (p==NULL)
13    {
14        printf("Nu exista spatiu liber in HEAP \n");
15        exit(0);
16    }
17    else
18        printf("Pointerul p pointeaza catre adresa = %d din HEAP\n",p);
19    int i;
20    for (i=0;i<5;i++)
21        p[i] = i;
22    free(p);
23
24    return 0;
25 }
```

Adresa lui a este = 1606416748

Adresa lui p este = 1606416736

Cerere alocare memorie in HEAP

Pointerul p pointeaza catre adresa = 1048704 din HEAP

Process returned 0 (0x0) execution time : 0.008 s

Press ENTER to continue.

# FUNCȚIA MALLOC

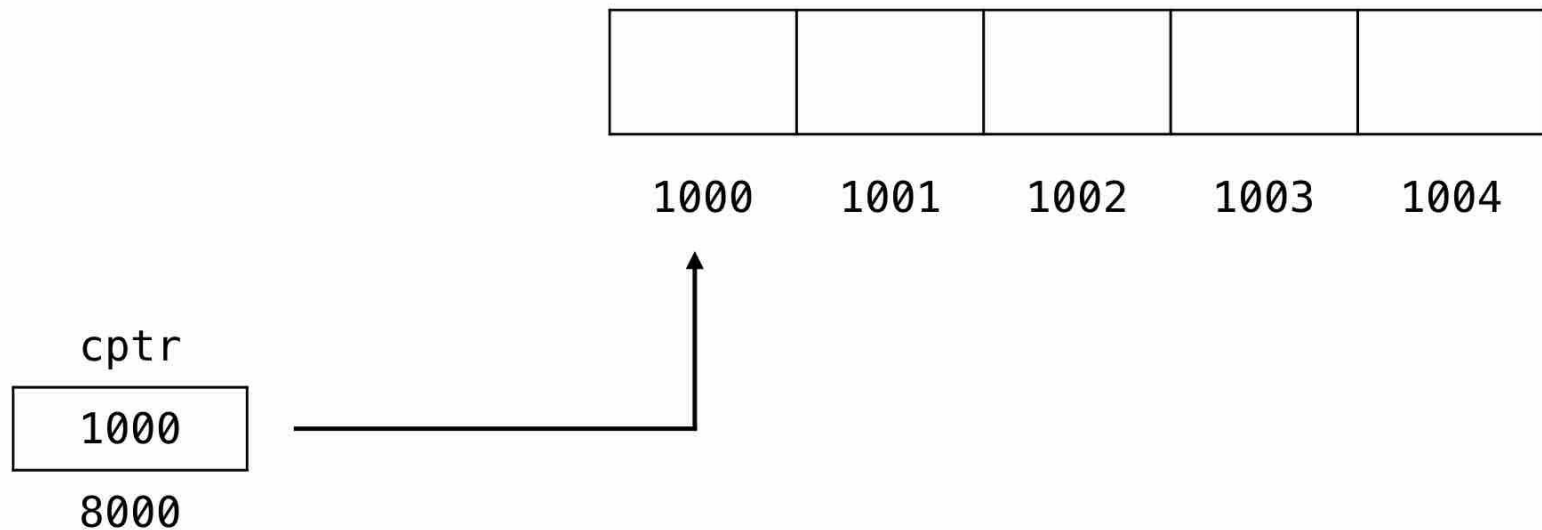
## Observatii:

- ❑ blocurile alocate în zona de memorie dinamică **nu au nume**
- ❑ **mod de acces**: adresa de memorie.
- ❑ accesul blocului de memorie se realizează prin intermediul unui pointer în care păstrăm adresa de început.
- ❑ orice bloc de memorie alocat dinamic trebuie **eliberat** înainte să se încheie execuția programului. Funcția **free** permite eliberarea memoriei (parametru: adresa de început a blocului).

# FUNCȚIA MALLOC

 CLASSROOM

```
char *cptr = (char *) malloc (5 * sizeof(char));
```



dyclassroom.com



# FUNCȚIA MALLOC

Funcție pentru citirea unui tablou unidimensional: se citește numărul de elemente, se alocă dinamic tabloul și se citesc elementele tabloului.

```
main.c x
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int citire(int *v)
5  {
6      int i,n;
7      printf("n=");scanf("%d",&n);
8      v =(int *)malloc(n*sizeof(int));
9      for (i=0;i<n;i++)
10         scanf("%d",&v[i]);
11     return n;
12 }
13
14
15 int main()
16 {
17     int n,*p=NULL;
18     n=citire(p);
19     int i;
20     for(i=0;i<n;i++)
21         printf("p[%d]=%d",i,p[i]);
22
23     return 0;
24 }
```

# FUNCȚIA MALLOC

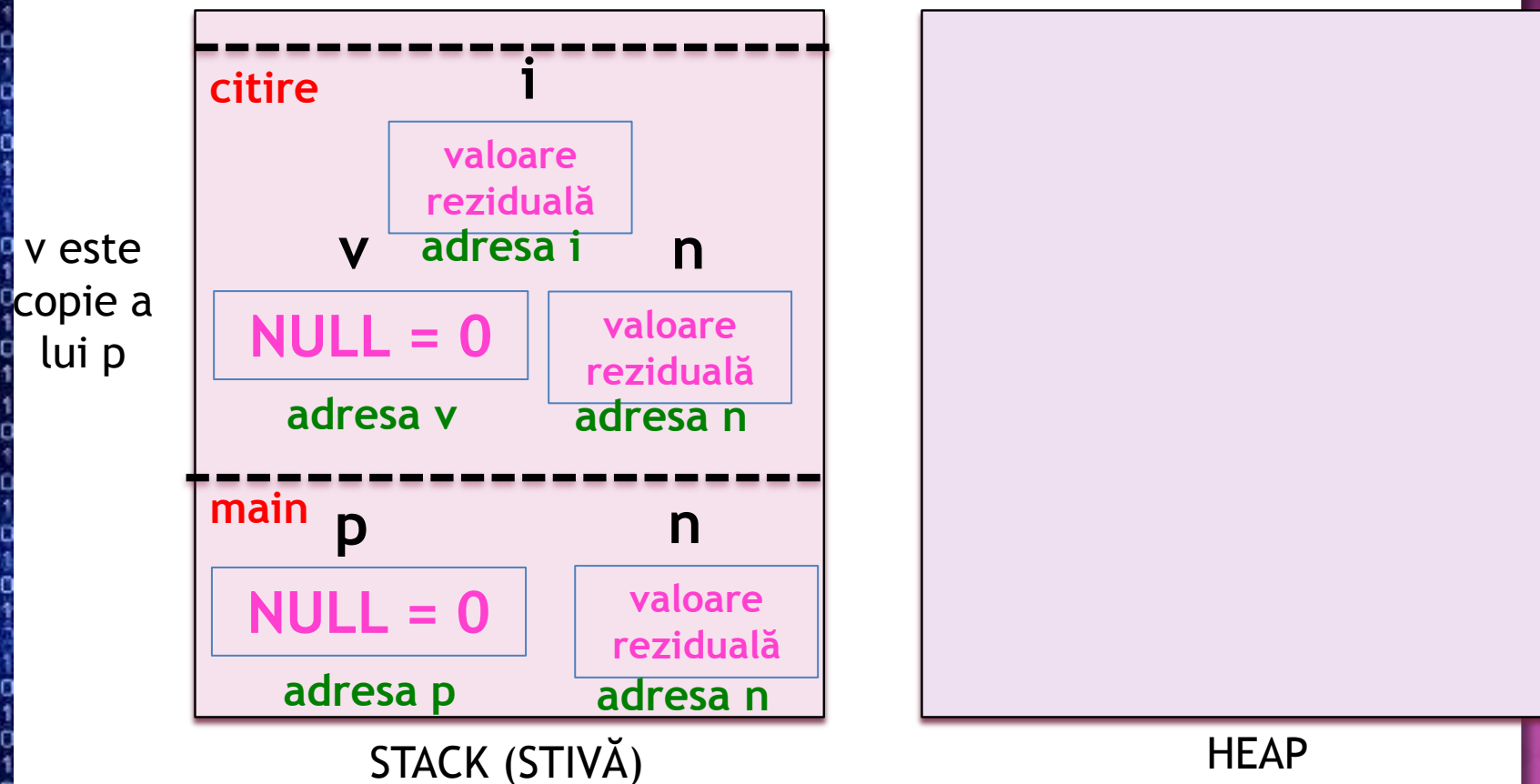
Funcție pentru citirea unui tablou unidimensional: se citește numărul de elemente, se alocă dinamic tabloul și se citesc elementele tabloului.

```
main.c x
1  #include <stdio.h>
2  #include <stdlib.h>
3                                     n=5
4  int citire(int *v)                  10
5  {                                  20
6      int i,n;                       30
7      printf("n=");scanf("%d",&n);    40
8      v =(int *)malloc(n*sizeof(int)); 50
9      for (i=0;i<n;i++)
10         scanf("%d",&v[i]);
11         return n;
12     }
13
14
15  int main()
16  {
17      int n,*p=NULL;
18      n=citire(p);
19      int i;
20      for(i=0;i<n;i++)
21         printf("p[%d]=%d",i,p[i]);
22
23      return 0;
24  }
```

Process returned -1 (0xFFFFFFFF) execution time : 6.938 s  
Press ENTER to continue.

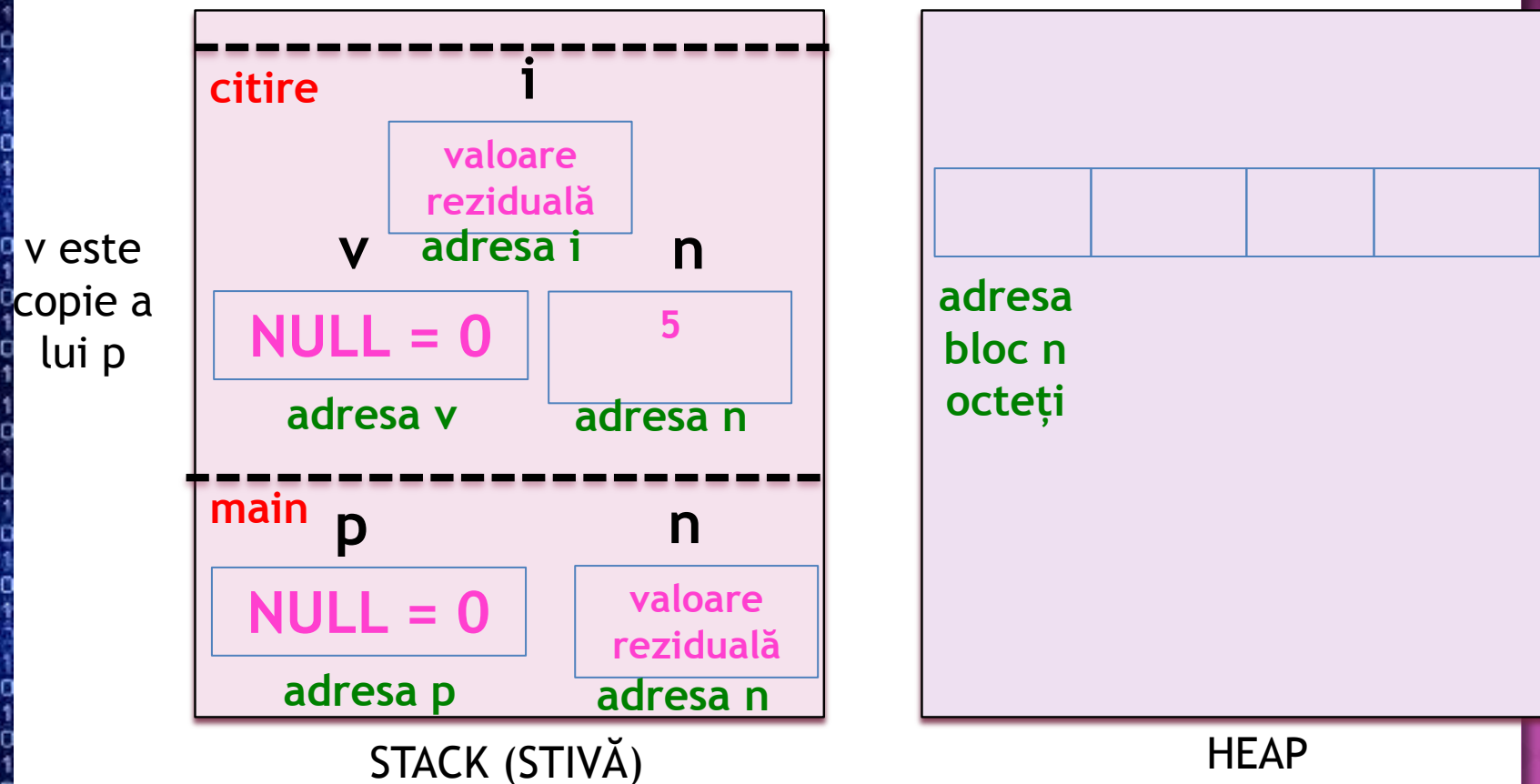
# FUNCȚIA MALLOC

Funcție pentru citirea unui tablou unidimensional: se citește numărul de elemente, se alocă dinamic tabloul și se citesc elementele tabloului.



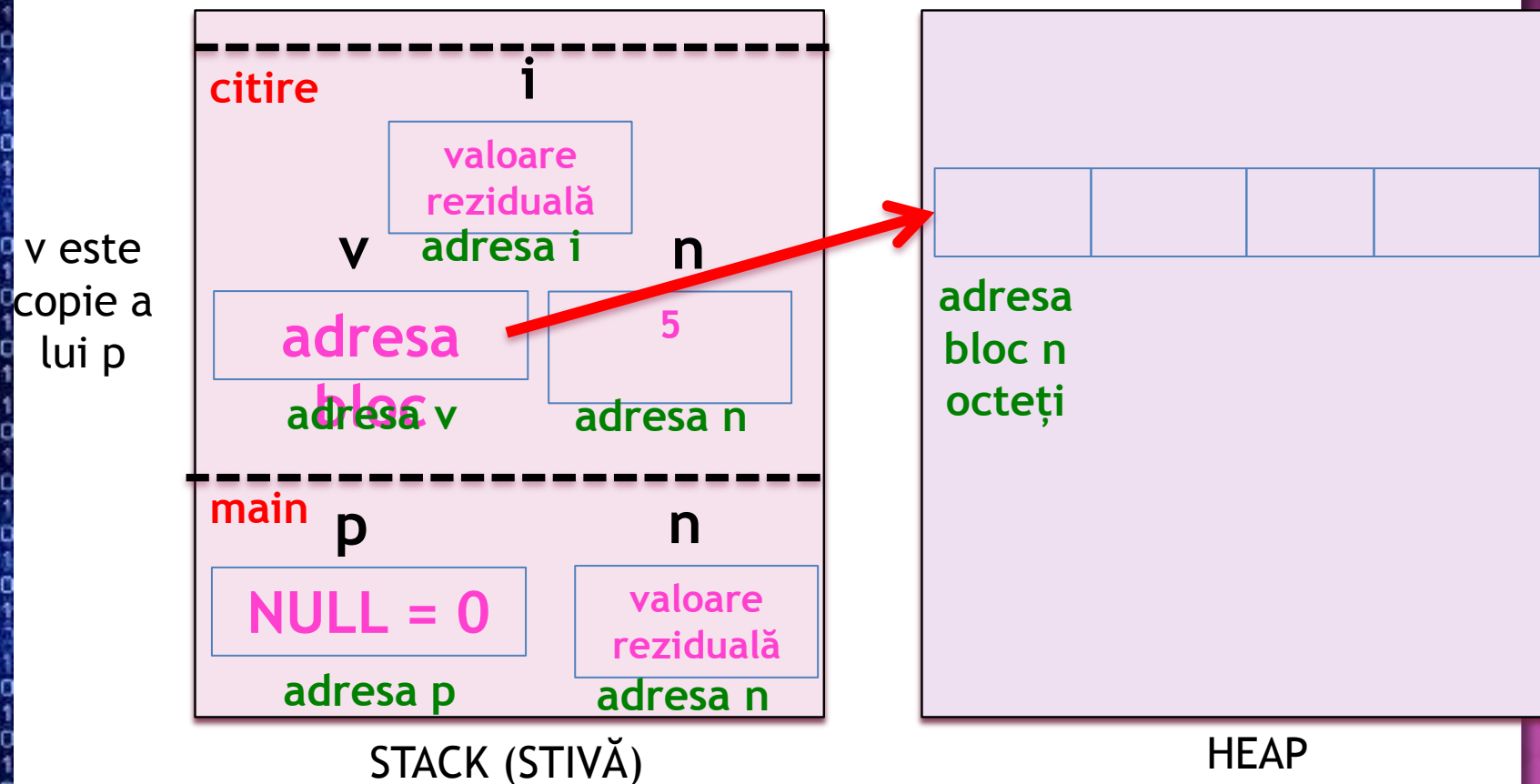
# FUNCȚIA MALLOC

Funcție pentru citirea unui tablou unidimensional: se citește numărul de elemente, se alocă dinamic tabloul și se citesc elementele tabloului.



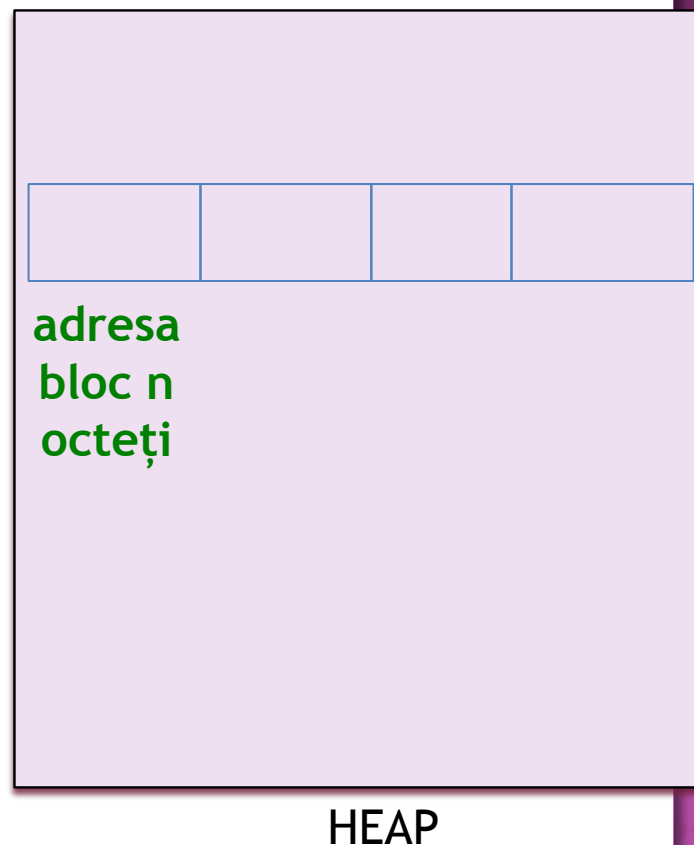
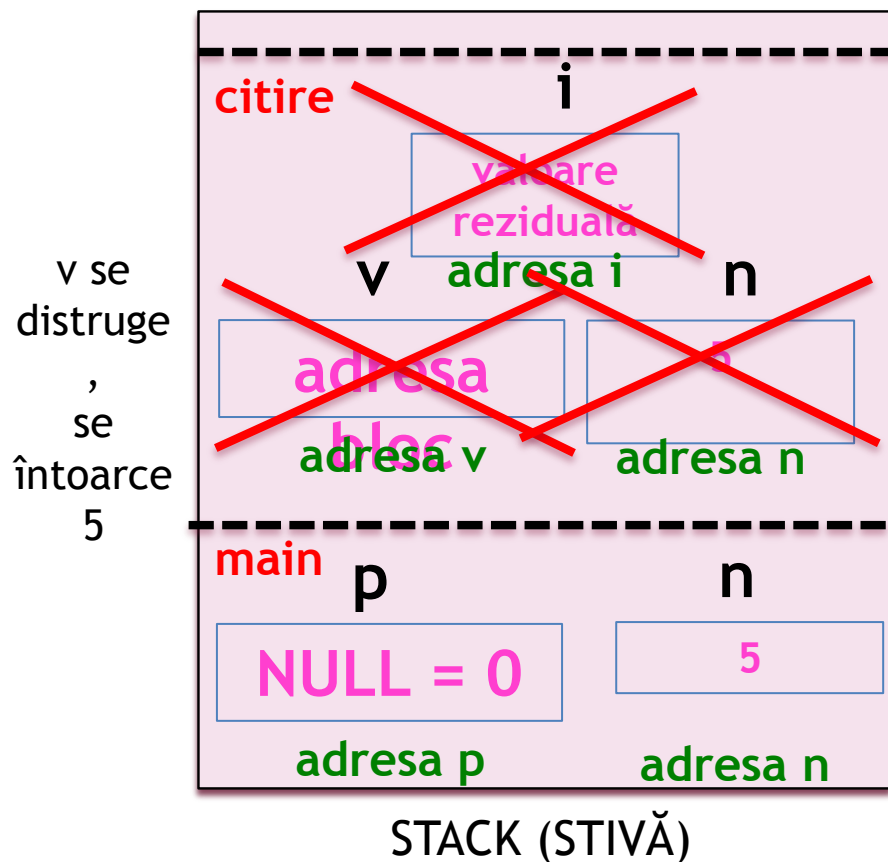
# FUNCȚIA MALLOC

Funcție pentru citirea unui tablou unidimensional: se citește numărul de elemente, se alocă dinamic tabloul și se citesc elementele tabloului.



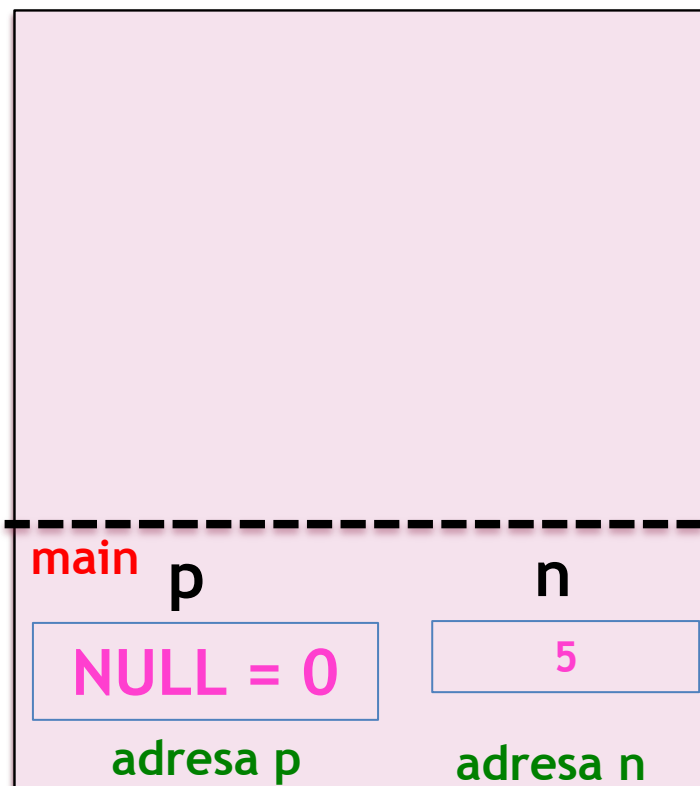
# FUNCȚIA MALLOC

Funcție pentru citirea unui tablou unidimensional: se citește numărul de elemente, se alocă dinamic tabloul și se citesc elementele tabloului.

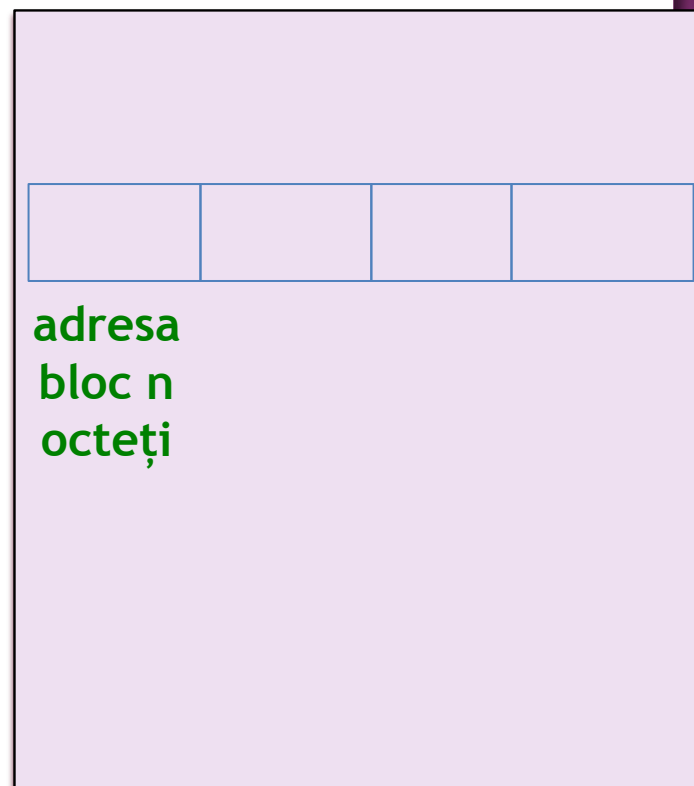


# FUNCȚIA MALLOC

Funcție pentru citirea unui tablou unidimensional: se citește numărul de elemente, se alocă dinamic tabloul și se citesc elementele tabloului.



STACK (STIVĂ)



HEAP



# FUNCȚIA MALLOC

Funcție pentru citirea unui tablou unidimensional: se citește numărul de elemente, se alocă dinamic tabloul și se citesc elementele tabloului.

```
main.c x
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int citire1(int **v)
5  {
6      int i,n;
7      printf("n=");scanf("%d",&n);
8      *v =(int *)malloc(n*sizeof(int));
9      for (i=0;i<n;i++)
10         scanf("%d",&(*v)[i]);
11     return n;
12 }
13
14
15 int main()
16 {
17     int n,*p=NULL;
18     n=citire1(&p);
19     int i;
20     for(i=0;i<n;i++)
21         printf("p[%d]=%d ",i,p[i]);
22
23     return 0;
24 }
```

n=5

10

20

30

40

50

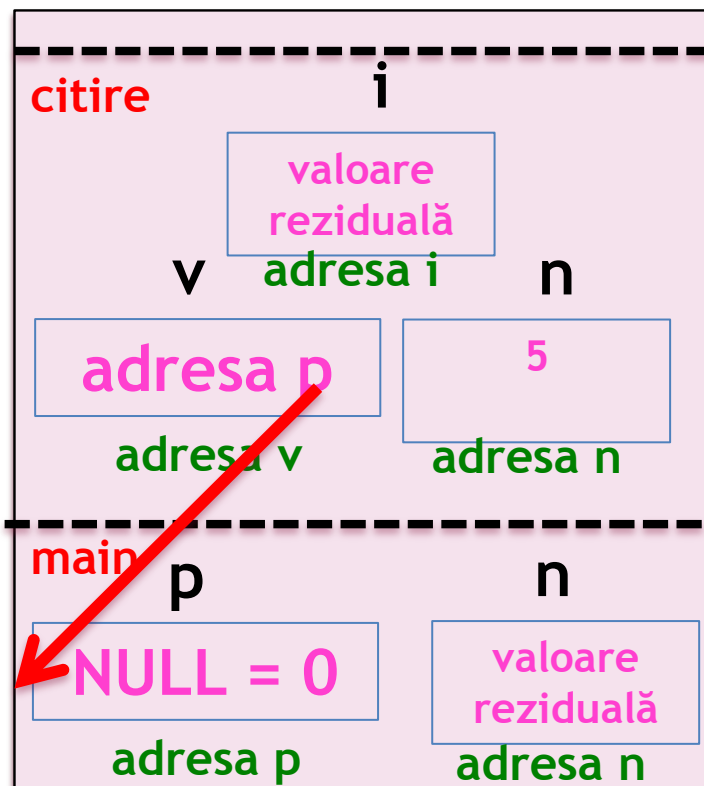
p[0]=10 p[1]=20 p[2]=30 p[3]=40 p[4]=50

Process returned 0 (0x0) execution time : 4.915

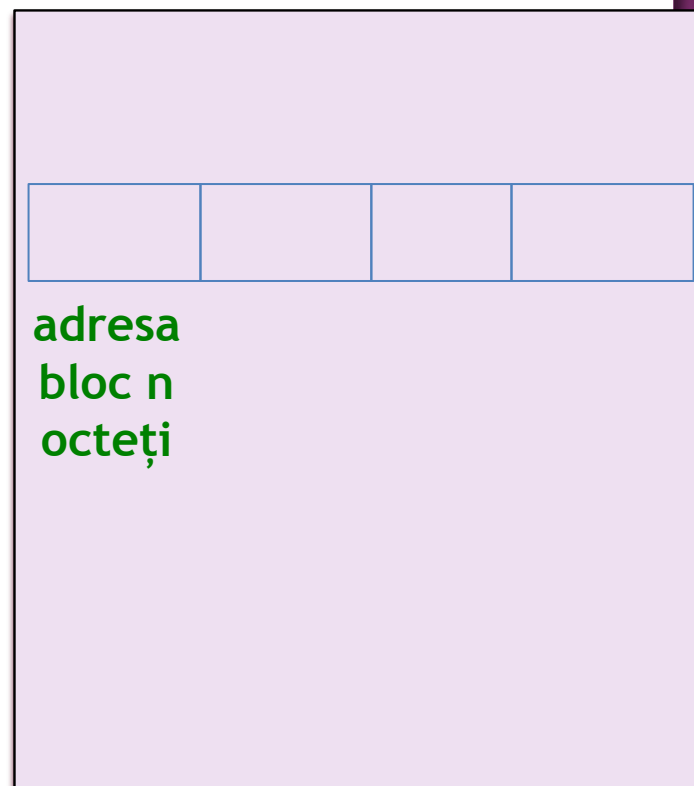
Press ENTER to continue.

# FUNCȚIA MALLOC

Funcție pentru citirea unui tablou unidimensional: se citește numărul de elemente, se alocă dinamic tabloul și se citesc elementele tabloului.



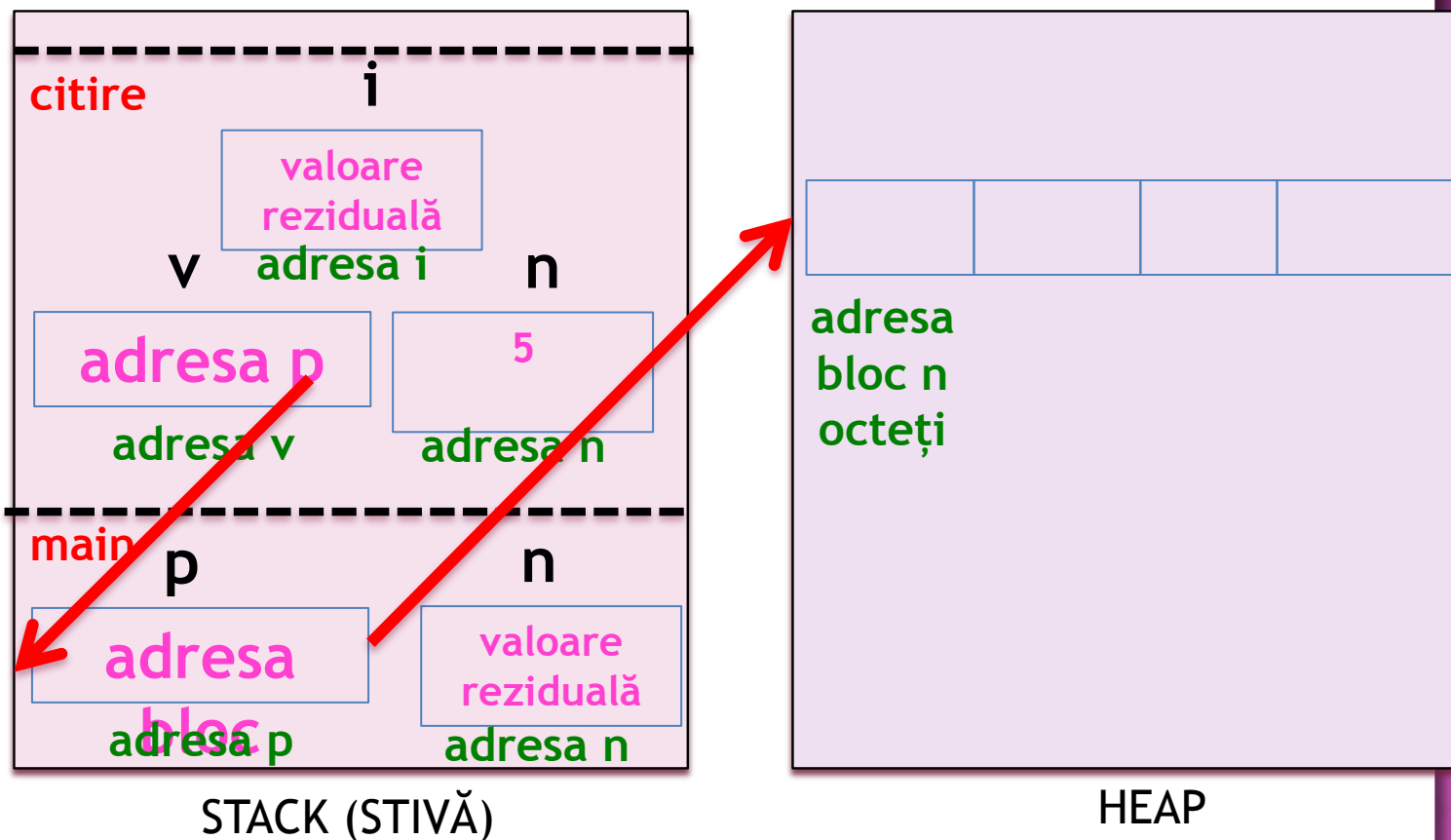
STACK (STIVĂ)



HEAP

# FUNCTIA MALLOC

- ❑ **exemplu:** scriu o funcție pentru citirea unui tablou unidimensional. În interiorul funcției citesc numărul  $n$  de elemente, aloc dinamic tabloul și citesc elementele tabloului.



# FUNCȚIA CALLOC

- ▣ prototipul funcției:

***void \* calloc( int numar, int dimensiune);***

unde:

- ▣ ***numar*** = numărul de blocuri/elemente a se aloca
- ▣ ***dimensiune*** = numărul de octeți ceruți pentru fiecare bloc
- ▣ dacă există suficient spațiu liber în HEAP atunci un bloc de memorie continuu de dimensiunea specificată va fi marcat ca ocupat, iar funcția **calloc** va returna un **pointer ce conține adresa de început a acelui bloc**. Dacă nu există suficient spațiu liber funcția **calloc** întoarce NULL.
- ▣ **diferența față de malloc**: funcția **calloc** inițializează toate blocurile cu 0.

# FUNCTIA CALLOC

□ exemplu;

exempluCalloc.c

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int main()
5  {
6
7      int n,i,*p1 = NULL;
8      double *p2 = NULL;
9      char *p3 = NULL;
10
11     scanf("%d",&n);
12
13     p1 = (int*) calloc(n,sizeof(int));
14     printf("\n Afisare adrese + valori vector de int alocat cu calloc \n");
15     for(i=0;i<n;i++)
16         printf("%x %d ", p1+i, p1[i]);
17
18     p2 = (double*) calloc(n,sizeof(double));
19     printf("\n Afisare adrese + valori vector de double alocat cu calloc \n");
20     for(i=0;i<n;i++)
21         printf("%x %f ", p2+i, p2[i]);
22
23     p3 = (char*) calloc(n,sizeof(char));
24     printf("\n Afisare adrese + valori vector de char alocat cu calloc \n");
25     for(i=0;i<n;i++)
26         printf("%x %d ", p3+i, p3[i]);
27     printf("\n");
28
29     return 0;
30 }
```

# FUNCTIA CALLOC

□ exemplu;

exempluCalloc.c

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int main()
5  {
6
7      int n,i,*p1 = NULL;
8      double *p2 = NULL;
9      char *p3 = NULL;
10
11     scanf("%d",&n);
12
13     p1 = (int*) calloc(n,sizeof(int));
14     printf("\n Afisare adrese + valori vector de int alocat cu calloc \n");
15     for(i=0;i<n;i++)
16         printf("%x %d ", p1+i, p1[i]);
17
18     p2 = (double*) calloc(n,sizeof(double));
19     printf("\n Afisare adrese + valori vector de double alocat cu calloc \n");
20     for(i=0;i<n;i++)
21         printf("%x %f ", p2+i, p2[i]);
22
23     p3 = 5
24     print
25     for(i=0;i<n;i++)
26         printf("%x %d ", p3+i, p3[i]);
27     print
28     Afisare adrese + valori vector de double alocat cu calloc
29     1000a0 0.000000 1000a8 0.000000 1000b0 0.000000 1000b8 0.000000 1000c0 0.000000
30     Afisare adrese + valori vector de char alocat cu calloc
31     100170 0 100171 0 100172 0 100173 0 100174 0
```



# FUNCȚIA REALLOC

## ▣ prototipul funcției:

***void \* realloc( void \*p, int dimensiune);***

unde:

- ▣ ***p*** reprezinta un pointer (începutul unui bloc de memorie pe care vreau să îl redimensionez (de obicei avem nevoie de mai multă memorie))
- ▣ ***dimensiune*** = numărul de octeți ceruți pentru alocare
- ▣ dacă există suficient spațiu liber în HEAP atunci un bloc de memorie continuu de dimensiunea specificată va fi marcat ca ocupat, iar funcția **realloc** va returna un pointer ce conține adresa de început a acelui bloc. Tot conținutul blocului de memorie inițial se copiază. Dacă nu există suficient spațiu liber **realloc** întoarce NULL.



# FUNCTIA REALLOC

## □ exemplu:

```
main.c x
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5
6      int *a,*aux;
7      a = (int*) malloc(100*sizeof(int));
8      if(!a)
9      {
10         printf("Nu pot aloca memorie");
11         exit(0);
12     }
13
14     aux = (int*) realloc(a,200*sizeof(int));
15     if(!aux)
16     {
17         printf("Nu pot redimensiona blocul a");
18         free(a);
19         exit(0);
20     }
21     else
22     {
23         printf("Redimensionare reusita \n");
24         a = aux;
25     }
26
27     free(a);
```

Redimensionare reusita

Process returned 0 (0x0)  
Press ENTER to continue.

execution time :

# FUNCȚIA FREE

- ▣ prototipul funcției:

***void free( void \*p);***

unde:

- ▣ ***p*** reprezintă un pointer (începutul unui bloc de memorie pe care vrem să-l eliberăm)
- ▣ funcția free eliberează zona de memorie alocată dinamic a cărei adresă de început este dată de ***p***. Zona de memorie dealocată este marcată ca fiind disponibilă pentru o nouă alocare.
- ▣ un bloc de memorie nu trebuie eliberat de mai multe ori.

# ALOCARE DINAMICĂ - APLICAȚII

- principalul avantaj al folosirii alocării dinamice este gestionarea eficientă a resurselor memoriei. Memoria necesară este alocată în timpul execuției programului (când e nevoie) și nu la compilarea programului
- **exemplu:** se citește de la tastatură un număr n și apoi n numere întregi. Să se afișeze numerele în ordinea inversă a citirii.

```
main.c x
1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main(){
6      int *p,n,i;
7      printf("n=");scanf("%d",&n);
8      p = (int*) malloc(n*sizeof(int));
9      for(i=0;i<n;i++)
10         scanf("%d",p+i);
11      for(i=n-1;i>=0;i--)
12         printf("%d ",*(p+i));
13      return 0;
14 }
```

```
n=5
10
20
30
40
50
50 40 30 20 10
Process returned 0 (0x0)
Press ENTER to continue.
```

execution time : 6.139 s

# ALOCARE DINAMICĂ - APLICAȚII

- ❑ **exemplu:** se citește de la tastatură un șir de numere întregi până la întâlnirea lui 0. Să se afișeze numerele în ordinea inversă a citirii.

```
main.c x
1  #include <stdio.h>
2  #include <stdlib.h>
3  void afisare(int *p, int dim)
4  {
5      int i;
6      printf("\nDupa %d realocari: ",dim);
7      for(i=dim-1;i>=0;i--)
8          printf("%d\t",*(p+i));
9  }
10 int main(){
11     int *p,*aux,i,valoareCitita;
12     printf("Dati numarul:");
13     scanf("%d",&valoareCitita);
14     p = (int*) malloc(sizeof(int));
15     i = 0;
16     while(valoareCitita!=0)
17     {
18         p[i] = valoareCitita;
19         afisare(p,i+1);
20         i++;
21         p = realloc(p,(i+1)*sizeof(int));
22         printf("\nDati un alt numar:");
23         scanf("%d",&valoareCitita);
24     }
25     free(p);
26     return 0;
27 }
```

Ce se întâmplă dacă nu pot  
să realoc memorie?  
p devine NULL (am pierdut  
tot conținutul de până  
atunci).

# ALOCARE DINAMICĂ - APLICAȚII

- ❑ **exemplu:** se citește de la tastatură un șir de numere întregi până la întâlnirea lui 0. Să se afișeze numerele în ordinea inversă a citirii.

```
main.c
1  #include <stdio.h>
2  #include <stdlib.h>
3  void afisare(int *p, int dim)
4  {
5      int i;
6      printf("\nDupa %d realocari: ", dim);
7      for(i=dim-1; i>=0; i--)
8          printf("%d\t", *(p+i));
9  }
10 int main(){
11     int *p, *aux, i, valoareCitita;
12     printf("Dati numarul:");
13     scanf("%d", &valoareCitita);
14     p = (int*) malloc(sizeof(int));
15     i = 0;
16     while(valoareCitita!=0)
17     {
18         p[i] = valoareCitita;
19         afisare(p, i+1);
20         i++;
21         aux = realloc(p, (i+1)*sizeof(int));
22         if(aux)
23             p = aux;
24         else
25         {
26             printf("Eroare la realocare\n");
27             free(p); exit(0);
28         }
29         printf("\nDati un alt numar:");
30         scanf("%d", &valoareCitita);
31     }
32     free(p);
```

Dati numarul:10

Dupa 1 realocari: 10

Dati un alt numar:20

Dupa 2 realocari: 20      10

Dati un alt numar:30

Dupa 3 realocari: 30      20      10

Dati un alt numar:40

Dupa 4 realocari: 40      30      20      10

Dati un alt numar:50

Dupa 5 realocari: 50      40      30      20      10

Dati un alt numar:0

Process returned 0 (0x0)      execution time : 9.421 s

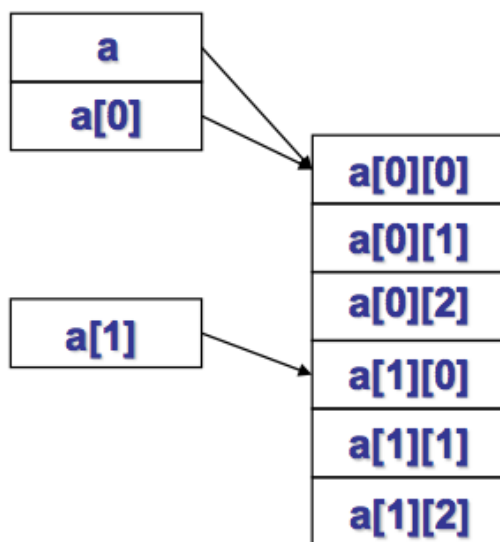
Press ENTER to continue

# ALOCARE DINAMICĂ - APLICAȚII

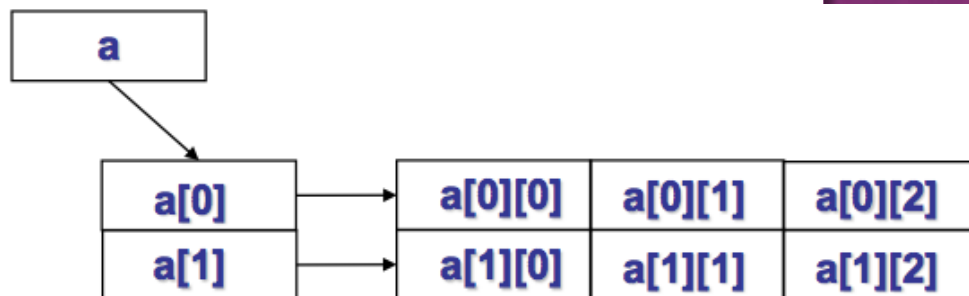
- alocarea dinamică a unui tablou bi-dimensional

## Alocarea statică (pe STIVĂ)

```
int a[2][3];
```



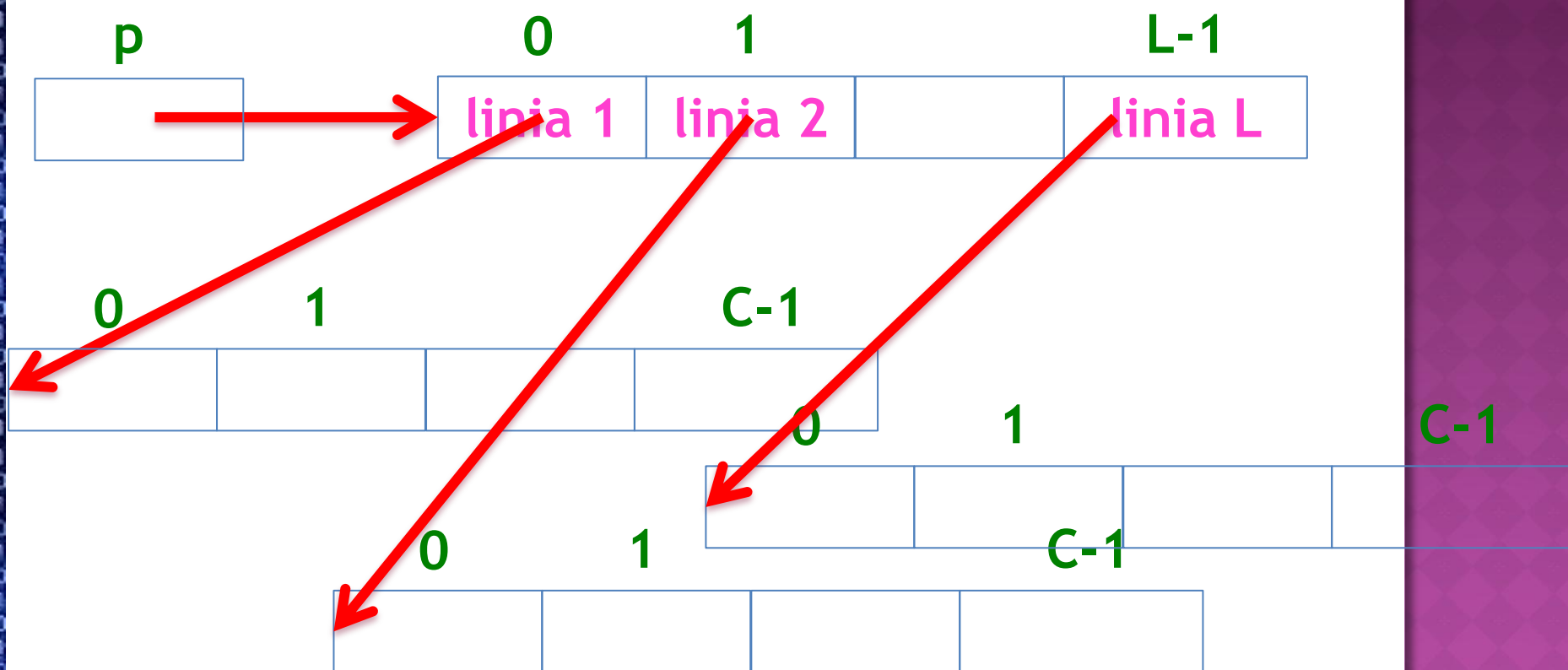
## Alocarea dinamică (pe HEAP)



`a` e pointer dublu

# ALOCARE DINAMICĂ - APLICAȚII

- alocarea dinamică a unui tablou bi-dimensional





# ALOCARE DINAMICĂ - APLICAȚII

❑ **exemplu:** alocarea dinamică a unui tablou bi-dimensional

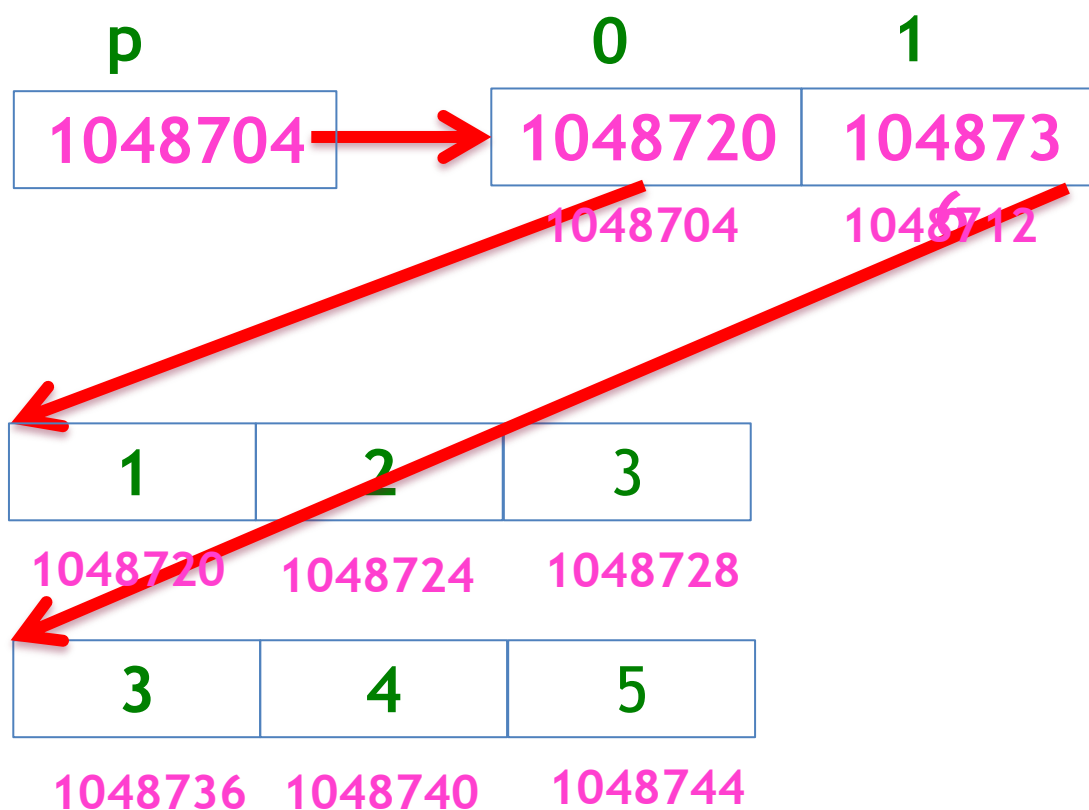
tablou\_bidimensional.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      int L, C, i, j;
6      int **p; // Adresa matrice
7
8      printf("Nr de linii L = "); scanf("%d", &L);
9      printf("Nr de coloane C = "); scanf("%d", &C);
10
11     p = (int**) malloc(L * sizeof(int*));
12     printf("Sizeof(int*) = %d \n", sizeof(int*));
13     printf("Pointerul p contine adresa %d \n", p);
14
15     for (i = 0; i < L; i++)
16     {
17         p[i] = calloc(C, sizeof(int));
18         printf("Linia %d incepe la %d \n", i, p[i]);
19     }
20
21     for (i = 0; i < L; i++) {
22         for (j = 0; j < C; j++) {
23             p[i][j] = L * i + j + 1;
24             printf("Adresa lui p[%d][%d] este = %d \n", i, j, &p[i][j]);
25         }
26     }
27
28     for (i = 0; i < L; i++) {
29         for (j = 0; j < C; j++) {
30             printf("%d ", p[i][j]);
31         }
32         printf("\n");
33     }
```

Nr de linii L = 2  
Nr de coloane C = 3  
Sizeof(int\*) = 8  
Pointerul p contine adresa 1048704  
Linia 0 incepe la 1048720  
Linia 1 incepe la 1048736  
Adresa lui p[0][0] este = 1048720  
Adresa lui p[0][1] este = 1048724  
Adresa lui p[0][2] este = 1048728  
Adresa lui p[1][0] este = 1048736  
Adresa lui p[1][1] este = 1048740  
Adresa lui p[1][2] este = 1048744  
1 2 3  
3 4 5

# ALOCARE DINAMICĂ - APLICAȚII

□ **exemplu:** alocarea dinamică a unui tablou bi-dimensional



```
Nr de linii L = 2
Nr de coloane C = 3
Sizeof(int*) = 8
Pointerul p contine adresa 1048704
Linia 0 incepe la 1048720
Linia 1 incepe la 1048736
Adresa lui p[0][0] este = 1048720
Adresa lui p[0][1] este = 1048724
Adresa lui p[0][2] este = 1048728
Adresa lui p[1][0] este = 1048736
Adresa lui p[1][1] este = 1048740
Adresa lui p[1][2] este = 1048744
1 2 3
3 4 5
```

# ALOCARE DINAMICĂ - AVANTAJE + DEZAVANTAJE

## ❑ **avantaje:**

- ❑ durată de viață: putem controla când are loc alocarea și dealocarea memoriei
- ❑ memorie: dimensiunea memoriei alocată poate fi controlată în timpul execuției programului. Spre exemplu un tablou poate fi alocat astfel încât are să aibă dimensiunea identică cu cea a unui tablou specificat în timpul execuției programului

## ❑ **dezavantaje:**

- ❑ mai mult de codat: alocarea memoriei trebuie făcută explicit în cod
- ❑ posibile bug-uri: lucrul cu pointerii (crash-uri de memorie)

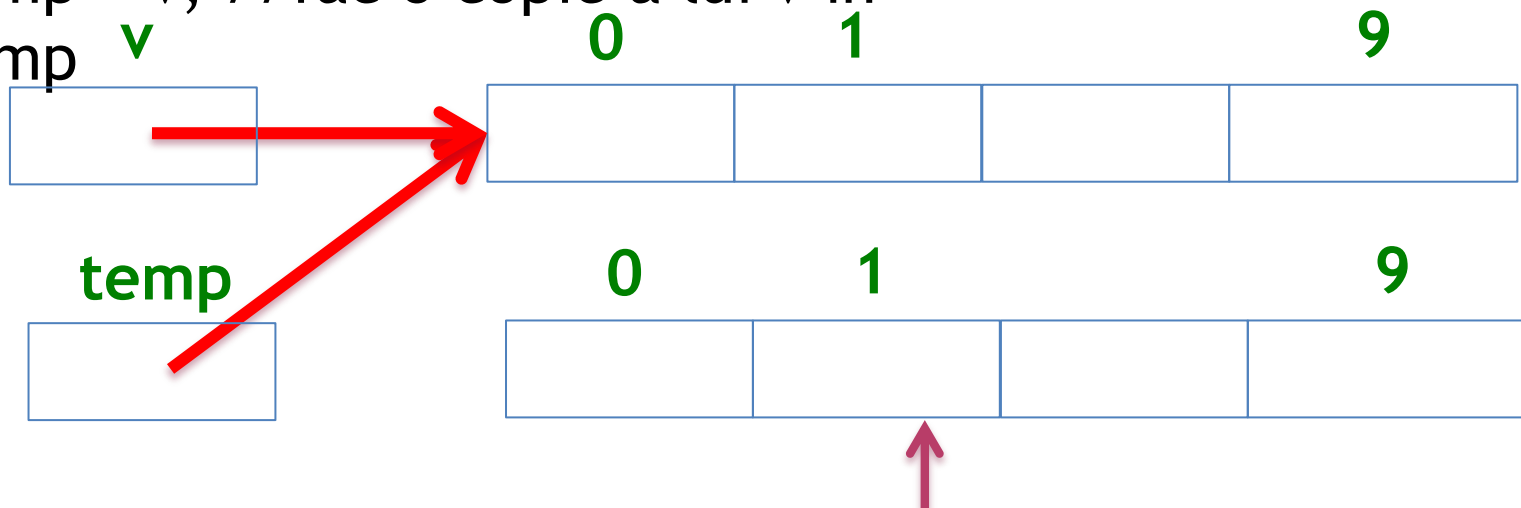
# ALOCARE DINAMICĂ - GREȘELI

```
int *v, *temp;  
v = (int*) malloc(10*sizeof(int));  
temp = (int*)  
malloc(10*sizeof(int));  
temp = v; //fac o copie a lui v in  
temp
```



# ALOCARE DINAMICĂ - GREȘELI

```
int *v, *temp;  
v = (int*) malloc(10*sizeof(int));  
temp = (int*)  
malloc(10*sizeof(int));  
temp = v; //fac o copie a lui v în  
temp
```



Zonă marcată de sistemul de operare ca fiind ocupată dar inutilizabilă întrucât am “pierdut” adresa de început a blocului.  
(zonă orfană de memorie)

# ALOCARE DINAMICĂ - GREȘELI

```
void f(...){  
    int *p = (int*) malloc(10*sizeof(int));  
    ...  
}
```



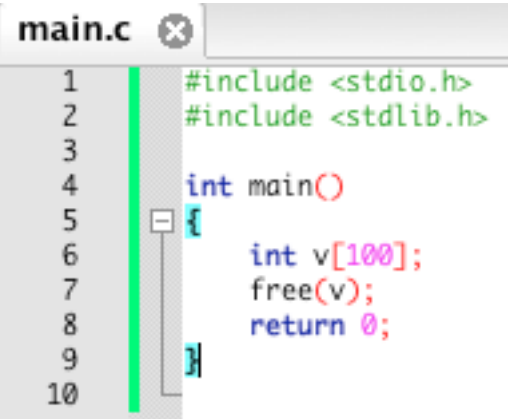
p este variabilă locală funcției f și va fi distrusă la ieșirea din funcție. Totuși memoria rămâne alocată și inutilizabilă (zonă orfană de memorie).

```
void f(...){  
    int *p = (int*) malloc(10*sizeof(int));  
    free(p); //eliberare memorie  
}
```

# ALOCARE DINAMICĂ - GREȘELI

```
int v[200];  
free(v);
```

v e alocat static, pot elibera cu functia free numai blocuri de memorie alocate dinamic



The screenshot shows a code editor window titled 'main.c'. The code is as follows:

```
1 #include <stdio.h>  
2 #include <stdlib.h>  
3  
4 int main()  
5 {  
6     int v[100];  
7     free(v);  
8     return 0;  
9 }  
10
```

```
(12054) malloc: *** error for object 0x7fff5fbff7d0: pointer being freed was  
allocated  
at a breakpoint in malloc_error_break to debug  
  
ss returned -1 (0xFFFFFFFF)   execution time : 0.053 s  
ENTER to continue.
```