



# PROGRAMARE PROCEDURALĂ

Bogdan Alexe

[bogdan.alexe@fmi.unibuc.ro](mailto:bogdan.alexe@fmi.unibuc.ro)

Secția Informatică, anul I,  
2016-2017

Cursul 6

# Cursul trecut

1. Pointeri
2. Funcții de citire/scriere
3. Tablouri. Siruri de caractere.
4. Structuri, câmpuri de biți, uniuni.

# 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. Siruri de caractere.
- Structuri, uniuni, câmpuri de biți, enumerări.
- Pointeri.

## ❑ Funcții (1)

- Declarație și definire. Apel. Metode de transmisie a parametrilor.
- Pointeri la funcții.

## ❑ Tablouri și pointeri

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

## ❑ Siruri 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.
- Programare generică.

## ❑ Recursivitate

# Cursul de azi

1. Enumerări, typedef.
2. Funcții: declarare și definire, apel, metode transmitere a parametrilor.
3. Pointeri la funcții.
4. Legătura dintre tablouri și pointeri.
5. Aritmetică pointerilor.

# Tipuri de date structurate

Limbajul C permite creare tipurilor de date în 5 moduri:

- structura (**struct**) – grupează mai multe variabile sub același nume;
- câmpul de biți (variațiune a structurii) → acces ușor la bitii individuali
- uniunea (**union**) – face posibil ca aceleași zone de memorie să fie definite ca două sau mai multe tipuri diferite de variabile
- enumerarea (**enum**) – listă de constante întregi cu nume
- tipuri definite de utilizator (**typedef**) – definește un nou nume pentru un tip existent

# Enumerări

- ❑ multime de constante de tip întreg care specifică toate valorile permise pe care le poate avea o variabilă de acel tip
- ❑ atât numele generic al enumerării cât și lista de variabile sunt optionale
- ❑ constanta unui element al enumerării este fie asociată implicit, fie explicit. Implicit, primul element are asociată valoarea 0, iar pentru restul este valoarea precedentă+1.

## ❑ **sintaxa:**

```
enum <nume> {  
    lista enumerarilor  
} lista variabile;
```

# Enumerări

## ❑ exemplu

enum {a, b, c, d}; → a = 0, b = 1, c = 2, d = 3

enum {a, b, c=7, d}; → a = 0, b = 1, c = 7, d = 8

enum {a=4, b=-3, c=9, d=-8}; enum {false,true};

```
main.c ✘
1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main(){
6
7      enum zile{
8          luni,
9          marti,
10         miercuri,
11         joi,
12         vineri,
13         sambata,
14         duminica};
15
16         printf("Azi e ziua a %d-a a saptamanii \n", joi);
17
18     return 0;
19 }
```

Azi e ziua a 3-a a saptamanii  
Process returned 0 (0x0) execution time : 0.00  
Press ENTER to continue.

# Enumerări

## □ exemplu

enum {a, b, c, d}; → a = 0, b = 1, c = 2, d = 3

enum {a, b, c=7, d}; → a = 0, b = 1, c = 7, d = 8

enum {a=4, b=-3, c=9, d=-8}; enum {false,true};

```
main.c x
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5     enum zile{
6         luni=1,
7         marti,
8         miercuri,
9         joi,
10        vineri,
11        sambata,
12        duminica};
13
14     printf("Azi e ziua a %d-a a saptamanii \n",joi);
15
16
17
18     return 0;
19 }
```

Azi e ziua a 4-a a saptamanii  
Process returned 0 (0x0) execution time :  
Press ENTER to continue.

# Enumerări

## ❑ exemplu

main.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4
5 int main(){
6
7     enum zile{
8         luni=1,
9         marti,
10        miercuri,
11        joi,
12        vineri,
13        sambata,
14        duminica} azi,maine;
15
16
17     azi = joi;
18     printf("Azi e ziua a %d-a a saptamanii \n",azi);
19     maine = azi+1;
20     printf("Maine va fi a %d-a a saptamanii \n",maine);
21
22
23
24 }
```

Azi e ziua a 4-a a saptamanii  
Maine va fi a 5-a a saptamanii

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

# Specificatorul `typedef`

- definește explicit noi tipuri de date.
- nu se declară o variabilă sau o funcție de un anumit tip, ci se asociază un nume (sinonimul) tipului de date.
- **sintaxa:**  
`typedef <definiție tip> <identificator>;`
- **exemple:**

```
typedef unsigned int natural;  
typedef long double tablouNumereReale [100] ;  
tablouNumereReale a, b, c;  
natural m,n,i;
```

# Specificatorul `typedef`

ex\_typedef.c

```
1 #include <stdio.h>
2
3 int main()
4 {
5     typedef long double tablouNumereReale[100] ;
6     tablouNumereReale a;
7     printf("dimensiunea lui a este %d\n",sizeof(a));
8     long double d;
9     printf("dimensiunea lui d este %d\n",sizeof(d));
10
11     return 0;
12 }
```

```
.....,.....,.....,.....
dimensiunea lui a este 1600
dimensiunea lui d este 16
Bogdan Alexeev MacBook Pro: ~ boggie$
```

# Cursul de azi

1. Enumerări, typedef.
2. Funcții: declarare și definire, apel, metode transmitere a parametrilor.
3. Pointeri la funcții.
4. Legătura dintre tablouri și pointeri
5. Aritmetică pointerilor

# Functii

- permit modularizarea programelor
  - variabilele declarate în interiorul funcțiilor – variabile locale (vizibile doar în interior)
- parametri funcțiilor
  - permit comunicarea informației între funcții
  - sunt variabile locale funcțiilor
- avantajele utilizării funcțiilor
  - divizarea problemei în subprobleme
  - managementul dezvoltării programelor
  - utilizarea/reutilizarea funcțiilor scrise în alte programe
  - elimină duplicarea codului scris

# Functii

- o funcție = bloc de instrucțiuni care nu se poate executa de sine stătător ci trebuie apelat.

- sintaxa:

**tip\_returnat nume\_functie (lista parametrilor formali)**

```
{     variabile locale  
       instructiuni;  
       return expresie;  
}
```



antetul funcției  
(declarare)

corpus funcției  
(definire)

- lista de parametri formalii poate fi reprezentata de:
  - nici un parametru:
    - **tip\_returnat nume\_functie ()**
    - **tip\_returnat nume\_functie (void)**
  - unul sau mai mulți parametri separați prin virgulă.

# Valoarea returnată de o funcție

- două categorii de funcții:
  - care returnează o valoare: prin utilizarea instrucțiunii **return expresie**;
  - care nu returnează o valoare: prin instrucțiunea **return**; (tipul returnat este void)
- returnarea valorii
  - poate returna orice tip standard (**void**, **char**, **int**, **float**, **double**) sau definit de utilizator (structuri, uniuni, enumerari, **typedef**)
  - declarațiile și instrucțiunile din funcții sunt executate până se întâlnește
    - instrucțiunea **return**
    - accolada închisă **}** - execuția atinge finalul funcției

# Valoarea returnată de o funcție

```
double f(double t)
{
    return t-1.5;
}
```

← definire de funcție

```
float g(int);
```

← declarație de funcție

```
int main()
{
    float a=11.5;
    printf("%f\n", f(a));
    printf("%f\n", g(a));
}
```

**Rezultat afișat**

10.000000

13.000000

```
float g(int z)
{
    return z+2.0;
}
```

← definire de funcție

# Prototipul și argumentele funcțiilor

- **prototipul** unei funcții (declararea ei) constă în specificarea antetului urmat de caracterul ;
  - nu este necesară specificarea numelor parametrilor formali  
**int adunare(int, int);**
  - este necesară inserarea prototipului unei funcții înaintea altor funcții în care este invocată dacă definirea ei este localizată după definirea acelor funcții
- **parametri** apar în definiții
- **argumentele** apar în apelurile de funcții
  - corespondența între parametrii formali (definiția funcției) și actuali (apelul funcției) este **pozițională**
  - regula de **conversie a argumentelor**
    - în cazul în care diferă, tipul fiecărui argument este convertit automat la tipul parametrului formal corespunzător (ca și în cazul unei simple atribuirii)

# Prototipul și argumentele funcțiilor

```
1 #include <stdio.h>
2
3 void f(char a)
4 {
5     printf("%d\n", a);
6 }
7
8 int main()
9 {
10    int a = 300;
11    float b = 305.7;
12    f(a);
13    f(b);
14    return 0;
15 }
```

Rezultatul afișat

44  
49

# Fișiere header cu extensia .h

- conțin prototipuri de funcții
- bibliotecile standard
  - conțin prototipuri de funcții standard regăsite în fișierele *header* corespunzătoare (ex. **stdio.h**, **stdlib.h**, **math.h**, **string.h**)
  - exemplu – biblioteca **stdio.h** care conține și prototipul funcției
  - **printf: int printf(const char\* format, ...);**
  - se încarcă cu **#include <filename.h>**
- biblioteci utilizator
  - conțin prototipuri de funcții și macrouri
  - se pot salva ca fișiere cu extensia *.h* : ex. **filename.h**
  - se încarcă cu **#include “filename.h”**

# Transmiterea parametrilor către funcții

- ❑ utilizată la apelul funcțiilor
- ❑ În limbajul C transmiterea parametrilor se poate face doar prin **valoare (pass-by-value)**
  - ❑ o copie a argumentelor este trimisă funcției
  - ❑ modificările în interiorul funcției nu afectează argumentele originale
- ❑ În limbajul C++ transmiterea parametrilor apelul se poate face și prin **referință (pass-by-reference)**
  - ❑ argumentele originale sunt trimise funcției
  - ❑ modificările în interiorul funcției afectează argumentele trimise

```
1 #include <stdio.h>
2
3 void interschimba(int x, int y)
4 {
5     int aux = x; x = y; y = aux;
6 }
7
8 void interschimba2(int& x, int& y)
9 {
10    int aux = x; x = y; y = aux;
11 }
12
13 void interschimba3(int* x, int* y)
14 {
15    int aux = *x; *x = *y; *y = aux;
16 }
17
18 int main()
19 {
20     int x=10, y =15;
21     interschimba(x,y);
22     printf("x = %d, y = %d \n",x,y);
23     x=10, y =15;
24     interschimba2(x,y);
25     printf("x = %d, y = %d \n",x,y);
26     x=10, y =15;
27     interschimba3(&x,&y);
28     printf("x = %d, y = %d \n",x,y);
29     return 0;
30 }
```

x = 10, y = 15  
x = 15, y = 10  
x = 15, y = 10

apel prin valoare

apel prin referință  
numai în C++

apel prin valoare

# Transmiterea parametrilor către funcții

- ❑ utilizat la apelul funcțiilor
- ❑ În limbajul C transmiterea parametrilor se poate face doar prin **valoare (pass-by-value)**
  - ❑ o copie a argumentelor este trimisă funcției
  - ❑ modificările în interiorul funcției nu afectează argumentele originale
- ❑ pentru modificarea parametrilor actuali, funcției i se transmit nu valorile parametrilor actuali, ci **adresele lor (pass by pointer)**. Funcția face o copie a adresei dar prin intermediul ei lucrează cu variabila “reală” (zona de memorie “reală”). Astfel **putem simula în C transmiterea prin referință cu ajutorul pointerelor.**

# Transmiterea parametrilor către funcții

```
main.c ✘
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int f1(int a, int b)
5 {
6     a++;
7     b++;
8     printf("In functia f1 avem a=%d,b=%d\n",a,b);
9     return a+b;
10}
11
12 int main(){
13     int a = 5, b= 8;
14     int c = f1(a,b);
15     printf("In functia main avem a=%d,b=%d,c=%d\n",a,b,c);
16     return 0;
17}
18
```

```
In functia f1 avem a=6,b=9
In functia main avem a=5,b=8,c=15
```

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

# Transmiterea parametrilor către funcții

```
main.c ✘
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int f1(int a, int b)
5 {
6     a++;
7     b++;
8     printf("In functia f1 avem a=%d,b=%d\n",a,b);
9     return a+b;
10}
11
12 int f2(int*a, int b)
13 {
14     *a = *a + 1;
15     b++;
16     printf("In functia f2 avem *a=%d,b=%d\n",*a,b);
17     return *a+b;
18}
19
20 int main(){
21     int a = 5, b= 8;
22     int c = f2(&a,b);
23     printf("In functia main avem a=%d,b=%d,c=%d\n",a,b,c);
24     return 0;
25}
```

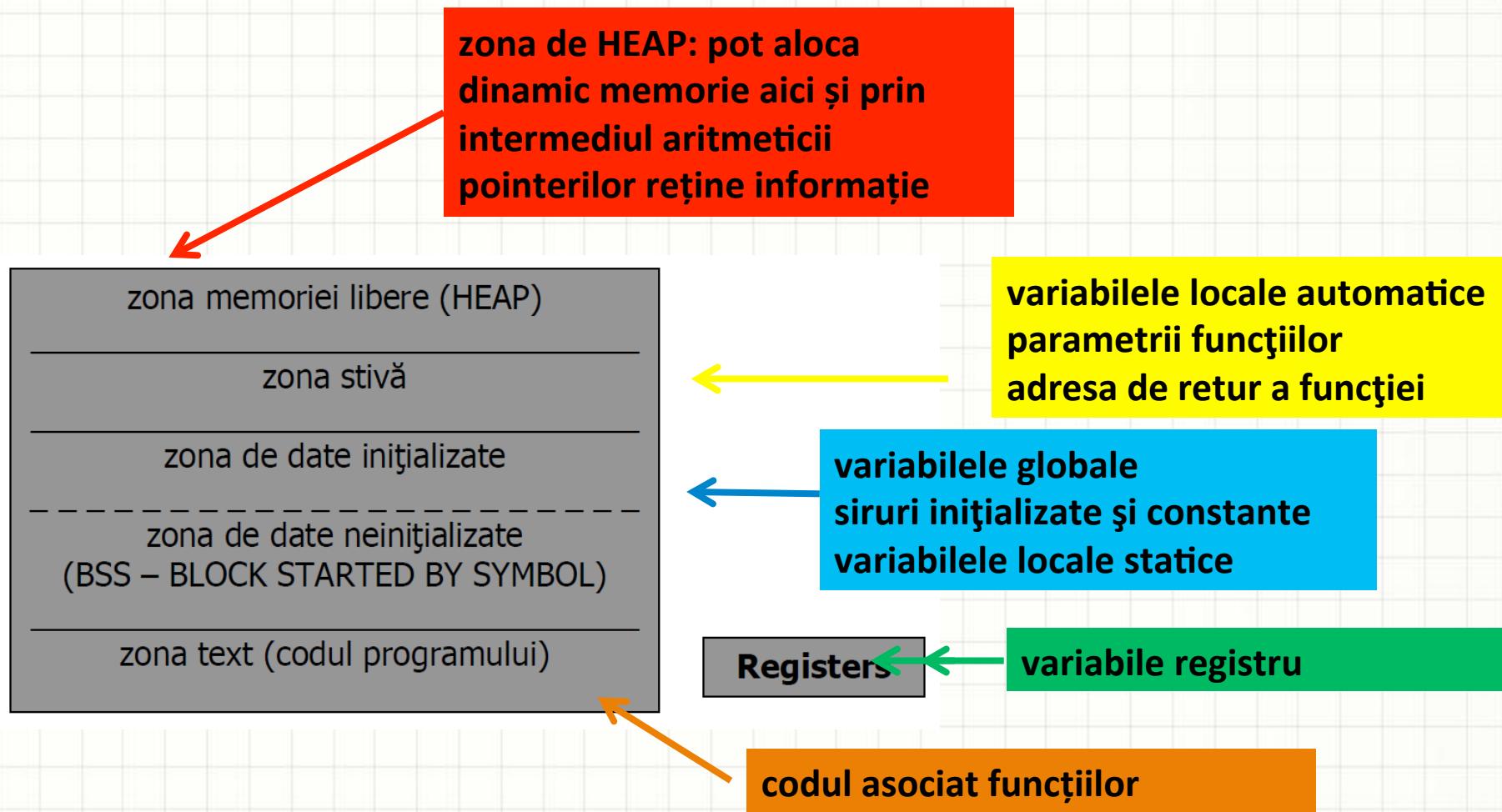
# Transmiterea parametrilor către funcții

```
main.c ✘
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int f1(int a, int b)
5 {
6     a++;
7     b++;
8     printf("In functia f1 avem a=%d,b=%d\n",a,b);
9     return a+b;
10}
11
12 int f2(int*a, int b)
13 {
14     *a = *a + 1;
15     b++;
16     printf("In functia f2 avem *a=%d,b=%d\n",*a,b);
17     return *a+b;
18}
19
20 int main(){
21     int a = 5, b= 8;           In functia f2 avem *a=6,b=9
22     int c = f2(&a,b);        In functia main avem a=6,b=8,c=15
23     printf("In functia main avem a=%d
24     return 0;                Process returned 0 (0x0)    execution t
25 }
```

# Apelul funcției și revenirea din apel

- etapele principale ale apelului unei funcție și a revenirii din acesta în funcția de unde a fost apelată:
  - argumentele apelului sunt evaluate și trimise funcției
  - adresa de revenire este salvată pe stivă
  - controlul trece la funcția care este apelată
  - funcția apelată alocă pe **stivă** spațiu pentru variabilele locale și pentru cele temporare
  - se execută instrucțiunile din corpul funcției
  - dacă există valoare returnată, aceasta este pusă într-un loc sigur
  - spațiul alocat pe stivă este eliberat
  - utilizând adresa de revenire controlul este transferat în funcția care a inițiat apelul, după acesta

# Harta simplificată a memoriei la rularea unui program



# Harta simplificată a memoriei la rularea unui program

hartaMemorie.c

```
01 #include <stdio.h>
02 // variabile globale neinitializate
03 int g1,g2;
04 // variabile globale initialize
05 int g3=5, g4 = 7;
06 int g5, g6;
07
08 void f1() {
09     int var1,var2;
10     printf("In Stiva prin f1:\t\t %p %p\n",&var1,&var2);
11 }
12
13 void f2() {
14     int var1,var2;
15     printf("In Stiva prin f2:\t\t %p %p\n",&var1,&var2);
16     f1();
17 }
18
19 int main() {
20     //variabile locale
21     int var1,var2;
22     printf("In Stiva prin main:\t\t %p %p\n",&var1,&var2);
23     f2();
24     // variabile globale initialize + neinitializate
25     printf("Variabile globale neinitializate:\t\t %p %p\n",&g1,&g2);
26     printf("Variabile globale initialize: \t\t %p %p\n",&g3,&g4);
27     printf("Variabile globale neinitializate:\t\t %p %p\n",&g5,&g6);
28     //cod
29     printf("Text Data:\t\t\t\t\t %p %p \n\n",main,f1);
30 }
31 }
```

# Harta simplificată a memoriei la rularea unui program

hartaMemorie.c

```
01 #include <stdio.h>
02 // variabile globale neinitialize
03 int g1,g2;
04 // variabile globale initialize
05 int g3=5, g4 = 7;
06 int g5, g6;
07
08 void f1() {
09     int var1,var2;
10     printf("In Stiva prin f1:\t\t %p %p\n",&var1,&var2);
11 }
12
13 void f2() {
14     int var1,var2;
15     printf("In Stiva prin f2:\t\t %p %p\n",&var1,&var2);
16     f1();
17 }
18
19 int main() {
20     //variabile locale
21     int var1,var2;
22     printf("In Stiva prin main:\t\t %p %p\n",&var1,&var2);
23     f2();
24     // variabili In Stiva prin main:          0xffff5fbff95c 0xffff5fbff958
25     printf("Val In Stiva prin f2:            0xffff5fbff93c 0xffff5fbff938
26     printf("Val In Stiva prin f1:            0xffff5fbff91c 0xffff5fbff918
27     printf("Val Variabile globale neinitialize: 0x100001070 0x100001074
28     //cod      Variabile globale initialize: 0x100001068 0x10000106c
29     printf("Text Variabile globale neinitialize: 0x100001078 0x10000107c
30     return 0;  Text Data:                  0x100000d54 0x100000d04
31 }
```

# Stiva în C

- ❑ la execuția programelor C se utilizează o structură internă numită **stivă** și care este utilizată pentru alocarea memoriei și manipularea variabilelor temporare
- ❑ pe stivă sunt alocate și memorate:
  - ❑ variabilele locale din cadrul funcțiilor
  - ❑ parametrii funcțiilor
  - ❑ adresele de return ale funcțiilor
- ❑ dimensiunea implicită a stivei este redusă
  - ❑ în timpul execuției programele trebuie să nu depășească dimensiunea stivei
  - ❑ dimensiunea stivei poate fi modificată în prealabil din setările editorului de legături (*linker*)

# Stiva în C – depășirea dimensiunii

- ambele programe eşuează în timpul execuției din cauza depășirii dimensiunii stivei

```
int f()
{
    int a[10000000]={0};

int main()
{
    f();
    return 0;
}
```

```
int f(int a,int b)
{
    if (a<b)
        return 1+f(a+1,b-1);
    else
        return 0;
}

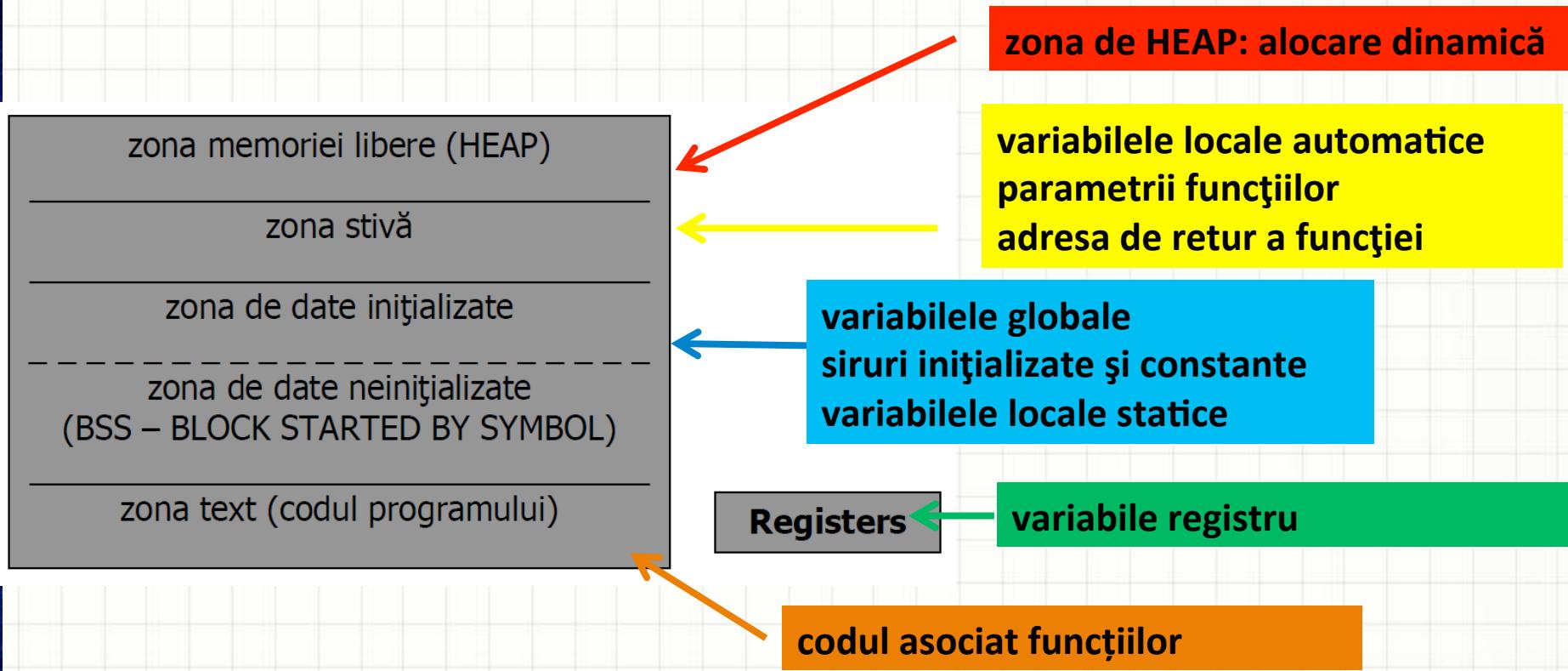
int main()
{
    printf ("%d", f(0,1000000));
}
```

# Cursul de azi

1. Enumerări, typedef.
2. Funcții: declarare și definire, apel, metode transmitere a parametrilor.
3. Pointeri la funcții.
4. Legătura dintre tablouri și pointeri
5. Aritmetică pointerilor

# Pointeri la funcții

- pointer la o funcție = variabilă ce stochează adresa de început a codului asociat funcției



# Harta simplificată a memoriei la rularea unui program

hartaMemorie.c

```
01 #include <stdio.h>
02 // variabile globale neinitializate
03 int g1,g2;
04 // variabile globale initialize
05 int g3=5, g4 = 7;
06 int g5, g6;
07
08 void f1() {
09     int var1,var2;
10     printf("In Stiva prin f1:\t\t %p %p\n",&var1,&var2);
11 }
12
13 void f2() {
14     int var1,var2;
15     printf("In Stiva prin f2:\t\t %p %p\n",&var1,&var2);
16     f1();
17 }
18
19 int main() {
20     //variabile locale
21     int var1,var2;
22     printf("In Stiva prin main:\t\t %p %p\n",&var1,&var2);
23     f2();
24     // variabile globale initialize + neinitializate
25     printf("Variabile globale neinitializate:\t\t %p %p\n",&g1,&g2);
26     printf("Variabile globale initialize: \t\t %p %p\n",&g3,&g4);
27     printf("Variabile globale neinitializate:\t\t %p %p\n",&g5,&g6);
28     //cod
29     printf("Text Data:\t\t\t\t %p %p \n\n",main,f1);←
30     return 0;
31 }
```

Numele unei funcții neînsorit de o listă de argumente este adresa de început a codului funcției și este interpretat ca un pointer la funcția respectivă

# Harta simplificată a memoriei la rularea unui program

hartaMemorie.c

```
01 #include <stdio.h>
02 // variabile globale neinitialize
03 int g1,g2;
04 // variabile globale initialize
05 int g3=5, g4 = 7;
06 int g5, g6;
07
08 void f1() {
09     int var1,var2;
10     printf("In Stiva prin f1:\t\t %p %p\n",&var1,&var2);
11 }
12
13 void f2() {
14     int var1,var2;
15     printf("In Stiva prin f2:\t\t %p %p\n",&var1,&var2);
16     f1();
17 }
18
19 int main() {
20     //variabile locale
21     int var1,var2;
22     printf("In Stiva prin main:\t\t %p %p\n",&var1,&var2);
23     f2();
24     // variabili In Stiva prin main:          0xffff5fbff95c 0xffff5fbff958
25     printf("Val In Stiva prin f2:            0xffff5fbff93c 0xffff5fbff938
26     printf("Val In Stiva prin f1:            0xffff5fbff91c 0xffff5fbff918
27     printf("Val Variabile globale neinitialize: 0x100001070 0x100001074
28     //cod      Variabile globale initialize: 0x100001068 0x10000106c
29     printf("Text Variabile globale neinitialize: 0x100001078 0x10000107c
30     return 0;  Text Data:                  0x100000d54 0x100000d04
31 }
```

# Pointeri la funcții

- pointer la o funcție = variabilă ce stochează adresa de început a codului asociat funcției
- sintaxa: **tip (\*nume\_pointer\_functie) (tipuri argumente)**
  - **tip** = tipul de bază returnat de funcția spre care pointeaza nume\_pointer\_functie
  - **nume\_variabila** = variabila de tip pointer la o functie care poate lua ca valori adrese de memorie unde începe codul unei funcții
  - **observație:** trebuie să pun paranteză în definiție altfel definesc o funcție care întoarce un un pointer de date
- exemplu:
  - void (\*pf)(int)
  - int (\*pf)(int, int)
  - double (\*pf)(int, double\*)

# Pointeri la funcții

```
main.c ✘
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int suma(int a, int b)
5 {
6     return a+b;
7 }
8
9
10
11 int main()
12 {
13     int (*pf)(int,int);
14     pf = &suma; ←
15     int s1 = (*pf)(2,5);
16     printf("s1 = %d\n",s1);
17     pf = suma; ←
18     int s2 = (*pf)(2,5);
19     printf("s2 = %d\n",s2);
20     return 0;
21 }
```

```
s1 = 7
s2 = 7
```

```
Process returned 0 (0x0)   execution
```

1. pentru a asigura unui pointer adresa unei functii, trebuie folosit numele functiei fara paranteze.
2. numele unei functii este un pointer spre adresa sa de inceput din segmentul de cod: f==&f

# Tablouri de pointeri la funcții

The screenshot shows a code editor with a file named "main.c" and a terminal window displaying the execution results.

**Code Editor (main.c):**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 float suma(float a, float b)
5 {return a+b;}
6
7 float diferență(float a, float b)
8 {return a-b;}
9
10 float înmulțire(float a, float b)
11 {return a*b;}
12
13 float împărțire(float a, float b)
14 {return a/b;}
15
16 int main()
17 {
18     float (*pf[4])(float,float);
19     pf[0] = suma;
20     pf[1] = diferență;
21     pf[2] = înmulțire;
22     pf[3] = împărțire;
23     float rez;
24     rez = (*pf[0])(2.3,4.7); printf("rez = %f\n",rez);
25     rez = (*pf[1])(2.3,4.7); printf("rez = %f\n",rez);
26     rez = (*pf[2])(2.3,4.7); printf("rez = %f\n",rez);
27     rez = (*pf[3])(2.3,4.7); printf("rez = %f\n",rez);
28     return 0;
29 }
```

**Terminal Window:**

```
rez = 7.000000
rez = -2.400000
rez = 10.809999
rez = 0.489362

Process returned 0 (0x0)   execution time : 0.00 secs
```

**Annotation:** An orange arrow points from the text "fiecare element din tablou poinează către o altă funcție" to the line of code where the pointers are assigned values.

fiecare element din tablou  
poinează către o altă funcție

# Tablouri de pointeri la funcții

```
main.c ✘
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void suma(float a, float b)
5 {printf("Suma dintre %f și %f este %f\n", a, b, a+b)}
6
7 void diferență(float a, float b)
8 {printf("Diferența dintre %f și %f este %f\n", a, b, a-b)}
9
10 void înmulțire(float a, float b)
11 {printf("Înmulțirea lui %f cu %f este %f\n", a, b, a*b)}
12
13 void împărțire(float a, float b)
14 {printf("Împărțirea lui %f la %f este %f\n", a, b, a/b);}
15
16 int main()
17 {
18     void (*pf[4])(float, float);
19     pf[0] = suma;
20     (*pf[0])(2.3, 4.7); ←
21     pf[1] = diferență;
22     (*pf[1])(2.3, 4.7);
23     pf[2] = înmulțire;
24     (*pf[2])(2.3, 4.7);
25     pf[3] = împărțire;
26     (*pf[3])(2.3, 4.7);
27     return 0;
28 }
```

Suma dintre 2.300000 și 4.700000 este 7.000000  
Diferența dintre 2.300000 și 4.700000 este -2.400000  
Înmulțirea lui 2.300000 cu 4.700000 este 10.809999  
Împărțirea lui 2.300000 la 4.700000 este 0.489362

Process returned 0 (0x0) execution time : 0.010 s  
Press F5 to continue.

fiecare element din tablou  
poinează către o altă funcție.  
Funcțiile au tipul void.

# Utilitatea pointerilor la funcții

- se folosesc în **programarea generică**, realizăm apeluri de tip **callback**;
- o funcție C transmisă, printr-un pointer, ca argument unei alte funcții F se numește și funcție **“callback”**, pentru că ea va fi apelată “înapoi” de funcția F
- **exemple:**
  1. int suma(int n, int (\*expresie)(int)); **(sumă generică de n numere)**
  2. void qsort(void \*adresa,int nr\_elemente, int dimensiune\_element, int (\*cmp)(const void \*, const void \*)); **(funcția qsort din stdlib.h)**

# Utilitatea pointerilor la funcții

- exemplul 1: vreau să calculez suma

$$S_k(n) = \sum_{i=1}^n i^k$$

$$S_1(n) = 1 + 2 + \dots + n$$

$$S_2(n) = 1^2 + 2^2 + \dots + n^2$$

$$S_k(n) = \sum_{i=1}^n expresie(i)$$

Folosind pointeri la funcții pot să văd funcția ca o variabilă

# Utilitatea pointerilor la funcții

- exemplul 1: vreau să calculez suma
- implementare elegantă:

$$S_k(n) = \sum_{i=1}^n i^k$$

```
int suma(int n, int (*expresie)(int))
{
    int i,s=0;
    for(i=1;i<=n;i++)
        s = s + expresie(i);
    return s;
}
```

```
int expresie1(int x)
{
    return x;
}
```

```
int expresie2(int x)
{
    return x*x;
}
```

# Utilitatea pointerilor la funcții

## □ exemplul 1: vreau să calculez suma

```
main.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int suma(int n, int (*expresie)(int))
5 {
6     int i,s=0;
7     for(i=1;i<=n;i++)
8         s = s + expresie(i);
9     return s;
10}
11
12 int expresie1(int x)
13 {
14     return x;
15 }
16
17 int expresie2(int x)
18 {
19     return x*x;
20 }
21
22 int main()
23 {
24
25     int S1 = suma(5,expresie1);
26     printf("S1 = %d\n",S1);
27     int S2 = suma(5,expresie2);
28     printf("S2 = %d\n",S2);
29     return 0;
30 }
```

$$S_k(n) = \sum_{i=1}^n i^k$$

S1 = 15  
S2 = 55

Process returned 0 (0x0) execution time : 0.0

# Utilitatea pointerilor la funcții

- exemplul 2: funcția qsort din stdlib.h folosită pentru sortarea unui vector/tablou. Antetul lui qsort este:

```
void qsort (void *adresa, int nr_elemente, int dimensiune_element,  
int (*cmp) (const void *, const void *))
```

- adresa = pointer la adresa primului element al tabloului ce urmeaza a fi sortat  
(pointer generic – nu are o aritmetică inclusă)
- nr\_elemente = numarul de elemente al vectorului
- dimensiune\_element = dimensiunea in octeți a fiecărui element al tabloului  
(char = 1 octet, int = 4 octeți, etc)
- cmp – funcția de comparare a două elemente

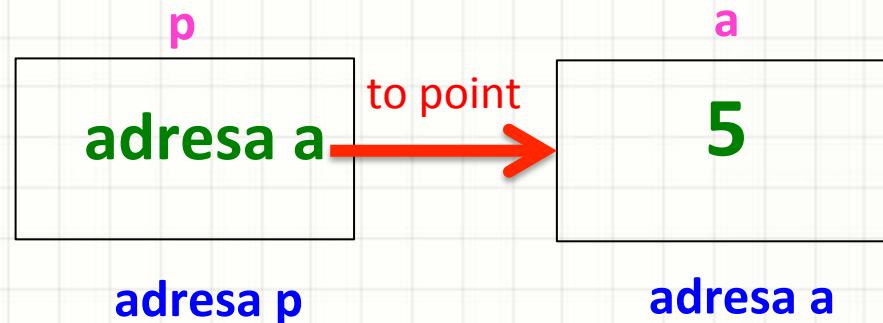
# Cursul de azi

1. Enumerări, typedef.
2. Funcții: declarare și definire, apel, metode transmitere a parametrilor.
3. Pointeri la funcții.
4. Legătura dintre tablouri și pointeri
5. Aritmetică pointerilor

# Legătura dintre pointeri și tablouri 1D

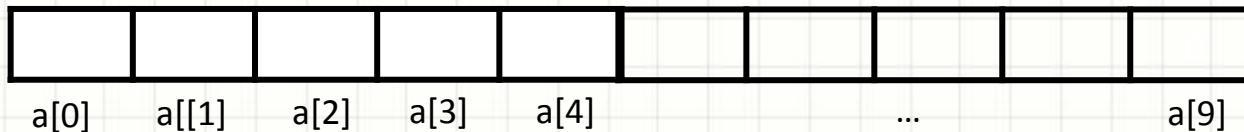
- un pointer: variabilă care poate stoca adrese de memorie

- exemplu:  
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
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 respectivă. Dacă se încearcă acest lucru se returnează eroare la compilarea programului.
- putem modifica valoarea unei variabile însotite de modelatorul **const** prin intermediul unui pointer (în mod indirect):

```
main.c ✘
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 constante

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

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

To the right of the code, the terminal output shows:

```
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:

The screenshot shows a code editor window titled "main.c". The code has been modified:

```
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 }
```

To the right of the code, the terminal output shows:

```
*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 initializare.

```
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)  execution time : 0.004 s
Press ENTER to continue.
```

# 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 initializare și de asemenea că nu poate schimba valoarea variabilei aflate la adresa pe care o conține.

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 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 }
```

Below the code, two error messages are displayed:

- Line 9: error: assignment of read-only location
- Line 11: error: assignment of read-only variable 'p'

At the bottom, a summary message states: "== Build failed: 2 error(s), 0 warning(s) (0 minute(s), 0 second(s))".

# 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

The screenshot shows a code editor window titled "main.c". The code is written in C and demonstrates how pointers can be used to access elements of an array. The code includes two loops: one using a pointer to the array itself and another using a pointer to the first element of the array.

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

int main(){
    int v[5] = {0,2,4,10,20};
    int *p = v;
    int i;
    for (i=0;i<5;i++)
    {
        printf("Accesam elementul %d din vector v prin intermediul lui p.\n",i);
        printf("Valoarea acestui element este = %d \n",*(p+i));
    }

    p = &v[0];
    for (i=0;i<5;i++)
    {
        printf("Accesam elementul %d din vector v prin intermediul lui p.\n",i);
        printf("Valoarea acestui element este = %d \n",*(p+i));
    }
    return 0;
}
```

# 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     int v[5] = {0,2,4,10,20};
7     int *p = v;
8     int i;
9     for (i=0;i<5;i++)
10    {
11        printf("Accesam elementul ");
12        printf("Valoarea acestui ");
13    }
14
15    p = &v[0];
16    for (i=0;i<5;i++)
17    {
18        printf("Accesam elementul ");
19        printf("Valoarea acestui ");
20    }
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 ✘
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 time ...
Press ENTER to continue.
```

Concluzie?

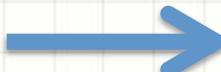
# Legătura dintre pointeri și tablouri 1D

- 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.
- la compilare, expresia  $v[i]$  se înlocuiește cu  $*(v+i)$ . Atunci, ***i[v] este o expresie corectă*** întrucât  $i[v]$  se înlocuiește cu  $*(i+v) = *(v+i)$ . Deci,  $v[i] = i[v]$ .
- $\&v[i] = \&(*(v+i)) = v + i \Rightarrow \&v[0] = v$

# Legătura dintre pointeri și tablouri 1D

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

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

`int a[10][10];`   $a = \&a[0][0];$

- elementele unui tablou pot fi accesate prin pointeri:

| index              | 0      | 1        | i        | n-1        |
|--------------------|--------|----------|----------|------------|
| accesare directă   | $v[0]$ | $v[1]$   | $v[i]$   | $v[n-1]$   |
| accesare indirectă | $*v$   | $*(v+1)$ | $*(v+i)$ | $*(v+n-1)$ |
| adresa             | $v$    | $v+1$    | $v+i$    | $v+n-1$    |

- operatorul \* are prioritate mai mare ca +
- $*(v+1)$  e diferit de  $*v+1$

# Legătura dintre pointeri și tablouri 1D

| index              | 0      | 1        | i        | n-1        |
|--------------------|--------|----------|----------|------------|
| accesare directă   | $v[0]$ | $v[1]$   | $v[i]$   | $v[n-1]$   |
| accesare indirectă | $*v$   | $*(v+1)$ | $*(v+i)$ | $*(v+n-1)$ |
| adresa             | $v$    | $v+1$    | $v+i$    | $v+n-1$    |

- o expresie cu tablou și indice este echivalentă cu una scrisă ca pointer și distanță de deplasare:  $v[i] = *(v+i)$
- **diferența dintre un nume de tablou și un pointer:**
  - un pointer își poate schimba valoarea:  $p = v$  și  $p++$  **sunt expresii corecte**
  - un nume de tablou este un pointer constant (nu își poate schimba valoarea):  $v = p$  și  $v++$  **sunt expresii incorecte**

# 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**

|   |     |    |   |   |
|---|-----|----|---|---|
| 3 | -12 | 10 | 7 | 1 |
|---|-----|----|---|---|

a[0]

|    |   |   |    |    |
|----|---|---|----|----|
| 10 | 2 | 0 | -7 | 41 |
|----|---|---|----|----|

a[1]

|    |    |   |   |   |
|----|----|---|---|---|
| -3 | -2 | 0 | 0 | 2 |
|----|----|---|---|---|

a[2]

# Pointeri la pointeri (pointeri dubli)

## □ sintaxa

**tip      `**nume_variabilă;`**

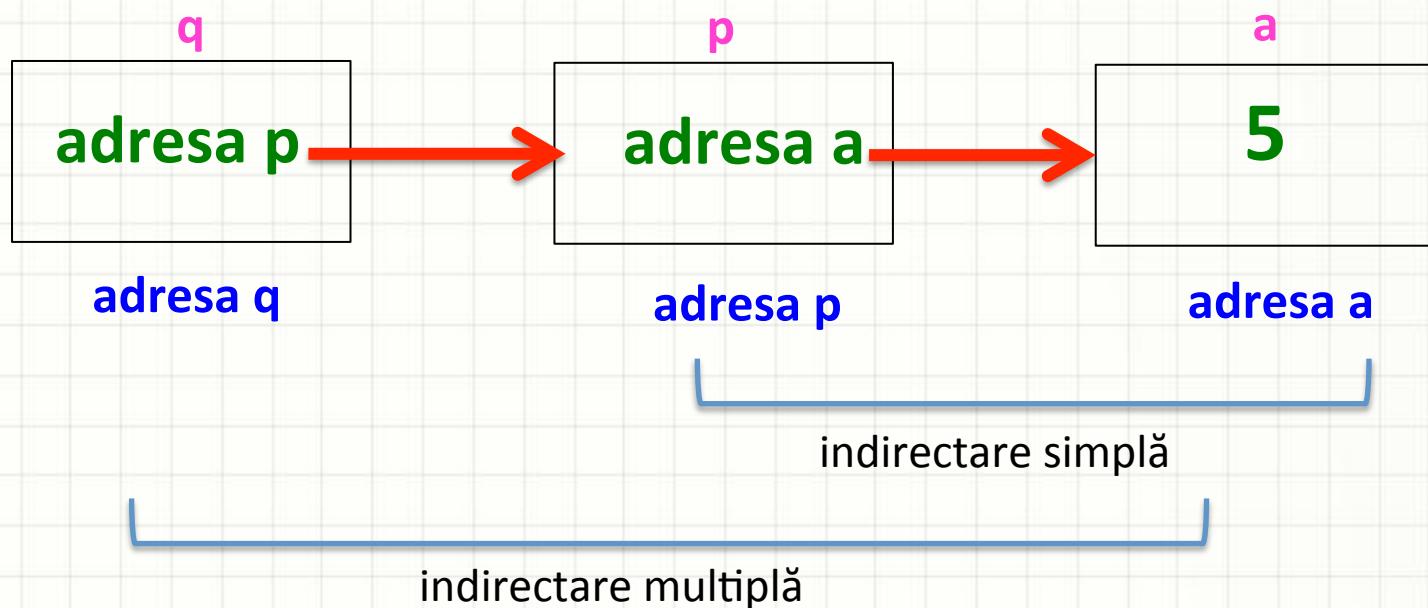
**tip** = tipul de bază al variabilei de tip pointer dublu nume\_variabilă;

**\*** = operator de indirectare;

**nume\_variabila** = variabila de tip pointer dublu care poate lua ca valori adrese de memorie ale unor variabile de tip pointer.

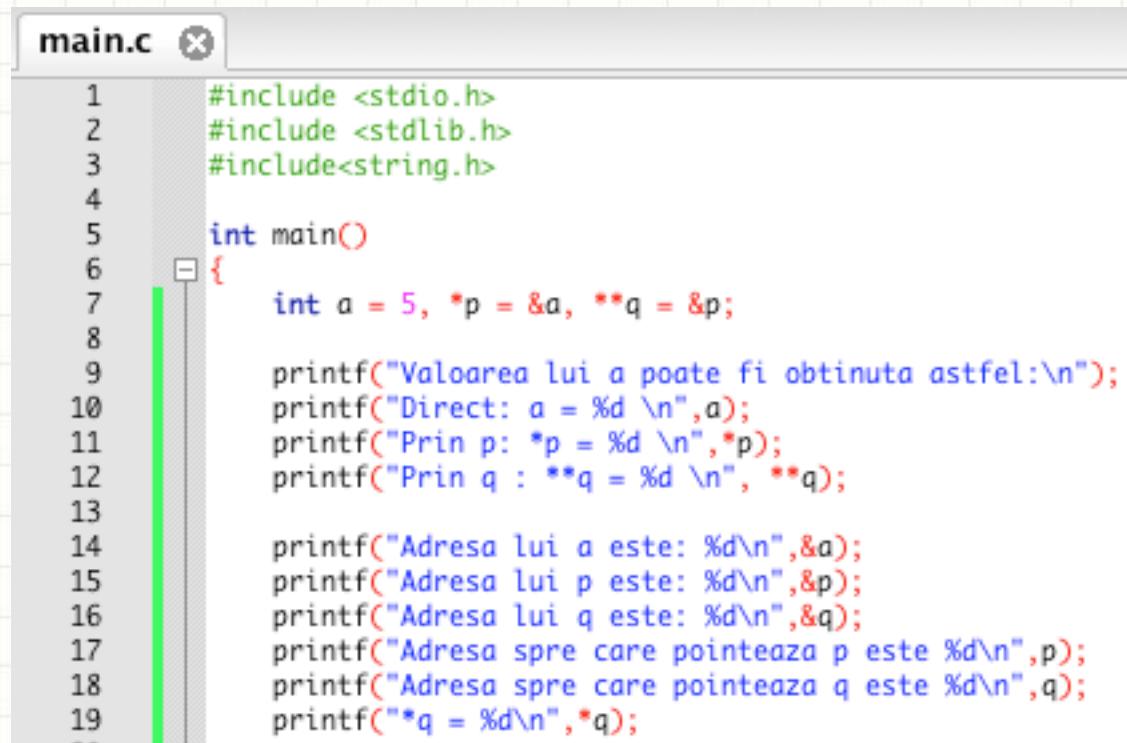
## □ exemplu:

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



# Pointeri la pointeri

## exemplu:



```
main.c
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);
```

# Pointeri la pointeri

## exemplu:

The diagram illustrates the memory layout for the variables defined in the code. Variable **q** is a pointer to variable **p**, which in turn points to variable **a**. The value of **a** is 5.

```
main.c
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);
```

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  
\*q = 1606416748

Process returned 0 (0x0) execution time : 0.009 s  
Press ENTER to continue.

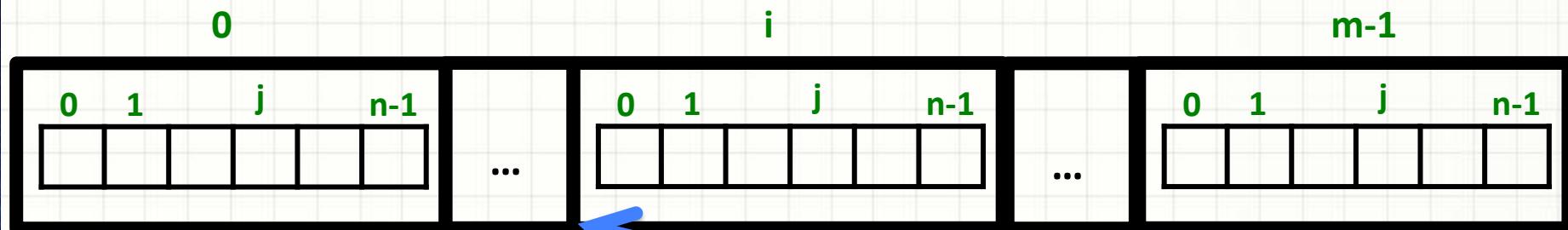
Diagram illustrating pointer relationships:

- Variable **q** (green box) contains the address **1606416736**.
- An arrow points from **q** to variable **p** (blue box), which contains the address **1606416748**.
- Another arrow points from **p** to variable **a** (red box), which contains the value **5**.

Below the boxes, their corresponding addresses are written again: **1606416728** under **q**, **1606416736** under **p**, and **1606416748** under **a**.

# Legătura dintre pointeri și tablouri 2D

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



Reprezentarea în memoria calculatorului a unui tablou bidimensional

$a[i]$  este un tablou unidimensional. La ce adresă incepe  $a[i]$ ?

# Legătura dintre pointeri și tablouri 2D

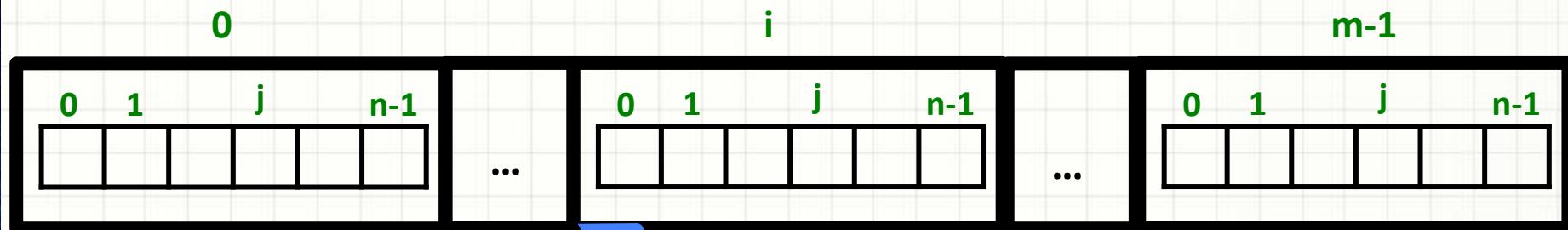
## □ tablou bidimensional = tablou de tablouri

```
tablou_pointer.c
01 #include <stdio.h>
02
03 int main()
04 {
05     int a[5][5],i,j;
06
07     printf("Adresa de inceput a tabloului a este %p \n",a);
08     for (i=0;i<5;i++)
09         printf("Adresa de inceput a tabloului a[%d] este %p \n",i,&a[i]);
10
11     for (i=0;i<5;i++)
12         printf("Adresa de inceput a tabloului a[%d] este %d \n",i,&a[i]);
13
14     return 0;
15 }
```

```
Adresa de inceput a tabloului a este 0x7fff5fbff8e0
Adresa de inceput a tabloului a[0] este 0x7fff5fbff8e0
Adresa de inceput a tabloului a[1] este 0x7fff5fbff8f4
Adresa de inceput a tabloului a[2] este 0x7fff5fbff908
Adresa de inceput a tabloului a[3] este 0x7fff5fbff91c
Adresa de inceput a tabloului a[4] este 0x7fff5fbff930
Adresa de inceput a tabloului a[0] este 1606416608
Adresa de inceput a tabloului a[1] este 1606416628
Adresa de inceput a tabloului a[2] este 1606416648
Adresa de inceput a tabloului a[3] este 1606416668
Adresa de inceput a tabloului a[4] este 1606416688
```

# Legătura dintre pointeri și tablouri 2D

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

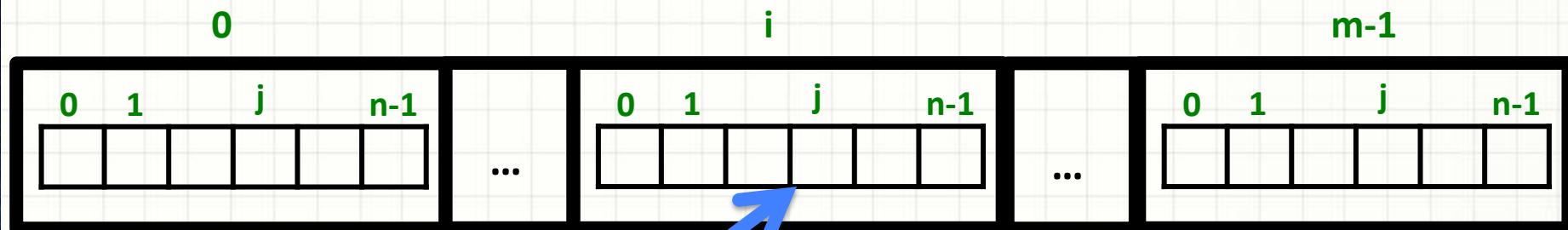


Reprezentarea în memoria calculatorului a unui tablou bidimensional

$a[i]$  este un tablou unidimensional. La ce adresă începe  $a[i]$ ?  
 $a[i]$  începe la adresa  $\&a[i] = \&(*(\&a+i)) = a+i$

# Legătura dintre pointeri și tablouri 2D

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

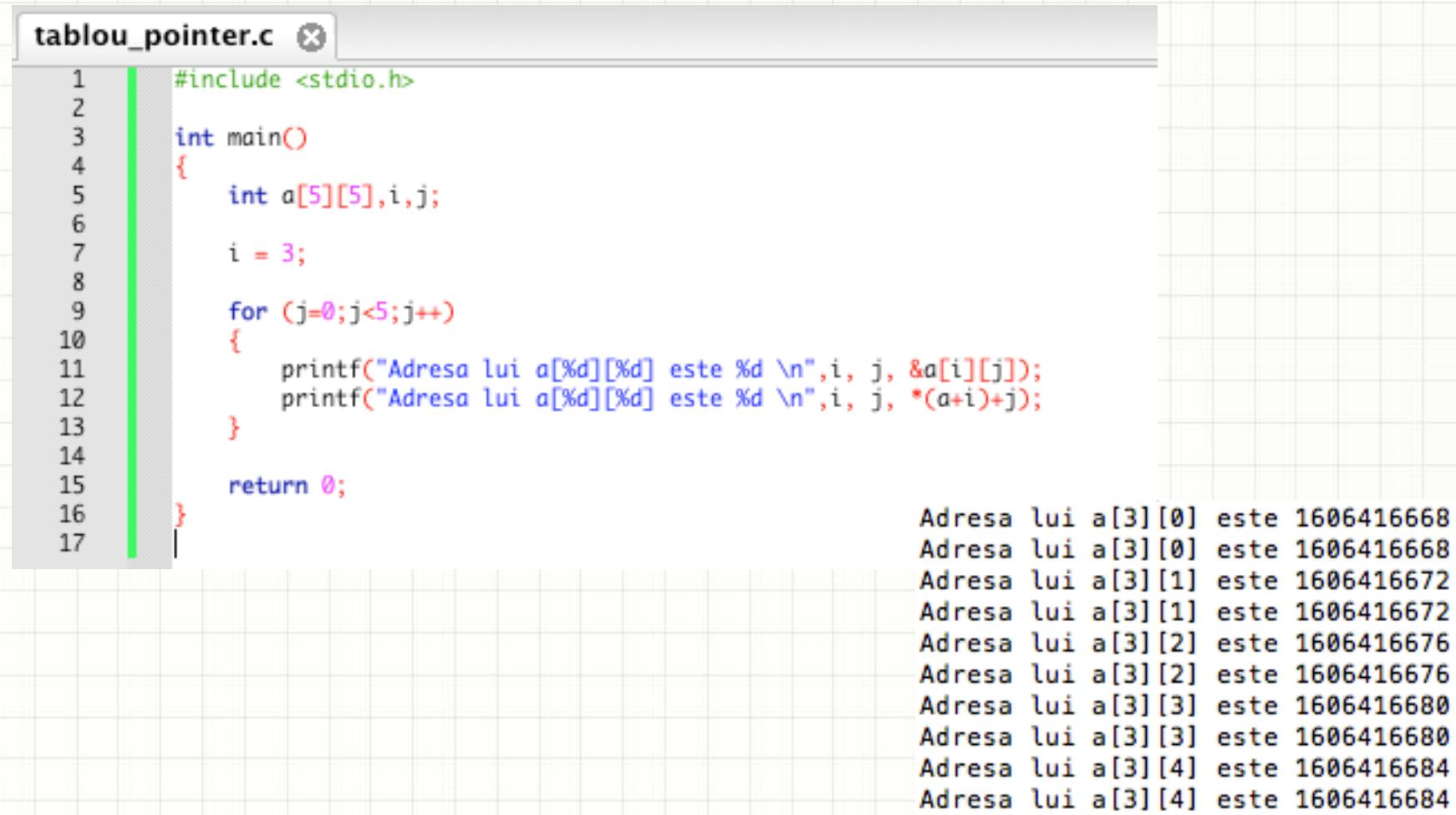


Reprezentarea în memoria calculatorului a unui tablou bidimensional

a[i] este un tablou unidimensional. La ce adresă începe a[i]?  
a[i] începe la adresa  $\&a[i] = \&(*(a+i)) = a+i$   
Care este adresa lui a[i][j]? Cum o exprim în aritmetică pointerilor în funcție de a, i, j?

# Legătura dintre pointeri și tablouri 2D

## ❑ tablou bidimensional = tablou de tablouri

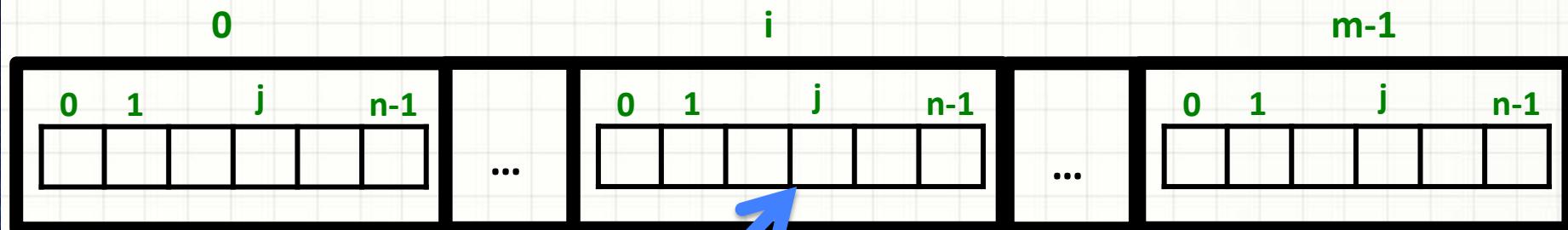


```
tablou_pointer.c
1 #include <stdio.h>
2
3 int main()
4 {
5     int a[5][5], i, j;
6
7     i = 3;
8
9     for (j=0;j<5;j++)
10    {
11        printf("Adresa lui a[%d][%d] este %d \n", i, j, &a[i][j]);
12        printf("Adresa lui a[%d][%d] este %d \n", i, j, *(a+i)+j);
13    }
14
15    return 0;
16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

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



Reprezentarea în memoria calculatorului a unui tablou bidimensional

a[i] este un tablou unidimensional. La ce adresă începe a[i]?  
a[i] începe la adresa  $\&a[i] = \&(*(a+i)) = a+i$   
Care este adresa lui a[i][j]? Cum o exprim în aritmetică pointerilor în funcție de a, i, j?  
Adresa lui a[i][j] =  $\&a[i][j] = *(a+i)+j$  (a este pointer dublu).  
Cum exprim valoarea lui a[i][j] în aritmetică pointerilor în funcție de a, i, j?

# Legătura dintre pointeri și tablouri 2D

## □ tablou bidimensional = tablou de tablouri

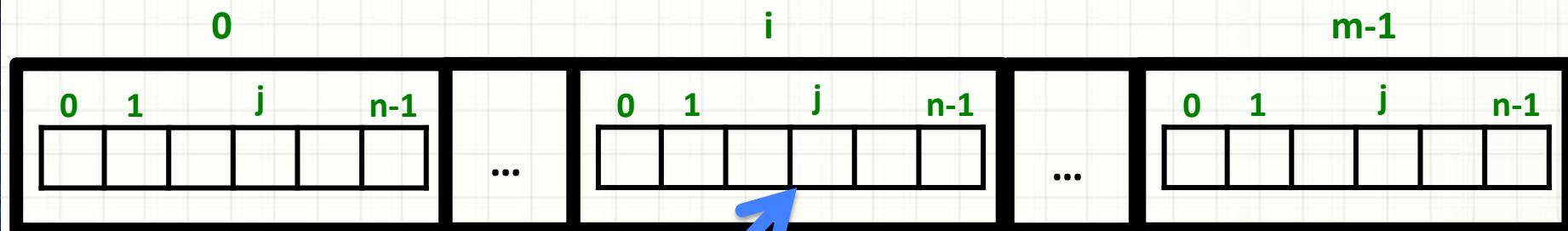
tablou\_pointer\_2.c

```
01 #include <stdio.h>
02
03 int main()
04 {
05     int a[5][5], i, j;
06
07     for(i=0; i<5; i++)
08         for(j=0; j<5; j++)
09             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
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**
- **cazul general:** int a[m][n];



Reprezentarea în memoria calculatorului a unui tablou bidimensional

**a[i] este un tablou unidimensional. La ce adresă începe a[i]?**

**a[i] începe la adresa  $\&a[i] = \&(*(a+i)) = a+i$**

**Care este adresa lui a[i][j]? Cum o exprim în aritmetică pointerilor în funcție de a, i, j?**

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

**Cum exprim valoarea lui a[i][j] în aritmetică pointerilor în funcție de a, i, j?**

**$a[i][j] = *(*(a+i)+j) = \text{valoarea lui } a[i][j]$**

# Legătura dintre pointeri și tablouri 2D

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

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. Enumerări, typedef.
2. Funcții: declarare și definire, apel, metode transmitere a parametrilor.
3. Pointeri la funcții.
4. Legătura dintre tablouri și pointeri
5. Aritmetică pointerilor

# Aritmetica pointerilor

- asupra pointerilor pot fi realizate operații aritmetice:
  - incrementare (++) , decrementare (--);
  - adăugare (+ sau +=) sau scădere a unui intreg (- 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
```

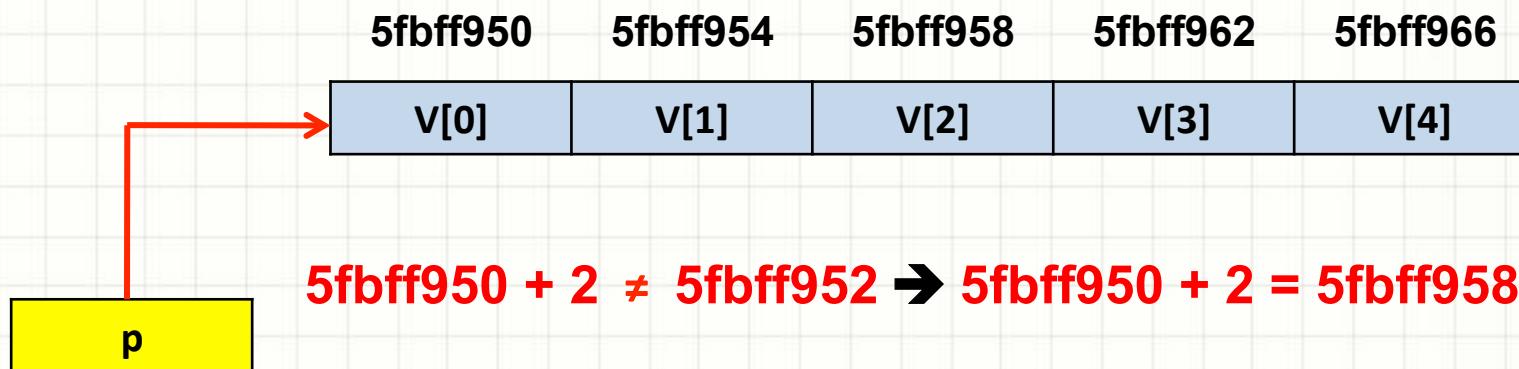
Adresa lui v[0] este 5fbff950  
Adresa lui v este 5fbff950

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

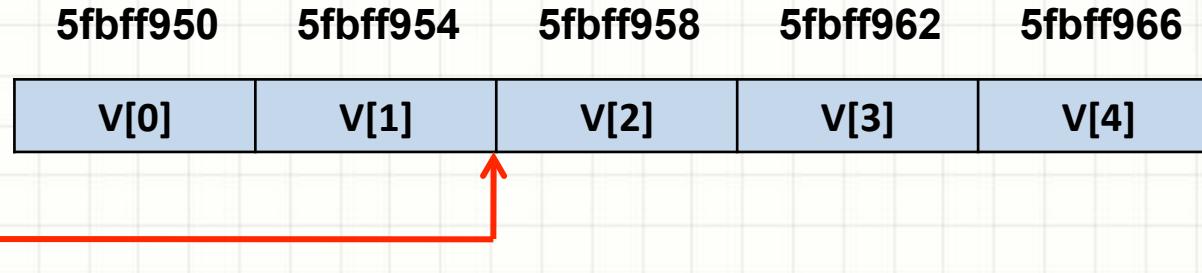
v este un pointer care  
pointeaza către v[0]

# 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

- adunarea/scăderea unui număr natural dintr-un pointer

```
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    p = p + 2;
17    printf("Adresa spre care pointeaza acum p este %x \n", p);
18
19    return 0;
20
21 }
```

Adresa lui v[0] este 5fbff950  
Adresa lui v este 5fbff950  
Adresa spre care pointeaza acum p este 5fbff958

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

# Aritmetica pointerilor

- adunarea/scăderea unui număr natural dintr-un pointer

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

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

int main(){
    int v[5];
    int *p;

    p = &v[0];
    printf("Adresa spre care pointeaza acum p este %d \n", p);
    p+=4;
    printf("Adresa spre care pointeaza acum p este %d \n", p);
    p-=2;
    printf("Adresa spre care pointeaza acum p este %d \n", p);
    p++;
    printf("Adresa spre care pointeaza acum | Adresa spre care pointeaza acum p este 1606416720
    ++p;                                Adresa spre care pointeaza acum p este 1606416736
    printf("Adresa spre care pointeaza acum | Adresa spre care pointeaza acum p este 1606416728
    p--;                                Adresa spre care pointeaza acum p este 1606416732
    printf("Adresa spre care pointeaza acum | Adresa spre care pointeaza acum p este 1606416736
    --p;                                Adresa spre care pointeaza acum p este 1606416732
    printf("Adresa spre care pointeaza acum | Adresa spre care pointeaza acum p este 1606416728

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

The output of the program is displayed below the code, showing the address of the variable p and its value after each modification. The addresses are 1606416720, 1606416736, 1606416728, 1606416732, 1606416736, 1606416732, and 1606416728 respectively. The final message at the bottom indicates the process returned 0 and the execution time was 0.007 seconds.

# Aritmetică 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

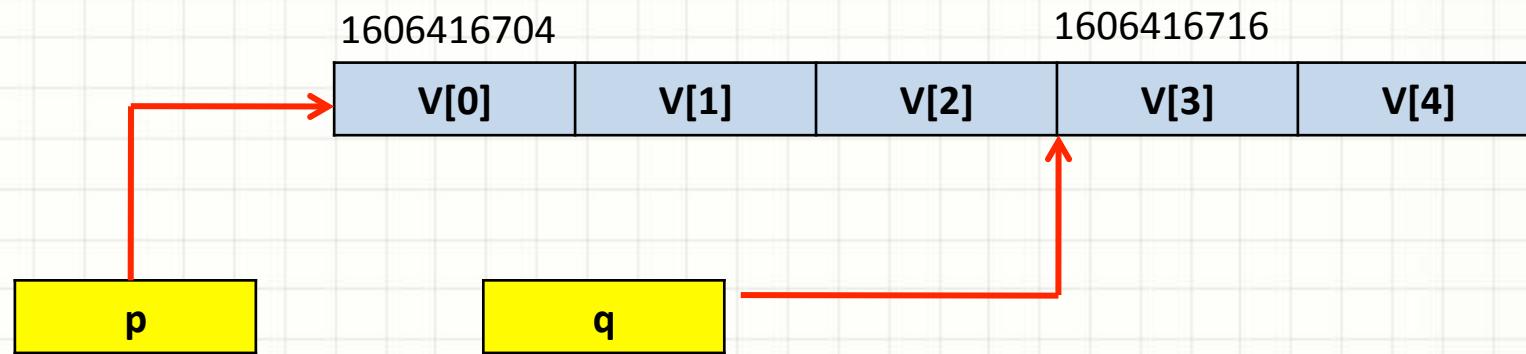
```
main.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4
5 int main(){
6     int v[5];
7     int *p,*q;
8
9     p = &v[0];
10    printf("Adresa spre care pointeaza acum p este %d \n", p);
11
12    q = &v[3];
13    printf("Adresa spre care pointeaza acum q este %d \n", q);
14
15    printf("Rezultatul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

Adresa spre care pointeaza acum p este 1606416704  
Adresa spre care pointeaza acum q este 1606416716  
Rezultatul diferenței dintre q și p este 3  
Rezultatul diferenței dintre p și 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

The screenshot shows a code editor window with the file "main.c" open. The code defines an array `v` and pointers `p` and `q`, then compares their addresses and outputs the results.

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

int main(){
    int v[5];
    int *p,*q;

    p = &v[2];
    printf("Adresa spre care pointeaza acum p este %d \n", p);
    q = &v[4];
    printf("Adresa spre care pointeaza acum q este %d \n", q);

    p > q ? printf("p este la dreapta lui q\n"):printf("p este la stanga lui q\n");
    q = &v[0];
    printf("Adresa spre care pointeaza acum q este %d \n", q);
    p > q ? printf("p este la dreapta lui q\n"):printf("p este la stanga lui q\n");

    return 0;
}
```

The output window shows the following results:

```
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     int v[5];
7     int *p,*q;
8
9     p = &v[2];
10    printf("Adresa spre care pointeaza acum p este %d \n", p);
11    q = &v[4];
12    printf("Adresa spre care pointeaza acum q este %d \n", q);
13
14    p - q > 0? printf("p este la dreapta lui q\n"):printf("p este la stanga lui q\n");
15    q = &v[0];
16    printf("Adresa spre care pointeaza acum q este %d \n", q);
17    p - q > 0? printf("p este la dreapta lui q\n"):printf("p este la stanga lui q\n");
18
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

- compararea unei variabile de tip pointer cu constanta NULL (0)

```
main.c  X
1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main(){
6
7      int v[5];
8      int *q=NULL;
9
10     printf("Adresa spre care pointeaza acum q este %d \n", q);
11     if(q)
12         printf("q contine o adresa valida\n");
13     else
14         printf("q nu contine o adresa valida\n");
15
16     return 0;
17 }
```

```
Adresa spre care pointeaza acum q este 0
q nu contine o adresa valida

Process returned 0 (0x0)  execution time : 0.009 s
Press ENTER to continue.
```

# Aritmetică pointerilor

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

```
main.c
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.
```

# Aritmetică pointerilor

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

```
5 int main(){
6
7     double a=3.14,b=2*a;
8     int x=10,y=20,z=30,w=40;
9
10
11    printf("Adresa lui a este %d \n",&a);
12    printf("Adresa lui b este %d \n",&b);
13    printf("Adresa lui x este %d \n",&x);
14    printf("Adresa lui y este %d \n",&y);
15    printf("Adresa lui z este %d \n",&z);
16    printf("Adresa lui w este %d \n",&w);
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    double *p = &b;
21    *p = 5.2;
22    *(p+1) = 6.4;
23    *(p+2) = 100.54;
24    *(p+3) = 1000.971;
25
26    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),
27
```

|                              |
|------------------------------|
| Adresa lui a este 1606416728 |
| Adresa lui b este 1606416720 |
| Adresa lui x este 1606416748 |
| Adresa lui y este 1606416744 |
| Adresa lui z este 1606416740 |
| Adresa lui w este 1606416736 |
| 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                 |