



PROGRAMARE PROCEDURALĂ

Bogdan Alexe

bogdan.alexe@fmi.unibuc.ro

Secția Informatică, anul I,

2018-2019

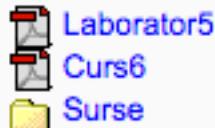
Cursul 7

Recapitulare – cursul trecut

1. Fișiere: noțiuni generale.
2. Fișiere text: funcții specifice de manipulare.
3. Fișiere binare: funcții specifice de manipulare.
4. Funcții

Seminarul 4

6



7



FIŞIERE TEXT ŞI BINARE

1. Scrieți o funcție care preia numerele întregi dintr-un fișier text și le scrie într-un fișier binar, respectiv o funcție care preia numerele dintr-un fișier binar și le scrie într-un fișier text.

Rezolvare:

Funcții de acest tip sunt necesare în programele care prelucrează informații stocate în fișiere binare, deoarece conținutul lor este codificat sub forma unui șir de octeți. Astfel, pentru a putea vizualiza datele dintr-un fișier binar într-un format specific unui tip de dată predefinit este necesară copierea lor într-un fișier text.

Funcția `textBinar` copiază conținutul unui fișier text într-un fișier binar. Astfel, funcția primește ca argumente două șiruri de caractere prin care se specifică calea fișierului text `nfin`, respectiv calea fișierului binar `nfout`. Pentru a copia numerele întregi din fișierului text `nfin` în fișierul binar `nfout`, fișierul text este deschis în modul citire și parcurs element cu element, folosind funcția `fscanf`. Fișierul binar `nfout` este deschis în modul scriere, iar fiecare număr întreg citit din fișierul text este scris în fișierul binar folosind funcția `fwrite`.

Programa cursului

- Introducere**
 - Algoritmi
 - Limbaje de programare.
- Fundamentele limbajului C**
 - Introducere în limbajul C. Structura unui program C.
 - Tipuri de date fundamentale. Variabile. Constante. Operatori. Expresii. Conversii.
 - Tipuri derivate de date: tablouri, siruri de caractere, structuri, uniuni, câmpuri de biți, enumerări, pointeri
 - Instrucțiuni de control
 - Directive de preprocessare. Macrodefiniții.
 - Funcții de citire/scriere.
 - Etapele realizării unui program C.
- Fișiere text**
 - Funcții specifice de manipulare.
- Fișiere binare**
 - Funcții specifice de manipulare.
- Funcții (1)**
 - Declarare și definire. Apel. Metode de transmitere 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.
- 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**

Cuprinsul cursului de azi

1. Funcții
2. Pointeri la funcții
3. Aritmetică pointerilor
4. Legătura dintre tablouri și pointeri

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)
 - 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 acestor 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)

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

Cod scris în C++ !!!

```
interschimbare.cpp  x

1 #include <stdio.h>
2
3 void interschimba1(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     interschimba1(x,y);
22     printf("x = %d, y = %d \n",x,y); <-- apel prin valoare
23
24     x = 10, y = 15;
25     interschimba2(x,y);
26     printf("x = %d, y = %d \n",x,y); <-- apel prin referință
27     numai în C++
28
29     x = 10, y = 15;
30     interschimba3(&x,&y);
31     printf("x = %d, y = %d \n",x,y); <-- apel prin valoare (se
32     transmit adresele variabilelor = pointeri)
33 }
```

```
Bogdan-Alexes-MacBook-Pro:curs7 bogdan$ g++ interschimbare.cpp
Bogdan-Alexes-MacBook-Pro:curs7 bogdan$ ./a.out
x = 10, y = 15
x = 15, y = 10
x = 15, y = 10
```

apel prin valoare

apel prin referință
numai în C++

apel prin valoare (se
transmit adresele
variabilelor = pointeri)

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
- ❑ **nu există apel prin referință în limbajul C**
- ❑ 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

```
exempluTransmitere1.c  x

1 #include <stdio.h>
2
3 int f1(int a, int b)
4 {
5     a++;
6     b++;
7     printf("In f1 avem a= %d \t b = %d \n",a,b);
8     return a + b;
9 }
10
11 int f2(int *a, int b)
12 {
13     *a = *a + 1;
14     b++;
15     return *a + b;
16 }
17
18 int main()
19 {
20     int x = 5, y = 8;
21     int z = f1(x, y);
22     printf("In main dupa f1 avem x = %d, y = %d, z = %d \n",x,y,z);
23     z = f2(&x, y);
24     printf("In main dupa f1 avem x = %d, y = %d, z = %d \n",x,y,z);
25     return 0;
26 }
```

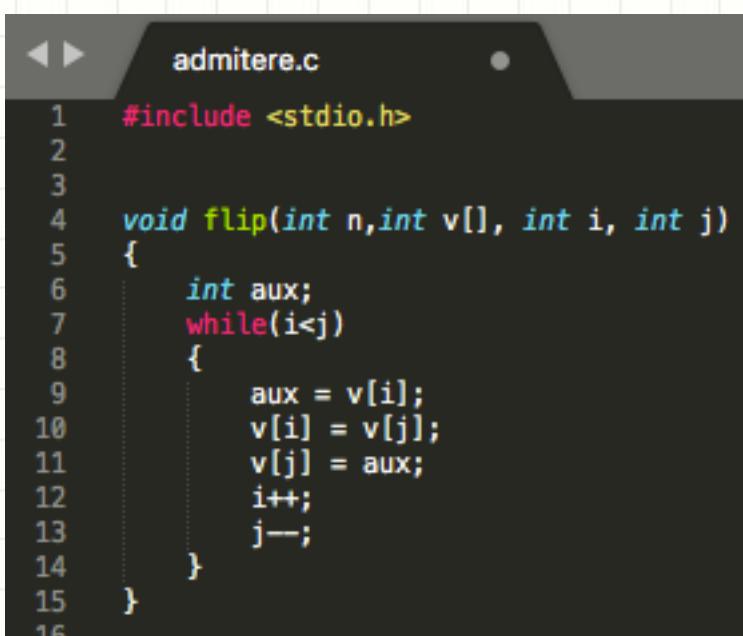
```
[Bogdan-Alexes-MacBook-Pro:curs7 bogdan$ gcc exempluTransmitere1.c
[Bogdan-Alexes-MacBook-Pro:curs7 bogdan$ ./a.out
In f1 avem a= 6          b = 9
In main dupa f1 avem x = 5, y = 8, z = 15
In main dupa f1 avem x = 6, y = 8, z = 15
[Bogdan-Alexes-MacBook-Pro:curs7 bogdan$ ]
```

Problema de la admitere iunie 2017

IV. Informatică.

Fie n un număr natural nenul. Fie v un vector cu n poziții numerotate de la 1 la n și elemente numere naturale diferite, de la 1 la n , într-o ordine oarecare. Pentru i și j numere naturale între 1 și n , numim $\text{FLIP}(n, v, i, j)$ operația care inversează ordinea elementelor din v situate pe pozițiile de la i la j .

- a) Să se scrie în limbaj de programare o procedură (sau funcție) care implementează operația $\text{FLIP}(n, v, i, j)$.

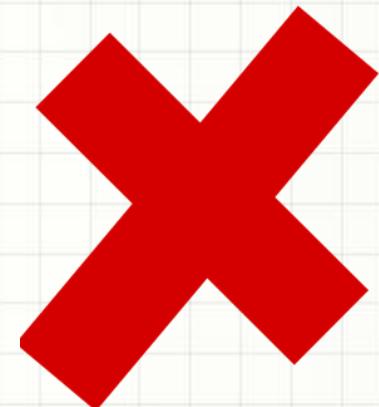


```
admitere.c
001 #include <stdio.h>
002
003
004 void flip(int n,int v[], int i, int j)
005 {
006     int aux;
007     while(i<j)
008     {
009         aux = v[i];
010         v[i] = v[j];
011         v[j] = aux;
012         i++;
013         j--;
014     }
015 }
```

Rezolvare 1

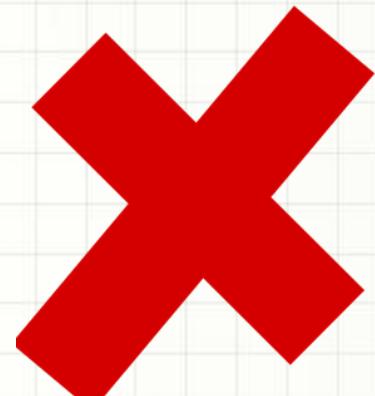
```
a) void FLIP (int n, int v[], int i, int j)
    {
        if (i < j)
            {
                int aux = v[i];
                v[i] = v[j];
                v[j] = aux;
            }
        FLIP (n, v, i+1, j-1);
    }
```

Dacă pun apelul recursiv în IF e soluție bună, altfel ieșe din vector la stânga și la dreapta



Rezolvare 2

```
a. void FLIP(int n, int V[], int i, j)
void FLIP(int n, int V[], int i, int j)
{
    int aux;
    for(j; j >= i; j--)
        for(i; i <= j; i++)
    {
        V[j] = V[i]
        aux = V[j];
        V[j] = V[i];
        V[i] = aux;
    }
}
```



La prima iteratie îl ajunge pe j, apoi algoritmul se opreste

Rezolvare 3

Subiectul IV

a) void FLIP(unsigned n, unsigned v[101], unsigned i,
unsigned j)

```
{ unsigned aux=0, nr=0;  
for( int l=i; l<=j; l++)  
{ aux=v[l];  
v[l]=v[j-nr];  
v[j-nr]=aux;  
nr++;  
if (nr===(j-i+1)/2)  
{ l=j; } }
```

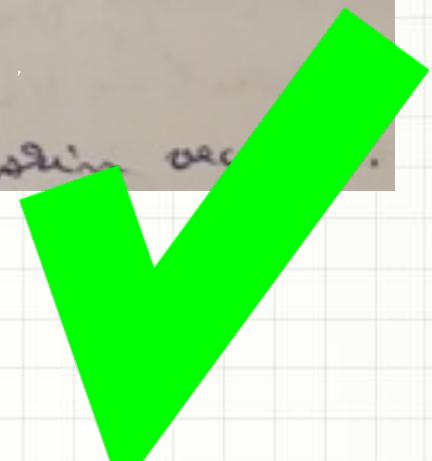


Rezolvare 4

IV Informatică

a) void FLIP(int n, int v[1000], int i, int j)
{ int k; aux;
for(k = i; k <= i + (j - i - 1)/2; k++)
{ aux = v[k];
v[k] = v[j + i - k];
v[j + i - k] = aux; } }

g. să se scrie o funcție care să înlocuiască elementele de la index i la j.



Returnarea de valori multiple

- ❑ o funcție returnază/întoarce maxim o singură valoare;
- ❑ putem întoarce (modifica) mai multe valori fie:
 - ❑ transmițând multiple variabile prin pointeri;
 - ❑ folosind tipuri de date derivate (structuri);
 - ❑ combinații (pointeri + valori întoarse);
- ❑ exemplu: funcție care calculează maximul și minimul valorilor unui tablou v unidimensional cu n numere întregi

Returnarea de valori multiple

- exemplu: funcție care calculează maximul și minimul valorilor unui tablou v unidimensional cu n numere întregi

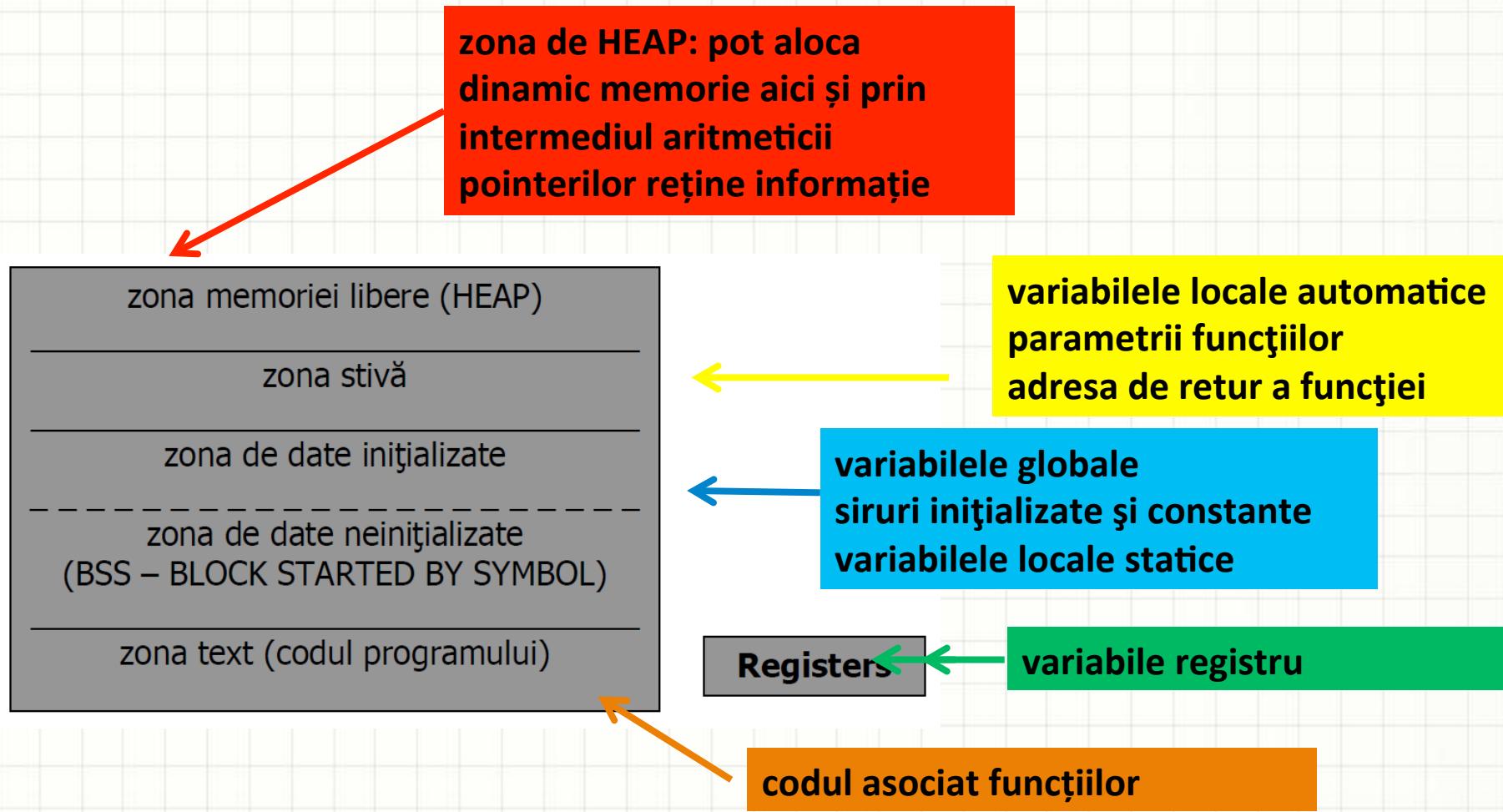
```
#include <stdio.h>
typedef struct
{
    int min;
    int max;
} minmax;
minmax calculeazaMinimMaximTablou1(int v[], int n)
{
    int minim = v[0];
    int maxim = v[0];
    for(int i = 1;i < n;i++)
    {
        if (minim > v[i])
            minim = v[i];
        if (maxim < v[i])
            maxim = v[i];
    }
    minmax x;
    x.min = minim;
    x.max = maxim;
    return x;
}
```

Returnarea de valori multiple

- exemplu: funcție care calculează maximul și minimul valorilor unui tablou v unidimensional cu n numere întregi

```
27 void calculeazaMinimMaximTablou2(int v[],int n, int* min,int *max)
28 {
29     *min = v[0];
30     *max = v[0];
31     for(int i = 1;i < n;i++)
32     {
33         if (*min > v[i])
34             *min = v[i];
35         if (*max < v[i])
36             *max = v[i];
37     }
38 }
39
40 int calculeazaMinimMaximTablou3(int v[],int n, int* min)
41 {
42     *min = v[0];
43     int max = v[0];
44     for(int i = 1;i < n;i++)
45     {
46         if (*min > v[i])
47             *min = v[i];
48         if (max < v[i])
49             max = v[i];
50     }
51     return max;
52 }
```

Harta simplificată a memoriei la rularea unui program



Harta simplificată a memoriei la rularea unui program

```
hartaMemorie.c
#include <stdio.h>

// variabile globale neinitialize
int g1,g2;

// variabile globale initialize
int g3=5, g4 = 7;

// variabile globale neinitialize
int g5, g6;

void f1() {
    int var1, var2;
    printf("In Stiva prin f1:\t\t %p %p\n",&var1,&var2);
}

void f2() {
    int var1, var2;
    printf("In Stiva prin f2:\t\t %p %p\n",&var1,&var2);
    f1();
}

int main() {
    //variabile locale
    int var1, var2;
    printf("In Stiva prin main:\t\t %p %p\n",&var1,&var2);
    f2();
    // variabile globale initialize + neinitialize
    printf("Variabile globale neinitialize: %p %p\n",&g1,&g2);
    printf("Variabile globale initialize: %p %p\n",&g3,&g4);
    printf("Variabile globale neinitialize: %p %p\n",&g5,&g6);
    //cod
    printf("Text Data:\t\t %p %p \n\n",main,f1);
    return 0;
}
```

Harta simplificată a memoriei la rularea unui program

```
hartaMemorie.c
```

```
1 #include <stdio.h>
2
3 // variabile globale neinitialize
4 int g1,g2;
5
6 // variabile globale initialize
7 int g3=5, g4 = 7;
8
9 // variabile globale neinitialize
10 int g5, g6;
11
12 void f1() {
13     int var1, var2;
14     printf("In Stiva prin f1:\t\t %p %p\n",&var1,&var2);
15 }
16
17 void f2() {
18     int var1, var2;
19     printf("In Stiva prin f2:\t\t %p %p\n",&var1,&var2);
20     f1();
21 }
22
23 int main() {
24     //variabile locale
25     int var1, var2;
26     printf("In Stiva prin main:\t\t %p %p\n",&var1,&var2);
27     f2();
28     // variabile globale initialize
29     printf("Variabile globale initialize: %p %p\n",&g1,&g2);
30     printf("Variabile globale neinitialize: %p %p\n",&g3,&g4);
31     printf("Variabile globale neinitialize: %p %p\n",&g5,&g6);
32     //cod
33     printf("Text Data:\t\t\t %p\n",&TextData);
34     return 0;
35 }
```

```
[Bogdan-Alexes-MacBook-Pro:curs7 bogdan$ gcc hartaMemorie.c
[Bogdan-Alexes-MacBook-Pro:curs7 bogdan$ ./a.out
In Stiva prin main:          0x7fff532fab58 0x7fff532fab54
In Stiva prin f2:            0x7fff532fab1c 0x7fff532fab18
In Stiva prin f1:            0x7fff532faafc 0x7fff532faaf8
Variabile globale initialize: 0x10c906020 0x10c906024
Variabile globale neinitialize: 0x10c906018 0x10c90601c
Variabile globale neinitialize: 0x10c906028 0x10c90602c
Text Data:                  0x10c905e10 0x10c905db0
```

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*)

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
 - 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

Stiva în C – depășirea dimensiunii

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

```
exempluStiva1.c
```

```
1 #include <stdio.h>
2
3 int f()
4 {
5     int a[10000000] = {0};
6     return 1;
7 }
8
9 int main()
10 {
11     f();
12     return 0;
13 }
```

```
[Bogdan-Alexes-MacBook-Pro:curs7 bogdan$ gcc exempluStiva1.c
[Bogdan-Alexes-MacBook-Pro:curs7 bogdan$ ./a.out
Segmentation fault: 11
```

```
exempluStiva2.c
```

```
1 #include <stdio.h>
2
3
4 int f(int a,int b)
5 {
6     printf("a = %d, b = %d \n", a, b);
7     if (a<b)
8         return 1+f(a+1,b-1);
9     return 0;
10 }
11
12 int main()
13 {
14     printf("%d",f(0,1000000));
15     return 0;
16 }
```

```
a = 262048, b = 737952
a = 262049, b = 737951
a = 262050, b = 737950
a = 262051, b = 737949
Segmentation fault: 11
```

Cuprinsul cursului de azi

1. Funcții
2. Pointeri la funcții
3. Aritmetică pointerilor
4. Legătura dintre tablouri și pointeri

Harta simplificată a memoriei la rularea unui program

```
00 //include <stdio.h>
01
02 // variabile globale neinitializate
03 int g1,g2;
04
05 // variabile globale initialize
06 int g3=5, g4 = 7;
07
08 // variabile globale neinitializate
09 int g5, g6;
10
11 void f1() {
12     int var1, var2;
13     printf("In Stiva prin f1:\t\t %p %p\n",&var1,&var2);
14 }
15
16 void f2() {
17     int var1, var2;
18     printf("In Stiva prin f2:\t\t %p %p\n",&var1,&var2);
19     f1();
20 }
21
22 int main() {
23     //variabile locale
24     int var1, var2;
25     printf("In Stiva prin main:\t\t %p %p\n",&var1,&var2);
26     f2();
27     // variabile globale initialize + neinitializate
28     printf("Variabile globale neinitializate: %p %p\n",&g1,&g2);
29     printf("Variabile globale initialize: %p %p\n",&g3,&g4);
30     printf("Variabile globale neinitializate: %p %p\n",&g5,&g6);
31     //cod
32     printf("Text Data:\t\t %p %p \n\n",main,f1); ←
33     return 0;
34 }
```

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

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_pointer_functie** = 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 pointer de un anumit tip de date
- exemplu:
 - void (*pf)(int)
 - int (*pf)(int, int)
 - double (*pf)(int, double*)

Pointeri la funcții

```
exempluPointeriFunctii.c  x

1 #include <stdio.h>
2
3 int suma(int a, int b)
4 {
5     return a + b;
6 }
7
8 int diferență(int a, int b)
9 {
10    return a - b;
11 }
12
13 int main()
14 {
15     int (*pf)(int,int);
16     pf = &suma;
17     int s = (*pf)(2,5);
18     printf("s = %d \n",s);
19     pf = diferență;
20     int d = pf(2,5);
21     printf("d = %d \n",d);
22     return 0;
23 }
```

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

```
[Bogdan-Alexes-MacBook-Pro:curs7 bogdan$ gcc exempluPointeriFunctii.c
[Bogdan-Alexes-MacBook-Pro:curs7 bogdan$ ./a.out
s = 7
d = -3
```

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

```
exempluPointeriFunctii2.c  x

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

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

```
[Bogdan-Alexes-MacBook-Pro:curs7 bogdan$ gcc exempluPointeriFunctii2.c
[Bogdan-Alexes-MacBook-Pro:curs7 bogdan$ ./a.out
```

S1 = 15

S2 = 55

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

Functia qsort (și în cursul 4)

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

adresele a două elemente din tablou

Cmp este o funcție generică comparator, compară 2 elemente de orice tip din tablou. Întoarce:

- un număr < 0 dacă vrem elementul de la adresa **a** la stânga (înaintea) elementului de la adresa **b**
- un număr >0 dacă vrem elementul de la adresa **a** la dreapta (după) elementului de la adresa **b**
- 0, dacă nu contează

Funcția qsort pentru întregi

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

Exemplu de funcție cmp pentru sortarea unui vector de numere
întregi:

```
int cmp(const void* a, const void *b)  
{  
    int va, vb;  
    va = *(int*)a;  
    vb = *(int*)b;  
    if(va < vb) return -1;  
    if(va > vb) return 1;  
    return 0;  
}
```



```
int cmp(const void* a, const void *b)  
{  
    return *(int*)a - *(int*)b;  
}
```

Funcția qsort pentru întregi

```
exempluqsort.c
```

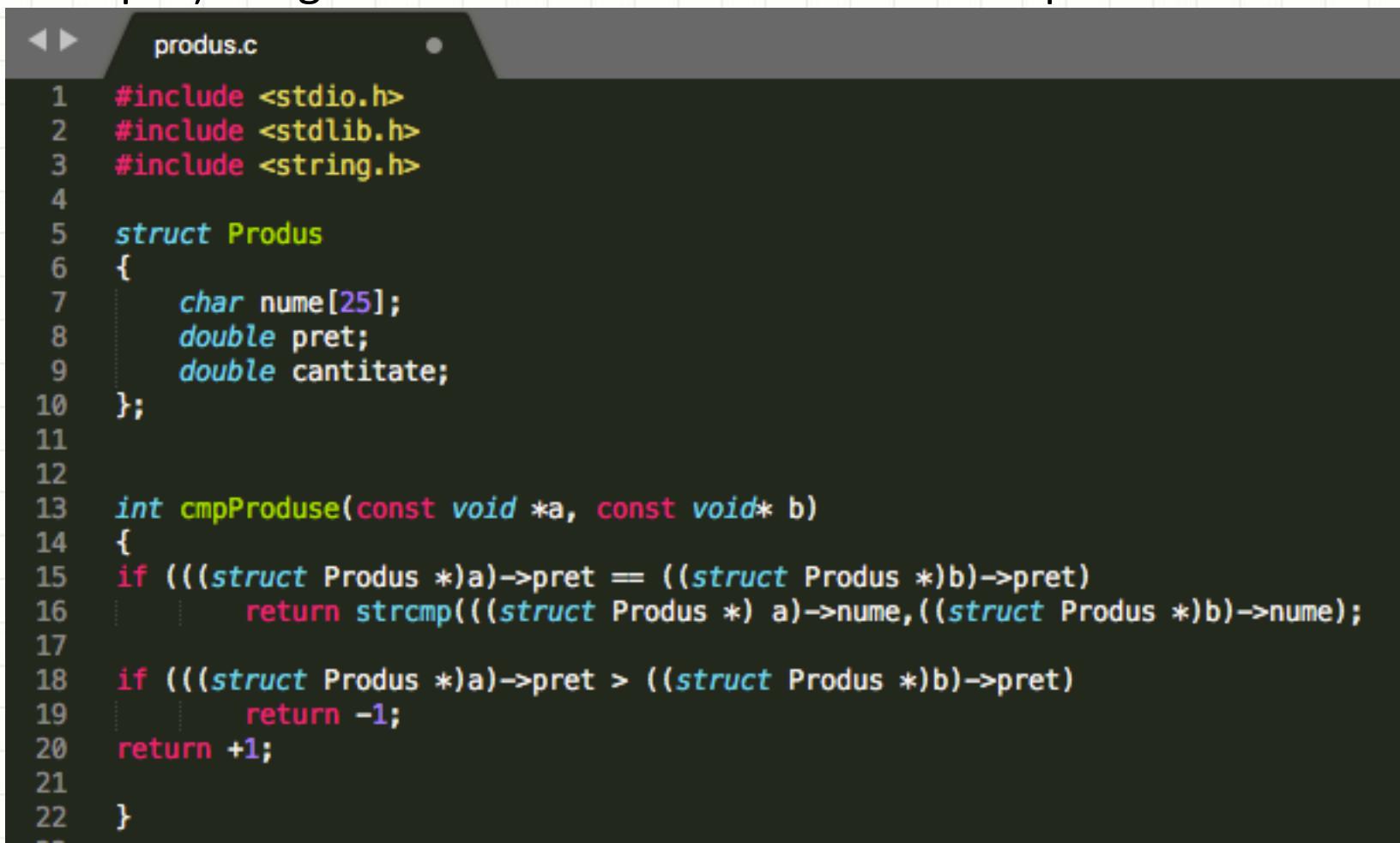
```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int cmp(const void* a, const void *b)
5 {
6     return *(int*)a - *(int*)b;
7 }
8
9 int main()
10 {
11     int v[] = {0, 5, -6, 9, 7, 12, 8, 7, 4};
12     qsort(v, 9, sizeof(int), cmp);
13     for( int i = 0; i < sizeof(v)/sizeof(int); i++)
14         printf("%d \t", v[i]);
15     printf("\n");
16     return 0;
17 }
```

```
[Bogdan-Alexes-MacBook-Pro:curs5 bogdan$ ./a.out
```

```
-6      0      4      5      7      7      8      9      12
```

Funcția qsort pentru structuri

Exercițiu seminar: avem o structură care reține numele, prețul, cantitatea pentru fiecare produs dintr-un magazin. Se dă un vector de asemenea produse pe care vreau să îl sorteze descrescător după preț și în caz de prețuri egale în ordinea alfabetică a numelor produselor.



```
produs.c

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 struct Produs
6 {
7     char nume[25];
8     double pret;
9     double cantitate;
10 };
11
12
13 int cmpProduse(const void *a, const void* b)
14 {
15     if (((struct Produs *)a)->pret == ((struct Produs *)b)->pret)
16         return strcmp(((struct Produs *) a)->nume,((struct Produs *)b)->nume);
17
18     if (((struct Produs *)a)->pret > ((struct Produs *)b)->pret)
19         return -1;
20     return +1;
21 }
22 }
```

Exercițiu seminar 5

Să se calculeze integrala definită (între a și b) a unei funcții f: R → R

$$\int_a^b f(x) dx$$

```
double calculeazaIntegralaDefinita(double a, double b, double precizie, double (*pf)(double))
```

Soluție: calculez suma Riemann superioară și inferioară pe o diviziune cu un pas foarte mic până când diferența dintre cele două sume este mai mică decât precizia pe care o doresc.

Cuprinsul cursului de azi

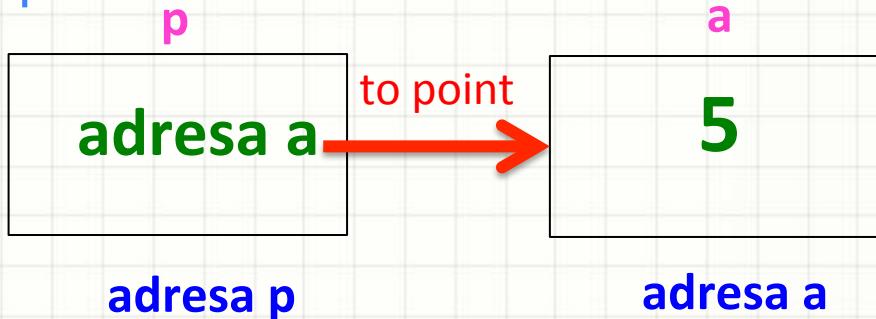
1. Funcții
2. Pointeri la funcții
3. Aritmetică pointerilor
4. Legătura dintre tablouri și pointeri

Aritmetica pointerilor

- un pointer: variabilă care poate stoca adrese de memorie

- exemple:

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

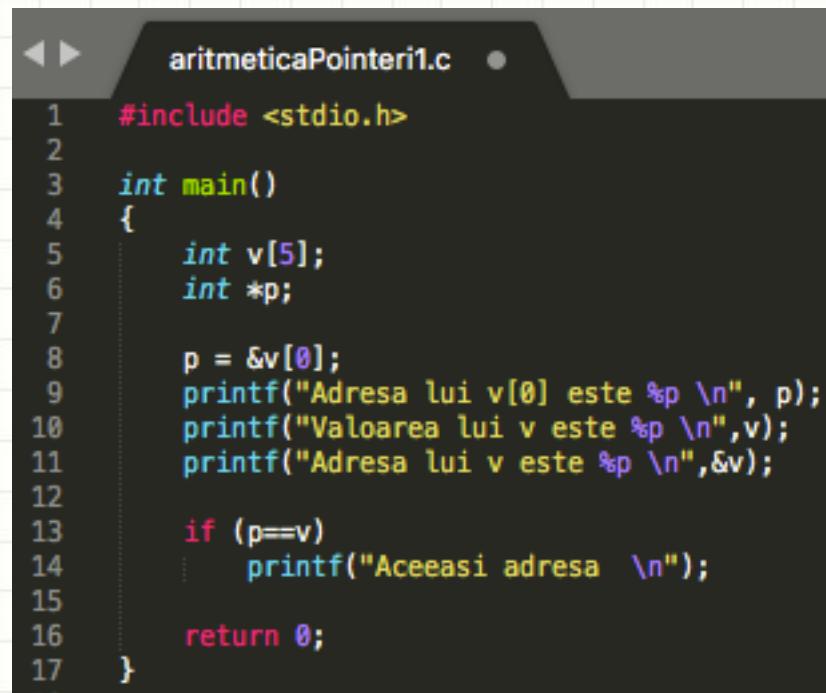


- asupra pointerilor pot fi realizate operații aritmetice:

- 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



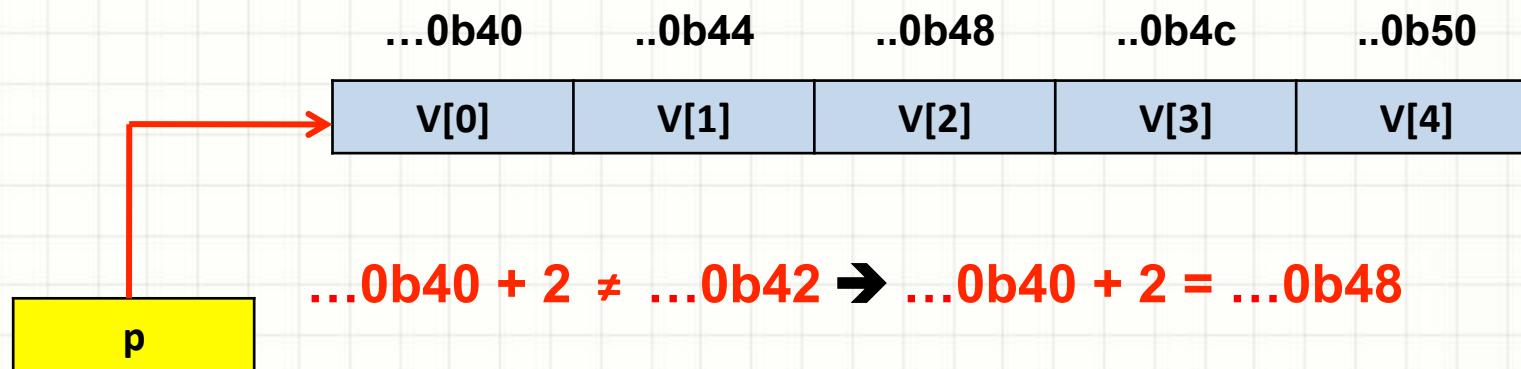
```
aritmeticaPointeri1.c
1 #include <stdio.h>
2
3 int main()
4 {
5     int v[5];
6     int *p;
7
8     p = &v[0];
9     printf("Adresa lui v[0] este %p \n", p);
10    printf("Valoarea lui v este %p \n",v);
11    printf("Adresa lui v este %p \n",&v);
12
13    if (p==v)
14        printf("Acleeasi adresa \n");
15
16    return 0;
17 }
```

v este un pointer care
pointeaza către v[0]

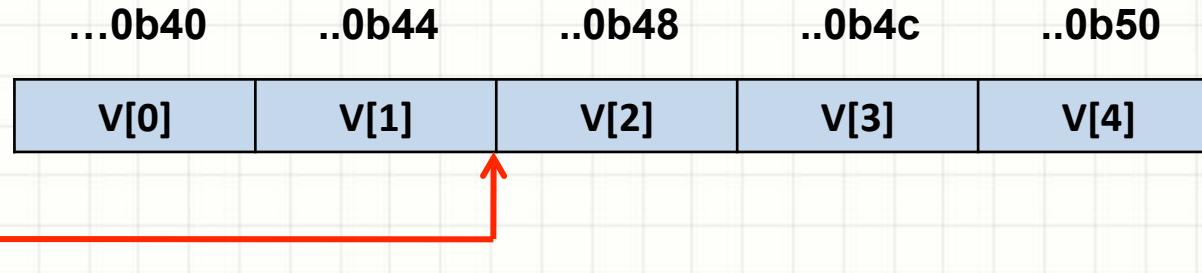
```
[Bogdan-Alexes-MacBook-Pro:curs7 bogdan$ gcc aritmeticaPointeri1.c
[Bogdan-Alexes-MacBook-Pro:curs7 bogdan$ ./a.out
Adresa lui v[0] este 0x7fff5e600b40
Valoarea lui v este 0x7fff5e600b40
Adresa lui v este 0x7fff5e600b40
Acleeasi adresa
```

Aritmetică pointerilor

- adunarea/scăderea unui număr natural dintr-un pointer



- în aritmetică 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

```
aritmeticaPointeri2.c  x

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

Aritmetica pointerilor

- adunarea/scăderea unui număr natural dintr-un pointer

The screenshot shows a terminal window with the following text:

```
Bogdan-Alexes-MacBook-Pro:curs7 bogdan$ gcc aritmeticaPointeri2.c
Bogdan-Alexes-MacBook-Pro:curs7 bogdan$ ./a.out
Adresa lui v[0] este 0x7fff51192b40
Valoarea lui v este 0x7fff51192b40
Adresa lui v este 0x7fff51192b40
Aceeași adresa
0x7fff51192b40
0x7fff51192b44
0x7fff51192b48
0x7fff51192b4c
0x7fff51192b50
Adresa spre care poateaza acum p este 0x7fff51192b48
```

The terminal output shows the execution of the C program 'aritmeticaPointeri2.c'. The program declares an integer array 'v' of size 5 and a pointer 'p' to its first element. It prints the address of the first element, the value at that address, and the address of the array itself. Then it increments the pointer by 2 and prints the new address, value, and array address again. Finally, it returns 0. The terminal also shows the command used to compile ('gcc') and run ('./a.out') the program.

```
aritmeticaPointeri2.c  x
1 #include <stdio.h>
2
3 int main()
4 {
5     int v[5];
6     int *p;
7
8     p = &v[0];
9     printf("Adresa lui v[0] este %p \n", p);
10    printf("Valoarea lui v este %p \n", v);
11    printf("Adresa lui v este %p \n", &v);
12
13    if (p==v)
14        printf("Acceași adresa \n");
15
16    printf("%p \n %p \n", p, v);
17
18    p = p + 2;
19    printf("Adresa spre ");
20
21    return 0;
22 }
```

Aritmetica pointerilor

- adunarea/scăderea unui număr natural dintr-un pointer

```
aritmeticaPointeri3.c  x

01 #include <stdio.h>
02
03 int main()
04 {
05     int v[5];
06     int *p;
07
08     p = &v[0];
09     printf("Adresa spre care pointeaza acum p este %p \n",p);
10
11     p += 2;
12     printf("Adresa spre care pointeaza acum p este %p \n",p);
13
14     p++;
15     printf("Adresa spre care pointeaza acum p este %p \n",p);
16
17     p -= 3;
18     printf("Adresa spre care pointeaza acum p este %p \n",p);
19
20
21 }
```

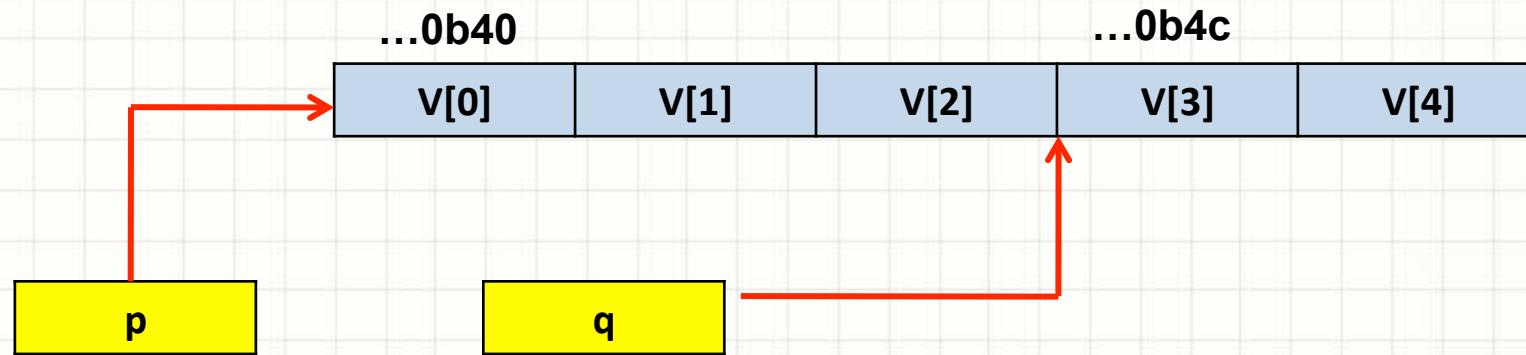
```
[Bogdan-Alexes-MacBook-Pro:curs7 bogdan$ gcc aritmeticaPointeri3.c
[Bogdan-Alexes-MacBook-Pro:curs7 bogdan$ ./a.out
Adresa spre care pointeaza acum p este 0xffff50f3fb40
Adresa spre care pointeaza acum p este 0xffff50f3fb48
Adresa spre care pointeaza acum p este 0xffff50f3fb4c
Adresa spre care pointeaza acum p este 0xffff50f3fb40
```

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



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

Aritmetică pointerilor

- compararea a două variabile de tip pointer
 - $p - q > 0$ înseamnă că p e la dreapta lui q
 - $p - q < 0$ înseamnă că p e la stânga lui q
- compararea a două variabile de tip pointer = compararea diferenței lor cu 0

Aritmetică pointerilor

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

The screenshot shows a code editor window with a dark theme. The title bar of the window says "aritmeticaPointeri4.c". The code itself is as follows:

```
#include <stdio.h>

int main()
{
    double a = 3.14, b = 2*a;
    printf("a = %f\n b = %f\n",a, b);

    double *p = &b;
    *p = 5.2;
    *(p+1) = 6.4;
    *(p+2) = 100.2;

    printf("a = %f\n b = %f\n",a, b);

    printf("&a = %p \n &b = %p \n",&a, &b);
    return 0;
}
```

Aritmetică pointerilor

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

The screenshot shows a terminal window with the following content:

```
aritmeticaPointeri4.c  x
1 #include <stdio.h>
2
3 int main()
4 {
5     double a = 3.14, b = 2*a;
6
7     printf("a = %f\n b = %f\n",a, b);
8
9     double *p = &b;
10    *p = 5.2;
11    *(p+1) = 6.4;
12    *(p+2) = 100.2;
13
14    printf("a = %f\n b = %f\n",a, b);
15
16    printf("&a = %p \n &b = %p\n", &a, &b);
17    return 0;
18 }
```

[Bogdan-Alexes-MacBook-Pro:curs7 bogdan\$ gcc aritmeticaPointeri4.c
[Bogdan-Alexes-MacBook-Pro:curs7 bogdan\$./a.out
a = 3.140000
b = 6.280000
a = 6.400000
b = 5.200000
&a = 0x7fff56c3eb50
&b = 0x7fff56c3eb48

Cuprinsul cursului de azi

1. Funcții
2. Pointeri la funcții
3. Aritmetică pointerilor
4. Legătura dintre tablouri și pointeri