PROGRAMARE PROCEDURALĂ

Bogdan Alexe
bogdan.alexe@fmi.unibuc.ro

Secția Informatică, anul I, seria 14,
2018-2019

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, șiruri de caractere, structuri, uniuni, câmpuri de biți, enumerări, pointeri
- Instrucțiuni de control
- Directive de preprocesare. 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
- Aritmetica pointerilor
- Legătura dintre tablouri și pointeri
- Alocarea dinamică a memoriei
- Clase de memorare

Șiruri 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 cu număr variabil de argumente
2. Programare generică

Funcții cu număr variabil de argumente

- funcții cu număr variabil de argumente utilizate de către voi până acum:
  - printf, fprintf
  - scanf, fscanf

```
int a, b, c;
scanf("%d %d", &a, &b);
printf("a = %d \n b = d \n", a ,b);
scanf("%d", &c);
printf("\n\n");
printf("c = %d\n", c);
```

Funcții cu număr variabil de argumente

- probleme:
  - nu se cunoaște numărul parametrilor funcției
  - nu se cunoaște tipul parametrilor
- header-ul stdarg.h cuprinde:
  - definiția unui tip de date specializat (va_list) dedicat manipulării listelor cu număr variabil de parametri
  - macro-uri (va_start, va_arg, va_end) care realizează operații pe acest tip de date
  - macro = fragment de cod căruia i se asociază un nume, la preprocesare se înlocuiește numele cu fragmentul de cod.

Funcții cu număr variabil de argumente

- **sintaxa:**

tip_rezultat nume_functie(tip_argument nume_argument, …)

unde (***cel putin***) primul argument este întotdeauna fix, vizibil, restul argumentelor fiind declarate prin *cele trei puncte*, (...) – *mecanismul elipsă*

- **exemplu:** funcție ce calculează suma a n numere întregi

int suma(int n,…);

suma(4,1,2,1,1) -> 5; suma(5,1,2,1,1,3) ->8

Funcții cu număr variabil de argumente

- macro-urile din stdarg.h:
  - va_list: tip de date dedicat manipulării listelor cu număr variabil de parametri (de obicei e unsigned char*)
  - va_start(va_list lp, numeArgument) : extrage în lista lp parametrii funcției care urmează după ultimul parametru fix specificat de numeArgument;
  - va_arg(va_list lp, tip_de_date): extrage la fiecare apel câte o valoare din lista lp – valoarea se consideră de tipul tip_de_date indicat ca parametru (poate fi int sau double);
  - va_end(va_list lp): obligatoriu cand se incheie operatiile pe lista lp

Funcții cu număr variabil de argumente

- **exemplul 1:** funcție ce calculează suma a n numere întregi

exemplu1.c

```c
#include <stdio.h>
#include <stdarg.h>
#include <string.h>

int suma(int n,...)
{
    int i, s;
    va_list listaParametri;
    va_start(listaParametri,n);
    s = 0;
    for(i = 0; i < n; i++)
        s = s + va_arg(listaParametri, int);
    va_end(listaParametri);
    return s;
}

int main()
{
    int a;
    a = suma(4,1,2,1,1);
    printf("a = %d\n", a);
    a = suma(5,1,2,1,1,3);
    printf("a = %d \n", a);
    return 0;
}
```

Funcții cu număr variabil de argumente

- **exemplul 1:** funcție ce calculează suma a n numere întregi

exemplu1.c

```c
#include <stdio.h>
#include <stdarg.h>
#include <string.h>

int suma(int n,...)
{
    int i, s;
    va_list listaParametri;
    va_start(listaParametri,n);
    s = 0;
    for(i = 0; i < n; i++)
        s = s + va_arg(listaParametri, int);
    va_end(listaParametri);
    return s;
}

int main()
{
    int a;
    a = suma(4,1,2,1,1);
    printf("a = %d\n", a);
    a = suma(5,1,2,1,1,3);
    printf("a = %d \n", a);
    return 0;
}
```

```
Bogdan-Alexes-MacBook-Pro:curs11 bogdan$ gcc exemplu1.c
Bogdan-Alexes-MacBook-Pro:curs11 bogdan$ ./a.out
a = 5
a = 8
```

Funcții cu număr variabil de argumente

```
void f(int x, ...)
{
 va_list lp; //declara lista de parametri
 va_start(lp ,x); //initializeaza lista de parametri, trebuie sa stiu unde incepe, dupa x
  for( ; ; ){
      tip_de_date t = va_arg(lp, tip_de_date);//extrage parametrul curent
      ... }
  va_end(lp); //elibereaza memoria
}
```

- apelam functia f dintr-o alta functie g;
- f trebuie să știe ce parametri primește;
- folosim va_start care apelează ultimul parametru formal cunoscut(x) transmis și reținut în stivă.
- transmiterea parametrilor se face de la dreapta la stânga într-o stivă

Funcții cu număr variabil de argumente

```
void f(int x, …)
{
 va_list lp; //declara lista de parametri
 va_start(lp ,x); //initializeaza lista de parametri, trebuie sa stiu unde incepe, dupa x
  for( ; ; ){
      tip_de_date t = va_arg(lp, tip_de_date);//extrage parametrul curent
      … }
  va_end(lp); //elibereaza memoria
}
```

- va_start găsește adresa lui x din stivă (e macro și nu funcție, are acces la stiva bună!) și apoi din stivă ia fiecare argument transmis cu ajutorul lui va_arg;
- va_arg trebuie să știe ce dimensiune în octeți are parametrul pe care trebuie să îl extragă din stivă;
- va_end este obligatoriu, altfel rezultatul e “undefined”;
- funcția f trebuie să știe unde se oprește cu citirea parametrilor.

Funcții cu număr variabil de argumente

- posibilă definire a macro-urilor va_list, va_start, va_arg:

```
typedef unsigned char * va_list;
#define  va_start(lp,param) (lp = (((va_list)&param) + sizeof(param)))
#define va_arg(lp,type) (*(type *)((lp += sizeof(type)) – sizeof(type)))
```

- va_list e un pointer la char (adresă de variabilă stocată pe un octet);
- va_start inițializează lista de argumente ca un pointer ce reține adresa imediată după ultimul parametru formal transmis în stivă. De la această adresa încep parametri în număr variabil;
- va_arg realizează două lucruri:
  - updateaza lista de argumente la urmatorul argument mutând pointerul (aritmetica pointerilor);
  - returneaza valoarea argumentului actual (se întoarce);

Funcții cu număr variabil de argumente

- **exemplul 1:** funcție ce calculează suma a n numere întregi

exemplu1.c

```c
#include <stdio.h>
#include <stdarg.h>
#include <string.h>

int suma(int n,...)
{
    int i, s;
    va_list listaParametri;
    va_start(listaParametri,n);
    s = 0;
    for(i = 0; i < n; i++)
        s = s + va_arg(listaParametri, int);
    va_end(listaParametri);
    return s;
}

int main()
{
    int a;
    a = suma(4,1,2,1,1);
    printf("a = %d\n", a);
    a = suma(5,1,2,1,1,3);
    printf("a = %d \n", a);
    a = suma(2,1,2,1,1,3);
    printf("a = %d \n", a);
    a = suma(7,1,2,1,1);
    printf("a = %d \n", a);
    return 0;
}
```

```
Bogdan-Alexes-MacBook-Pro:curs11 bogdan$ gcc exemplu1.c
Bogdan-Alexes-MacBook-Pro:curs11 bogdan$ ./a.out
a = 5
a = 8
a = 3
a = -515301515
```

Funcții cu număr variabil de argumente

- **exemplul 2:** suma unui șir de numere întregi ce se temină cu 0

exemplu2.c

```c
#include <stdio.h>
#include <stdarg.h>
#include <string.h>

int suma(int x,...)
{
    if(x == 0)
        return 0;

    int t, s;
    va_list lp;
    va_start(lp,x);
    s = x;
    do
    {
        t = va_arg(lp,int);
        s = s + t;
    }
    while(t);
    va_end(lp);
    return s;
}

int main()
{
    int a;
    a = suma(4,1,2,1,1,0);
    printf("a = %d\n", a);

    return 0;
}
```

```
Bogdan-Alexes-MacBook-Pro:curs11 bogdan$ gcc exemplu2.c
Bogdan-Alexes-MacBook-Pro:curs11 bogdan$ ./a.out
a = 9
```

Funcții cu număr variabil de argumente

- **exemplul 3:** maximul a n numere întregi

exemplu3.c

```c
#include <stdio.h>
#include <stdarg.h>
#include <string.h>

int maxim(int n,...)
{
    int max, i, aux;
    va_list lp;
    va_start(lp,n);
    max = va_arg(lp,int);
    for(i=2; i<=n;i++)
    {
        aux = va_arg(lp,int);
        if(max < aux)
            max = aux;
    }
    va_end(lp);
    return max;
}

int main()
{
    int a;
    a = maxim(7,1,2,19,5,7,4,3);
    printf("a = %d\n", a);

    return 0;
}
```

```
Bogdan-Alexes-MacBook-Pro:curs11 bogdan$ gcc exemplu3.c
Bogdan-Alexes-MacBook-Pro:curs11 bogdan$ ./a.out
a = 19
```

Funcții cu număr variabil de argumente

- **exemplul 4:** concatenarea unui număr variabil de șiruri de caractere într-un singur șir alocat dinamic. Marcăm sfârșitul șirurilor printr-un șir vid.

```
6   char *concateneazaSiruri(const char *primulSir, ...)
7   {
8       va_list listaParametri;
9       char *p,*q;
10
11      if(primulSir == NULL) return NULL;
12      int lungimeSir = strlen(primulSir);
13      va_start(listaParametri, primulSir);
14      while((p = va_arg(listaParametri, char *)) != NULL)
15          lungimeSir += strlen(p);
16      va_end(listaParametri);
17      q = (char *) malloc(lungimeSir + 1);
18      if(q == NULL) return NULL;
19      strcpy(q, primulSir);
20      va_start(listaParametri, primulSir);
21      while((p = va_arg(listaParametri, char *)) != NULL)
22          strcat(q, p);
23      va_end(listaParametri);
24      return q;
25  }
```

Funcții cu număr variabil de argumente

exemplu4.c

```c
#include <stdio.h>
#include <stdarg.h>
#include <string.h>
#include <stdlib.h>

char *concateneazaSiruri(const char *primulSir, ...)
{
    va_list listaParametri;
    char *p,*q;

    if(primulSir == NULL) return NULL;
    int lungimeSir = strlen(primulSir);
    va_start(listaParametri, primulSir);
    while((p = va_arg(listaParametri, char *)) != NULL)
        lungimeSir += strlen(p);
    va_end(listaParametri);
    q = (char *) malloc(lungimeSir + 1);
    if(q == NULL) return NULL;
    strcpy(q, primulSir);
    va_start(listaParametri, primulSir);
    while((p = va_arg(listaParametri, char *)) != NULL)
        strcat(q, p);
    va_end(listaParametri);
    return q;
}

int main()
{
    char *str = concateneazaSiruri("Functiile cu numar variabil ", "de argumente sunt ", "foarte simple!!!", (char *)'\0');
    printf("%s \n",str);
    return 0;
```

```
Bogdan-Alexes-MacBook-Pro:curs11 bogdan$ gcc exemplu4.c
Bogdan-Alexes-MacBook-Pro:curs11 bogdan$ ./a.out
Functiile cu numar variabil de argumente sunt foarte simple!!!
```

Funcții cu număr variabil de argumente

- **probleme la seminar:**

Probleme propuse

1. Scrieţi o funcţie cu număr variabil de parametri care să concateneze mai multe şiruri de caractere, parcurgând o singură dată lista parametrilor variabili.

2. Scrieţi o funcţie cu număr variabil de parametri care să determine prefixul comun de lungime maximă al mai multor şiruri de caractere.

3. Scrieţi o funcţie cu număr variabil de parametri care să returneze numărul de apariţii ale unui număr întreg dat într-un şir de numere întregi. Folosind apeluri utile ale funcţiei definite anterior, scrieţi o funcţie care verifică dacă 4 numere întregi sunt distincte sau nu.

4. Scrieţi o funcţie cu număr variabil de parametri care să creeze un şir de caractere din mai multe caractere date.

5. Scrieţi o funcţie cu număr variabil de parametri care să creeze un tablou de numere întregi din mai multe numere întregi. Atenţie, funcţia trebuie să furnizeze şi lungimea tabloului!

Cuprinsul cursului de azi

1. Funcții cu număr variabil de argumente
2. Programare generică

Programarea generică

- paradigmă de programare în care codăm diferiți algoritmi pentru tipuri de date generice (neinstanțiate).

- exemplu: un algoritm de sortare al unui tablou de elemente (numere de tipul char, int, float, double, structuri) este invariant la tipurile de date.
- funcția **qsort** din stdlib.h
- putem particulariza funcția de comparare să lucreze cu date de tipul char, int, float, double sau tipuri de date derivate(structuri).

- în programare generică folosim pointeri generici (void *)
- asigură o mai mare generalitate în raport cu tipul de date a variabilelor procesate în cadrul functiilor

Pointeri generici

- pointeri la tipul void (pointer generic)
- pot stoca adresa de memorie a unui obiect oarecare
  - dimensiunea zonei de memorie indicate și interpretarea informației conținute nu sunt definite;
  - nu poate fi dereferențiat întrucât nu are aritmetică indusă

pointeriGenerici1.c

```c
#include<stdio.h>

int main()
{
  int i = 6;
  void *p;

  p = &i;

  printf("p pointeaza catre un int cu valoare = %d\n", *p);

  return 0;
}
```

```
Bogdan-Alexes-MacBook-Pro:curs13 bogdan$ gcc pointeriGenerici1.c
pointeriGenerici1.c:10:56: error: argument type 'void' is incomplete
  printf("p pointeaza catre un int cu valoare = %d\n", *p);
                                                       ^

1 error generated.
```

Pointeri generici

- pointeri la tipul void (pointer generic)
- pot stoca adresa de memorie a unui obiect oarecare
  - pentru a-l folosi (la dereferențiere) trebuie convertit la un alt tip de pointer prin operatorul cast (tip *)
  - un singur pointer poate pointa la diferite tipuri de date la momente diferite pe parcursul execuției unui program

pointeriGenerici2.c

```
1   #include<stdio.h>
2
3   int main()
4   {
5     int i = 6;
6     char c = 'a';
7     void *p;
8
9     p = &i;
10    printf("p pointeaza catre un int cu valoarea = %d\n", *(int*)p);
11
12    p = &c;
13    printf("p pointeaza catre un char cu valoarea = %c\n", *(char*)p);
14
15    return 0;
16  }
```

Pointeri generici

- pointeri la tipul void (pointer generic)
- pot stoca adresa de memorie a unui obiect oarecare
  - pentru a-l folosi (la dereferențiere) trebuie convertit la un alt tip de pointer prin operatorul cast (tip *)
  - un singur pointer poate pointa la diferite tipuri de date la momente diferite pe parcursul execuției unui program

pointeriGenerici2.c

```c
#include<stdio.h>

int main()
{
  int i = 6;
  char c = 'a';
  void *p;

  p = &i;
  printf("p pointeaza catre un int cu valoarea = %d\n", *(int*)p);

  p = &c;
  printf("p pointeaza catre

  return 0;
}
```

```
Bogdan-Alexes-MacBook-Pro:curs13 bogdan$ gcc pointeriGenerici2.c
Bogdan-Alexes-MacBook-Pro:curs13 bogdan$ ./a.out
p pointeaza catre un int cu valoarea = 6
p pointeaza catre un char cu valoarea = a
```

Utilitatea pointerilor la funcții

- se folosesc în programarea generică, realizăm apeluri de tip callback;
- o funcţie C transmisă, printr-un pointer, ca argument unei alte funcţii F se numeşte şi funcţie **“callback”**, pentru că ea va fi apelată “înapoi” de funcţia F
- **exemple:**

1. void **qsort**(void *adresa,int nr_elemente, int dimensiune_element, int (*cmp)(const void *, const void *)) ;
2. void* **bsearch**(const void *cheie, const void* adresa, int nr_elemente, int dimensiune_element, int (*cmp) (const void *, const void *))

Funcția generică de sortare qsort

- **funcția qsort din stdlib.h folosită pentru sortarea unui 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**

Funcția generică de sortare qsort

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

exempluqsort1.c

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

int cmp1(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 cmp2(const void* a, const void* b)
{
  return *(int *)a - *(int *)b;
}

int main()
{
  int i;
  int v[5] = {0, 5, -6, 9, 7 };
  qsort(v,5,sizeof(int),cmp1);
  for(i = 0; i< 5; i++)
    printf("%d ", v[i]);
  printf("\n");

  int w[5] = {0, 5, -6, 9, 7 };
  qsort(w,5,sizeof(int),cmp2);
  for(i = 0; i < 5; i++)
    printf("%d ", w[i]);
  printf("\n");

  return 0;
}
```

```
Bogdan-Alexes-MacBook-Pro:curs13 bogdan$ gcc exempluqsort1.c
Bogdan-Alexes-MacBook-Pro:curs13 bogdan$ ./a.out
-6 0 5 7 9
-6 0 5 7 9
```

exempluqsort2.c

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

int cmp1(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 cmp2(const void* a, const void* b)
{
  return *(int *)a - *(int *)b;
}

int main()
{
  int i;
  int v[5] = {INT_MIN+5, INT_MIN, INT_MAX-4, INT_MAX-1, INT_MAX};
  qsort(v,5,sizeof(int),cmp1);
  for(i = 0; i< 5; i++)
    printf("%d ", v[i]);
  printf("\n");

  int w[5] = {INT_MIN+5, INT_MIN, INT_MAX-4, INT_MAX-1, INT_MAX};
  qsort(w,5,sizeof(int),cmp2);
  for(i = 0; i < 5; i++)
    printf("%d ", w[i]);
  printf("\n");

  return 0;
}
```

```
Bogdan-Alexes-MacBook-Pro:curs13 bogdan$ gcc exempluqsort2.c
Bogdan-Alexes-MacBook-Pro:curs13 bogdan$ ./a.out
-2147483648 -2147483643 2147483643 2147483646 2147483647
2147483643 2147483646 2147483647 -2147483648 -2147483643
```

exempluqsort2.c

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

int cmp1(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 cmp2(const void* a, const void* b)
{
  return *(int *)a - *(int *)b;
}

int main()
{
  int i;
  int v[5] = {INT_MIN+5, INT_MIN, INT_MAX-4, INT_MAX-1, INT_MAX};
  qsort(v,5,sizeof(int),cmp1);
  for(i = 0; i< 5; i++)
    printf("%d ", v[i]);
  printf("\n");

  int w[5] = {INT_MIN+5, INT_MIN, INT_MAX-4, INT_MAX-1, INT_MAX};
  qsort(w,5,sizeof(int),cmp2);
  for(i = 0; i < 5; i++)
    printf("%d ", w[i]);
  printf("\n");

  return 0;
}
```

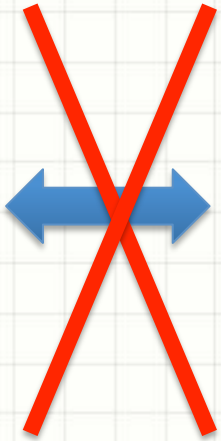
```
Bogdan-Alexes-MacBook-Pro:curs13 bogdan$ gcc exempluqsort2.c
Bogdan-Alexes-MacBook-Pro:curs13 bogdan$ ./a.out
2147483643 2147483646 2147483647 -2147483648 -2147483643
```

Funcția comparator pentru nr. întregi

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

Exemplu de funcție cmp pentru sortarea în ordine crescătoare a unui vector de numere întregi:

```
int cmp1(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 cmp2(const void* a, const void* b)
{
    return *(int *)a - *(int *)b;
}
```

Functia cmp2 poate face underflow sau overflow

Funcția comparator pentru nr. reale

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

Exemplu de funcție cmp pentru sortarea în ordine crescătoare a unui vector de numere reale:

```
int cmp1(const void* a, const void* b)
{
    double va, vb;
    va = *(double *)a;
    vb = *(double *)b;
    if(va < vb) return -1;
    if(va > vb) return 1;
    return 0;
}
```

```
int cmp2(const void* a, const void* b)
{
    return *(double *)a - *(double *)b;
}
```

Functia cmp2 poate face underflow sau overflow

Alt bug?

exempluqsort3.c

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

int cmp1(const void* a, const void* b)
{
  double va, vb;
  va = *(double *)a;
  vb = *(double *)b;
  if(va < vb) return -1;
  if(va > vb) return 1;
  return 0;
}

int cmp2(const void* a, const void* b)
{
  return *(double *)a - *(double *)b;
}

int main()
{
  int i;
  double v[5] = {0.2, 0.1, 0.5, -0.2,-0.3};
  qsort(v,5,sizeof(double),cmp1);
  for(i = 0; i< 5; i++)
    printf("%f ", v[i]);
  printf("\n");

  double w[5] = {0.2, 0.1, 0.5, -0.2,-0.3};
  qsort(w,5,sizeof(double),cmp2);
  for(i = 0; i < 5; i++)
    printf("%f ", w[i]);
  printf("\n");

  return 0;
}
```

```
Bogdan-Alexes-MacBook-Pro:curs13 bogdan$ gcc exempluqsort3.c
Bogdan-Alexes-MacBook-Pro:curs13 bogdan$ ./a.out
-0.300000 -0.200000 0.100000 0.200000 0.500000
0.200000 0.100000 0.500000 -0.200000 -0.300000
```

exempluqsort3.c

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

int cmp1(const void* a, const void* b)
{
  double va, vb;
  va = *(double *)a;
  vb = *(double *)b;
  if(va < vb) return -1;
  if(va > vb) return 1;
  return 0;
}

int cmp2(const void* a, const void* b)
{
  return *(double *)a - *(double *)b;
}

int main()
{
  int i;
  double v[5] = {0.2, 0.1, 0.5, -0.2,-0.3};
  qsort(v,5,sizeof(double),cmp1);
  for(i = 0; i< 5; i++)
    printf("%f ", v[i]);
  printf("\n");

  double w[5] = {0.2, 0.1, 0.5, -0.2,-0.3};
  qsort(w,5,sizeof(double),cmp2);
  for(i = 0; i < 5; i++)
    printf("%f ", w[i]);
  printf("\n");

  return 0;
}
```

```
Bogdan-Alexes-MacBook-Pro:curs13 bogdan$ gcc exempluqsort3.c
Bogdan-Alexes-MacBook-Pro:curs13 bogdan$ ./a.out
-0.300000 -0.200000 0.100000 0.200000 0.500000
0.200000 0.100000 0.500000 -0.200000 -0.300000
```

Funcția comparator pentru nr. reale

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

Exemplu de funcție cmp pentru sortarea în ordine crescătoare a unui vector de numere reale:

```
int cmp1(const void* a, const void* b)
{
    double va, vb;
    va = *(double *)a;
    vb = *(double *)b;
    if(va < vb) return -1;
    if(va > vb) return 1;
    return 0;
}
```

```
int cmp2(const void* a, const void* b)
{
    return *(double *)a - *(double *)b;
}
```

Functia cmp2 poate face underflow sau overflow

Face conversie implicită la int

Funcția comparator pentru lungimea șirurilor

Vrea să sortez un tablou de șiruri de caractere crescător după lungimea șirurilor

exempluqsort4.c

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

int cmpSiruri(const void* a, const void* b)
{
    if(strlen(*(char** )a) < strlen(*(char **)b)) return -1;
    if(strlen(*(char** )a) > strlen(*(char **)b)) return 1;
    return 0;
}

int main()
{
    char* nume[5] = {"Miruna","Claudiu","Ionut","Paul","Stefan"};
    int i;

    for(i=0;i<5;i++)
      printf("%p %p %s \n",nume+i, nume[i], nume[i]);

    qsort(nume,5,sizeof(char*), cmpSiruri);

    printf("Dupa qsort \n");

    for(i=0;i<5;i++)
      printf("%p %p %s \n",nume+i, nume[i], nume[i]);

    return 0;
}
```

Funcția comparator pentru lungimea șirurilor

Vrea să sortez un tablou de șiruri de caractere crescător după lungimea șirurilor

exempluqsort4.c

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

int cmpSiruri(const void* a, const void* b)
{
    if(strlen(*(char** )a) < strlen(*(char **)b)) return -1;
    if(strlen(*(char** )a) > strlen(*(char **)b)) return 1;
    return 0;
}

int main()
{
    char* nume[5] = {"Miruna","Claudiu","Ionut","Paul","Stefan"};
    int i;

    for(i=0;i<5;i++)
      printf("%p %p %s \n",nume+i

    qsort(nume,5,sizeof(char*), c

    printf("Dupa qsort \n");

    for(i=0;i<5;i++)
      printf("%p %p %s \n",nume+i

    return 0;
}
```

```
Bogdan-Alexes-MacBook-Pro:curs13 bogdan$ gcc exempluqsort4.c
Bogdan-Alexes-MacBook-Pro:curs13 bogdan$ ./a.out
0x7fff51838ae0 0x10e3c7f6a Miruna
0x7fff51838ae8 0x10e3c7f71 Claudiu
0x7fff51838af0 0x10e3c7f79 Ionut
0x7fff51838af8 0x10e3c7f7f Paul
0x7fff51838b00 0x10e3c7f84 Stefan
Dupa qsort
0x7fff51838ae0 0x10e3c7f7f Paul
0x7fff51838ae8 0x10e3c7f79 Ionut
0x7fff51838af0 0x10e3c7f6a Miruna
0x7fff51838af8 0x10e3c7f84 Stefan
0x7fff51838b00 0x10e3c7f71 Claudiu
```

Funcția comparator pentru structuri

Am 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 sortez descrescător după preț și în caz de prețuri egale în ordinea alfabetică a numelor produselor.

produs.c

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

struct Produs
{
    char nume[25];
    double pret;
    double cantitate;
};


int cmpProduse(const void *a, const void* b)
{
if (((struct Produs *)a)->pret == ((struct Produs *)b)->pret)
        return strcmp(((struct Produs *) a)->nume,((struct Produs *)b)->nume);

if (((struct Produs *)a)->pret > ((struct Produs *)b)->pret)
        return -1;
return +1;

}
```

Funcția generică de căutare binară bsearch

- **funcția bsearch (binary search) din stdlib.h folosită pentru căutare binară într-un *tablou SORTAT*.** Antetul lui bsearch este:

void* bsearch (const void *cheie, const void* adresa, int nr_elemente, int dimensiune_element, int (*cmp) (const void *, const void *))

- cheie = pointer cu adresa elementului pe care îl caut în tablou
- adresa = pointer cu adresa primului element al tabloului sortat în care căutăm
- nr_elemente = numarul de elemente al tabloului;
- dimensiune_element = dimensiunea in octeți a fiecărui element al tabloului (char = 1 octet, int = 4 octeți, etc)
- **cmp – funcția de comparare a două elemente**
- **returnează adresa către primul element găsit sau NULL**

Funcția generică de căutare binară bsearch

void* bsearch (const void *cheie, const void* adresa, int nr_elemente, int dimensiune_element, int (*cmp) (const void *, const void *))

int cmp(const void *a, const void *b)

cmp este o funcție generică comparator, compară 2 elemente (primul este cheia pe care o caut). Întoarce:

- un număr < 0 dacă vrem să căutăm cheia în stânga lui b
- un număr >0 dacă vrem să căutăm cheia în dreapta lui b
- 0, dacă s-a găsit cheia

Funcția bsearch - exemplu

exemplubsearch1.c

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

int cmp1(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 main()
{
    int i, j, v[6] = {10,1,12,3,14,5};
    qsort(v,6,sizeof(int),cmp1);
    printf("Vectorul sortat este \n");
    for(i=0; i< 6;i++)
        printf("%d ",v[i]);
    printf("\n");

    for(j=0;j<=5;j++)
    {
        int *p = (int *) bsearch(&j,v,6,sizeof(int),cmp1);
        if(p)
            printf("Elementul %d se gaseste in vector pe pozitia %ld \n",j,p-v);
        else
            printf("Elementul %d nu se gaseste in vector \n",j);
    }
    return 0;
}
```

aceeasi functie comparator folosita de qsort si bsearch

Funcția bsearch - exemplu

exemplubsearch1.c

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

int cmp1(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 main()
{
    int i, j, v[6] = {10,1,12,3,14,5};
    qsort(v,6,sizeof(int),cmp1);
    printf("Vectorul sortat este \n");
    for(i=0; i< 6;i++)
        printf("%d ",v[i]);
    printf("\n");

    for(j=0;j<=5;j++)
    {
        int *p = (int *) bsearch(
        if(p)
            printf("Elementul %d
        else
            printf("Elementul %d
    }
    return 0;
}
```

```
Bogdan-Alexes-MacBook-Pro:curs13 bogdan$ gcc exemplubsearch1.c
Bogdan-Alexes-MacBook-Pro:curs13 bogdan$ ./a.out
Vectorul sortat este
1 3 5 10 12 14
Elementul 0 nu se gaseste in vector
Elementul 1 se gaseste in vector pe pozitia 0
Elementul 2 nu se gaseste in vector
Elementul 3 se gaseste in vector pe pozitia 1
Elementul 4 nu se gaseste in vector
Elementul 5 se gaseste in vector pe pozitia 2
```

Funcția bsearch - exemplu

exemplubsearch2.c

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

int cmpSiruri(const void* a, const void* b)
{
    if(strlen(*(char** )a) < strlen(*(char **)b)) return -1;
    if(strlen(*(char** )a) > strlen(*(char **)b)) return 1;
    return 0;
}

int cmp2(const void* a, const void* b)
{
int l = *(int*)a;
int lSir = strlen(*(char **)b);
if (l == lSir)
  return 0;
if (l > lSir)
  return 1;
return -1;
}

int main()
{
    char* nume[5] = {"Miruna","Claudiu","Ionut","Paul","Stefan"};
    int i;

    qsort(nume,5,sizeof(char*), cmpSiruri);

    int lungimeSir = 5;
    char** p = bsearch(&lungimeSir,nume,5,sizeof(char *),cmp2);
    if (p)
    {
      printf("Am gasit sirul %s care are lungimea %d \n",*p,lungimeSir);
    }
    return 0;
}
```

functii comparator diferite folosite de qsort si bsearch

Funcția bsearch - exemplu

exemplubsearch2.c

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

int cmpSiruri(const void* a, const void* b)
{
    if(strlen(*(char** )a) < strlen(*(char **)b)) return -1;
    if(strlen(*(char** )a) > strlen(*(char **)b)) return 1;
    return 0;
}

int cmp2(const void* a, const void* b)
{
int l = *(int*)a;
int lSir = strlen(*(char **)b);
if (l == lSir)
  return 0;
if (l > lSir)
  return 1;
return -1;
}

int main()
{
    char* nume[5] = {"Miruna","Claudiu","Ionut","Paul","Stefan"};
    int i;

    qsort(nume,5,sizeof(char*), cmpSiruri);

    int lungimeSir = 5;
    char** p = bsearch(&lungimeSir,nume,5,sizeof(char *),cmp2);
    if (p)
    {
      printf("Am gasit sirul %s care
    }
    return 0;
}
```

```
Bogdan-Alexes-MacBook-Pro:curs13 bogdan$ gcc exemplubsearch2.c
Bogdan-Alexes-MacBook-Pro:curs13 bogdan$ ./a.out
Am gasit sirul Ionut care are lungimea 5
```

Programarea generică vs. specifică

- exemplul 1: scrieți o funcție generică de căutare care să returneze un pointer generic către prima apariție a valorii x în tabloul unidimensional t format din n elemente, fiecare având dimensiunea d octeți, sau pointerul NULL dacă valoarea x nu se găsește în tablou.

- funcție specifică care rezolvă problema pentru numere întregi

```
int lsearch(int x, int v[], int n)
{
  for(int i = 0; i< n; i++)
    if(v[i] == x)
      return i;

  return -1;
}
```

- ce se modifică și ce rămâne fix la implementarea generică?

Programarea generică vs. specifică

- ce se modifică și ce rămâne fix la implementarea generică?

```
int lsearch(int x, int v[], int n)
{
  for(int i = 0; i< n; i++)
    if(v[i] == x)
      return i;

  return -1;
}
```

- antetul trebuie să permită date de tip generic (void *)
- for-ul rămâne acolo
- tipul void* nu are aritmetică -> trebuie să precizez dimensiunea în octeți a unui element
- compararea a două elemente generice din tablou nu o pot face cu == (nu merge pe structuri, șiruri de caractere, etc)

Programarea generică vs. specifică

Varianta 1

```
void* lsearch1(void* x,void* t,int n, int d)
{
  char* p = t;
  for(int i=0; i< n; i++)
    if(memcmp(p+i*d,x,d)==0)
      return p+i*d;
  return NULL;
}
```

- lucrul cu o adresa spre un tip generic (void *) necesită transmiterea unui parametru suplimentar care sa exprime lungimea tipului de date a elementelor tabloului unidimensional

Funcțiile din string.h

`char* strcpy(char* d,const char* s)`	copiază şirul **s** în **d**, inclusiv **'\0'**, întoarce **d**
`char* strncpy(char* d,const char* s, int n)`	copiază **n** caractere din şirul **s** în **d**, completând eventual cu **'\0'**, întoarce **d**
`char* strcat(char* d,const char* s)`	concatenează şirul **s** la sfârşitul lui **d**, întoarce **d**
`char* strncat(char* d,const char* s, int n)`	concatenează cel mult **n** caractere din şirul **s** la sfârşitul lui **d**, completând cu **'\0'**, întoarce **d**
`int strcmp(const char* d, const char* s)`	compară şirurile **d** şi **s**, întoarce **-1** dacă **d<s**, **0** dacă **d==s** şi **1** dacă **d>s**
`int stricmp(const char* d, const char* s)`	compară şirurile **d** şi **s** (ca şi **strcmp()**) fără a face distincţie între litere mari şi mici
`int strncmp(const char* d, const char* s, int n )`	similar cu **strcmp()**, cu deosebirea că se compară cel mult **n** caractere
`char* strchr(const char* d,char c)`	caută caracterul **c** în şirul **d**; întoarce un pointer la prima apariţie a lui **c** în **d**, sau **NULL**
`char* strrchr(const char* d,char c)`	întoarce un pointer la ultima apariţie a lui **c** în **d**, sau **NULL**
`char* strstr(const char* d, const char* s)`	întoarce un pointer la prima apariţie a subşirului **s** în **d**, sau **NULL**
`char* strpbrk(const char* d, const char* s)`	întoarce un pointer la prima apariţie a unui caracter din subşirul **s** în **d**, sau **NULL**

Funcțiile din string.h

`int strspn(const char* d, const char* s)`	întoarce lungimea prefixului din **d** care conţine numai caractere din **s**
`int strcspn(const char* d, const char* s)`	întoarce lungimea prefixului din **d** care conţine numai caractere ce nu apar în **s**
`int strlen(const char* s)`	întoarce lungimea lui **s** (`'\0'` nu se numără)
`void* memcpy(void* d, const void* s,int n)`	copiaza **n** octeţi din **s** în **d**; întoarce **d**
`void* memmove(void* d, const void* s,int n)`	ca şi **memcopy**, folosită daca **s** şi **d** se întrepătrund
`void* memset(void* d,const int c, int n)`	copiază caracter **c** în primele **n** poziţii din **d**
`int memcmp(const void* d, const void* s,int n)`	compară zonele adresate de **s** şi **d**
`char* strtok(const char* d, const char* s)`	caută în **d** subşirurile delimitate de caracterele din **s**;primul apel întoarce un pointer la primul subşir din **d** care nu conţine caractere din **s** următoarele apeluri se fac cu primul argument **NULL**, întorcându-se de fiecare dată un pointer la următorul subşir din **d** ce nu conţine caractere din **s**; în momentul în care nu mai există subşiruri, funcţia întoarce **NULL**

Funcții predefinite pentru manipularea blocurilor de memorie

- compararea a două tablouri pe octeți – funcția **memcmp**
  - antet: int memcmp(const void *s1, const void * s2, int n);
  - compara primii **n** octeți corespondenți începând de la adresele **s1** și **s2**.
  - returnează 0 dacă octeții sunt identici, ceva mai mic decât 0 dacă **s1** $<_L$ **s2**, ceva mai mare decât 0 dacă **s1** $>_L$ **s2**

Funcții predefinite pentru manipularea blocurilor de memorie

- compararea a două tablouri pe octeți – funcția **memcmp**
  - antet: int memcmp(const void *s1, const void * s2, int n);
  - compara primii **n** octeți corespondenți începând de la adresele **s1** și **s2**.

exempluMemcmp.c

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

int main ()
{
    char t[] = "Ana are mere!!!";
    char s[] = "Ana are pere!!!";
    int i = memcmp(t,s,6);
    printf("%d \n",i);
    i = memcmp(t,s,strlen(t));
    printf("%d \n",i);

    int v[] = {1,2,4,5,6};
    int w[] = {1,2,3,7,8};
    i = memcmp(v,w,8);
    printf("%d \n",i);
    i = memcmp(v,w,sizeof(v));
    printf("%d \n",i);

    return 0;
}
```

```
Bogdan-Alexes-MacBook-Pro:curs10 bogdan$ gcc exempluMemcmp.c
Bogdan-Alexes-MacBook-Pro:curs10 bogdan$ ./a.out
0
-3
0
1
```

Varianta 1

exemplul1.c

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

void* lsearch1(void* x,void* t,int n, int d)
{
  char* p = t;
  for(int i=0; i< n; i++)
    if(memcmp(p+i*d,x,d)==0)
      return p+i*d;
  return NULL;
}

int main()
{
  int v[6] = {1, 2, 3, 4, 5 ,6};
  int x=2;
  int* p = lsearch1(&x,v,6,sizeof(int));
  if(p)
    printf("Prima aparitie a lui %d in tabloul v este pe pozitia = %ld \n",x, p-v);
  else
    printf("%d nu apare in tabloul v\n",x);

  char* nume[5] = {"Miruna","Claudiu","Ionut","Paul","Stefan"};
  char* w = "Stefan";
  char** q = lsearch1(w,nume,5,sizeof(char *));
  if(q)
    printf("Prima aparitie a lui %s in tabloul nume este pe pozitia = %ld \n",w, q-nume);
  else
    printf("%s nu apare in tabloul nume\n",w);


  return 0;
}
```

Varianta 1

exemplul1.c

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

void* lsearch1(void* x,void* t,int n, int d)
{
  char* p = t;
  for(int i=0; i< n; i++)
    if(memcmp(p+i*d,x,d)==0)
      return p+i*d;
  return NULL;
}

int main()
{
  int v[6] = {1, 2, 3, 4, 5 ,6};
  int x=2;
  int* p = lsearch1(&x,v,6,sizeof(int));
  if(p)
    printf("Prima aparitie a lui %d in tabloul v este pe pozitia = %ld \n",x, p-v);
  else
    printf("%d nu apare in tabloul v\n",x);

  char* nume[5] = {"Miruna","Claudiu","Ionut","Paul","Stefan"};
  char* w = "Stefan";
  char** q = lsearch1(w,nume,5,sizeof(char *));
  if(q)
    printf("Prima aparitie a lui %s in tabloul nume este pe pozitia = %ld \n",w, q-nume);
  else
    printf("%s nu apare in tabloul nume\n",w);


  return 0;
}
```

```
Bogdan-Alexes-MacBook-Pro:curs13 bogdan$ gcc exemplul1.c
Bogdan-Alexes-MacBook-Pro:curs13 bogdan$ ./a.out
Stefan nu apare in tabloul nume
```

Programarea generică vs. specifică

Varianta 1

```
void* lsearch1(void* x,void* t,int n, int d)
{
  char* p = t;
  for(int i=0; i< n; i++)
    if(memcmp(p+i*d,x,d)==0)
      return p+i*d;
  return NULL;
}
```

- funcția memcmp va compara adresele pointerilor și nu șirurile de caractere
- nu e potrivită pentru comparare generică
- folosim o funcție comparator

Programarea generică vs. specifică

Varianta 2

```
void* lsearch2(void* x,void* t,int n, int d, int (*cmp)(const void*,const void*))
{
  char* p = t;
  for(int i=0; i< n; i++)
    if(cmp(p+i*d,x)==0)
      return p+i*d;
  return NULL;
}
```

- folosim o funcție comparator
- definesc funcția comparator în funcție de ce am nevoie

exemplul1_var2.c

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

void* lsearch2(void* x,void* t,int n, int d, int (*cmp)(const void*,const void*))
{
  char* p = t;
  for(int i=0; i< n; i++)
    if(cmp(p+i*d,x)==0)
      return p+i*d;
  return NULL;
}

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


int cmpSiruri(const void* a, const void* b)
{
  return strcmp(*(char **)a, (char *)b);
}

int main()
{
  int v[6] = {1, 2, 3, 4, 5 ,6};
  int x=2;
  int* p = lsearch2(&x,v,6,sizeof(int),cmpIntregi);
  if(p)
    printf("Prima aparitie a lui %d in tabloul v este pe pozitia = %ld \n",x, p-v);
  else
    printf("%d nu apare in tabloul v\n",x);

  char* nume[5] = {"Miruna","Claudiu","Ionut","Paul","Stefan"};
  char* w = "Stefan";
  char** q = lsearch2(w,nume,5,sizeof(char *),cmpSiruri);
  if(q)
    printf("Prima aparitie a lui %s in tabloul nume este pe pozitia = %ld \n",w, q-nume);
  else
    printf("%s nu apare in tabloul nume\n",w);
  return 0;
}
```

exemplul1_var2.c

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

void* lsearch2(void* x,void* t,int n, int d, int (*cmp)(const void*,const void*))
{
  char* p = t;
  for(int i=0; i< n; i++)
    if(cmp(p+i*d,x)==0)
      return p+i*d;
  return NULL;
}

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


int cmpSiruri(const void* a, const void* b)
{
  return strcmp(*(char **)a, (char *)b);
}

int main()
{
  int v[6] = {1, 2, 3, 4, 5 ,6};
  int x=2;
  int* p = lsearch2(&x,v,6,sizeof(int),cmpIntregi);
  if(p)
    printf("Prima aparitie a lui %d in tabloul v este pe pozitia = %ld \n",x, p-v);
  else
    printf("%d nu apare in tabloul v\n",x);

  char* nume[5] = {"Miruna","Claudiu","Ionut","Paul","Stefan"};
  char* w = "Stefan";
  char** q = lsearch2(w,nume,5,sizeof(char *),cmpSiruri);
  if(q)
    printf("Prima aparitie a lui %s in tabl
  else
    printf("%s nu apare in tabloul nume\n",
  return 0;
}
```

```
Bogdan-Alexes-MacBook-Pro:curs13 bogdan$ gcc exemplul1_var2.c
Bogdan-Alexes-MacBook-Pro:curs13 bogdan$ ./a.out
Prima aparitie a lui 2 in tabloul v este pe pozitia = 1
Prima aparitie a lui Stefan in tabloul nume este pe pozitia = 4
```

```c
int cmpSiruri(const void* a, const void* b)
{
  return strcmp(*(char **)a, (char *)b);
}

int cmpSiruri2(const void* a, const void* b)
{
  return strcmp(*(char **)a, *(char **)b);
}


int main()
{
  int v[6] = {1, 2, 3, 4, 5 ,6};
  int x=2;
  int* p = lsearch2(&x,v,6,sizeof(int),cmpIntregi);
  if(p)
    printf("Prima aparitie a lui %d in tabloul v este pe pozitia = %ld \n",x, p-v);
  else
    printf("%d nu apare in tabloul v\n",x);

  char* nume[5] = {"Miruna","Claudiu","Ionut","Paul","Stefan"};
  char* w = "Stefan";
  char** q = lsearch2(w,nume,5,sizeof(char *),cmpSiruri);
  if(q)
    printf("Prima aparitie a lui %s in tabloul nume este pe pozitia = %ld \n",w, q-nume);
  else
    printf("%s nu apare in tabloul nume\n",w);

  q = lsearch2(&w,nume,5,sizeof(char *),cmpSiruri2);
  if(q)
    printf("Prima aparitie a lui %s in tabloul nume este pe pozitia = %ld \n",w, q-nume);
  else
    printf("%s nu apare in tabloul nume\n",w);
  return 0;
```

```
Bogdan-Alexes-MacBook-Pro:curs13 bogdan$ gcc exemplul1_var2.c
Bogdan-Alexes-MacBook-Pro:curs13 bogdan$ ./a.out
Prima aparitie a lui 2 in tabloul v este pe pozitia = 1
Prima aparitie a lui Stefan in tabloul nume este pe pozitia = 4
Prima aparitie a lui Stefan in tabloul nume este pe pozitia = 4
```