Programarea Calculatoarelor
Cursul 7

Pointeri la pointeri
Alocare dinamică
Pointeri la funcții

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

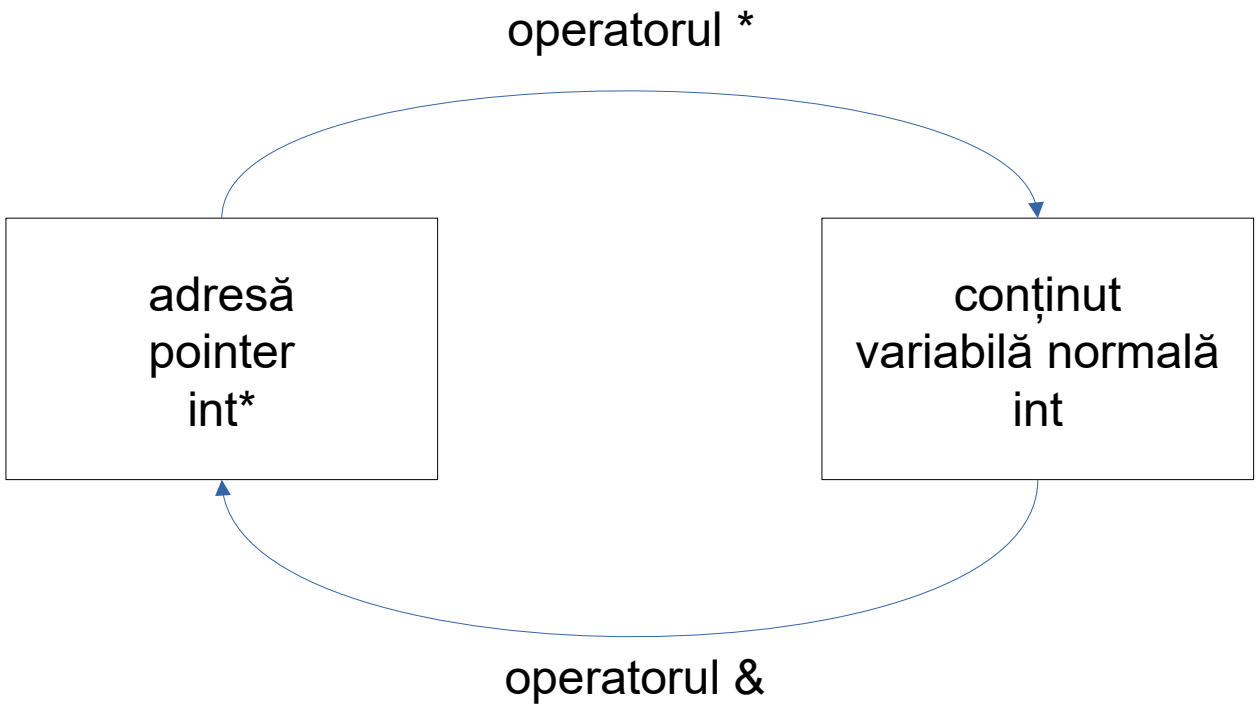
Obiective

- Legătură pointeri și tablouri multidimensionale
  - pointeri la pointeri
- Cum putem crea variabile de dimensiuni mari?
- Ce este un pointer la o funcție?

Pointeri - recapitulare

- Definiție
  - Un pointer este menit să conțină adresa din memorie a unei variabile
  - Este un număr întreg care reprezintă poziția în memorie
    - În general este un segment și offset
  - Este legat la un tip anume
  - Nu este controlat sau verificat
- Adresa unei variabile se poate obține utilizând operatorul unar referință &
- Adresele se afișează cu specificatorul de format %p
- Valoarea stocată la adresa de memorie referită de un pointer se poate determina utilizând operatorul unar dereferențiere *

Pointeri - recapitulare



Pointeri la pointeri

- Putem defini pointeri la alte variabile care sunt pointeri
- Numărul de legături poate fi oricât

```
tip** nume          int** ppx;
```

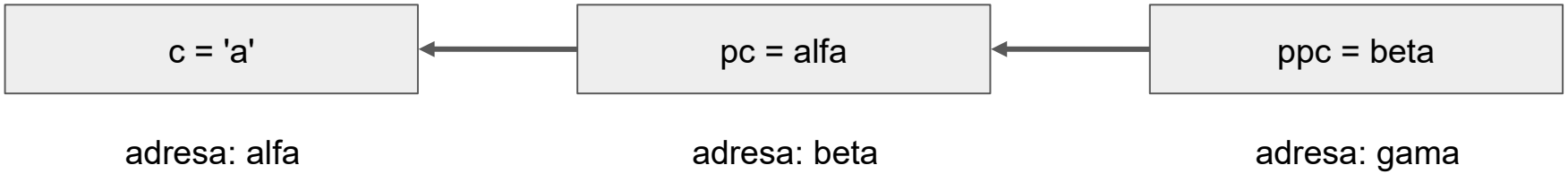
- `nume` este dublu pointer la `tip`
- `nume` este un pointer la un pointer la `tip`
- Numărul de modificatori * indică de câte ori este pointer
  - simplu, dublu, triplu, … de ordin k
- La fiecare dereferențiere se renunță la un ordin
  - un asterisk * de la dereferențiere anulează un asterisk * de la tip

Pointeri la pointeri - exemplu simplu

```c
#include <stdio.h>

int main(){
    char c = 'a';
    char* pc = &c;    //alfa
    char** ppc = &pc; //beta
    printf("%p %p %c", ppc, *ppc, **ppc);
    return 0;
}
```

- *c* este o variabilă de tip char
- *pc* este un pointer la char
- *ppc* este un pointer la un pointer la char
- afișăm adresa lui *pc*, adresa lui *c* și *c*



Pointeri la pointeri

- Utilitate
  - creare tablouri multidimensionale cu dimensiuni variabile
    - un pointer la pointer poate reprezenta primul element dintr-un șir de pointeri
  - modificare pointeri în funcții
    - pentru a modifica un pointer trebuie trimisă adresa lui = dublu pointer

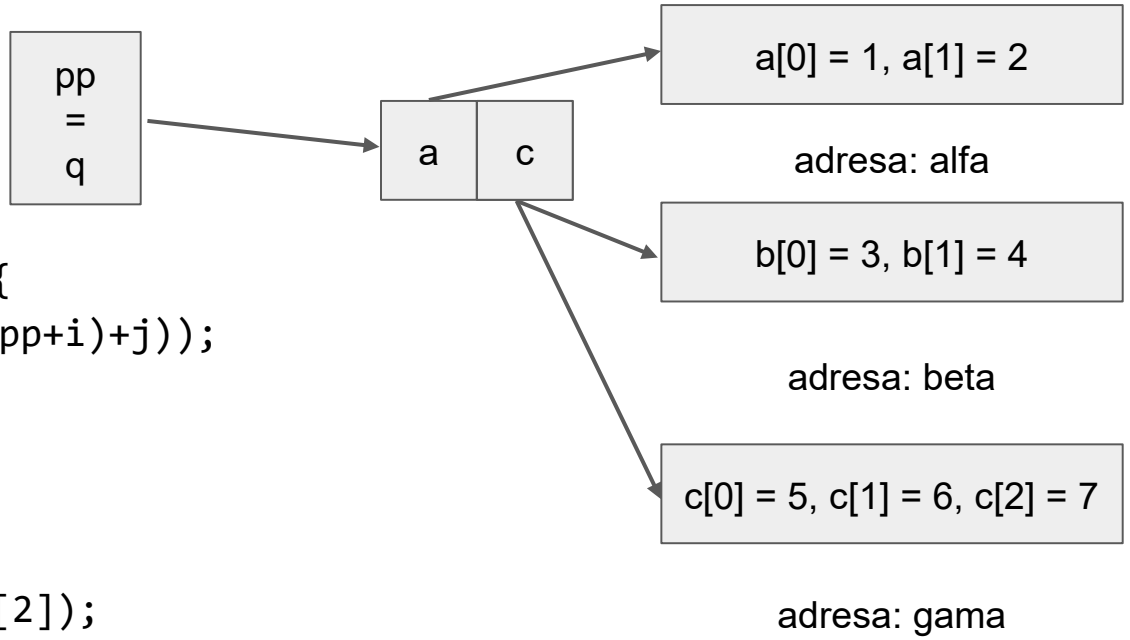
```
qii`pqfqitlxd!`
xmcoemge`yghsbx
w!rbyffscvjngco
pedtyvgosgn{wru
```

Pointeri la pointeri - exemplu complex

```c
#include <stdio.h>

int main(){
    int a[2] = {1, 2};
    int b[2] = {3, 4};
    int c[3] = {5, 6, 7};
    int* pp[] = {a, b};
    int** q = pp;
    for(int i=0; i<2; i++){
        for(int j=0; j<2; j++){
            printf("%d ", *(*(pp+i)+j));
        }
    }
    q[1] = c;
    printf("%d %d %d",
        q[1][0], q[1][1], q[1][2]);
    return 0;
}
```

- *pp* va fi un șir de pointeri: primul element este un pointer la elementele din *a* și al doilea element este pointerul *b*
- schimbăm al doilea pointer să arate la începutul șirului din *c*



Zone diferite de memorie

- Stiva
  - variabilele locale, argumentele funcțiilor și adresele de revenire după apeluri trăiesc până la ieșirea din funcție
- Global
  - variabilele globale și cele statice trăiesc până la terminarea execuției programului
- Altele
  - codul, valorile returnate din funcții, șirurile de caractere constante
- Heap (grămadă? morman?)
  - o zonă predefinit
  - este administrată
  - zonele de memo

```
qii`pqfqitlxd!`
xmcoemge`yghsbx
w!rbyffscvjngco
pedtyvgosgn{wru
```

Avantaje/dezavantaje heap

- Avantaje
  - Durată de viață
    - programatorul controlează când are loc alocarea și dezalocarea memoriei
    - adresele variabilelor alocate dinamic se pot returna din funcții
  - Dimensiuni
    - variabilele de dimensiuni mari ~GB sunt alocate aproape exclusiv pe heap
- Dezavantaje
  - Lucru în plus
    - programatorul trebuie să apeleze funcțiile speciale
  - Sursă de bug-uri
    - un program care rulează mult timp și nu dezalocă memoria de care nu mai are nevoie = eng. memory leak
  - Memoria de pe stivă este gestionată în mod corect și automat

Funcții pentru gestionarea memoriei heap - stdlib.h

```
void* calloc(size_t num, size_t size);
```

- Funcția alocă o zonă continuă de memorie pe heap de dimensiune `num * size` bytes și o setează la 0
- `size_t` este echivalent cu `unsigned long`
- Parametrii
  - `size_t num` - numărul de elemente
  - `size_t size` - dimensiunea în bytes a unui singur element
- Valoare returnată
  - pointer `void` la începutul zonei de memorie alocate
    - pentru derefențiere se convertește implicit sau explicit la pointer la un tip specific (tip*)
  - pointerul NULL dacă nu se poate aloca memoria

Funcții pentru gestionarea memoriei heap - stdlib.h

```
void* malloc(size_t size);
```

- Funcția alocă o zonă continuă de memorie pe heap de dimensiune `size` bytes neinițializată (valori aleatoare)
- `size_t` este echivalent cu `unsigned long`
- Parametrii
  - `size_t size` - dimensiunea în bytes a zonei totale
- Valoare returnată
  - pointer `void` la începutul zonei de memorie alocate
    - pentru derefențiere se convertește implicit sau explicit la pointer la un tip specific (tip*)
  - pointerul NULL dacă nu se poate aloca memoria

Funcții pentru gestionarea memoriei heap - stdlib.h

```
void* realloc(void* block, size_t new_size);
```

- Funcția realocă o zonă continuă de memorie pe heap care deja a fost alocată
- Necesar dacă vrem să redimensionăm (micșorăm/mărim)
  - dacă este posibil se redimensionează zona alocată pe loc
  - dacă este nevoie se alocă o zonă nouă și se copiază conținutul vechi în limita dimensiunii noi
- Parametrii
  - `void* block` - pointer la zona alocată din heap prin malloc/calloc/realloc
  - `size_t new_size` - dimensiunea nouă în bytes a zonei totale
- Valoare returnată
  - pointer void la începutul zonei de memorie alocate
    - pentru derefențiere se convertește implicit sau explicit la pointer la un tip specific (tip*)
  - pointerul NULL dacă nu se poate aloca memoria

Funcții pentru gestionarea memoriei heap - stdlib.h

```
void free(void* block);
```

FLIP LSB

- Funcția dezalocă (eliberează) o zonă de memorie de pe heap care a fost alocată anterior prin malloc/calloc/realloc
- Parametrii
  - `void* block` - pointer la zona alocată din heap prin malloc/calloc/realloc
- Erori posibile
  - Pointer invalid sau zonă nealocată prin funcțiile din stdlib
  - Încercarea de a dezaloca o zonă deja dezalocată
  - Accesarea unei zone dezalocate
- Zonele neeliberate de pe heap se dezalocă automat la terminarea programului

Alocare dinamică - șablon alocare

```c
#include <stdio.h>
#include <stdlib.h>
int main(){
    int n = 1e6;
    float* a = calloc(n, sizeof(float));
    if (a == NULL){
        puts("eroare la alocare");
        return -1;
    }
    for(int i=0; i<n; i++)
        a[i] = 1.f/(1+i);
    a = realloc(a, 10*sizeof(float));
    if (a == NULL){
        puts("eroare la realocare");
        return -2;
    }
    for(int i=0; i<10; i++)
        printf("%f ", a[i]);
    free(a);
    return 0;
}
```

- *n* este 10^6, numărul double 1e6 fiind convertit implicit la *int*
- se alocă *n* float-uri pe heap
- `void*` este convertit implicit la `float*`
- dacă nu se poate aloca
- se întâmplă foarte rar

- populăm tabloul cu reciprocele numerelor
- micșorăm zona
- `void*` este convertit implicit la `float*`
- dacă nu se poate realoca
- se întâmplă foarte rar

- elementele vechi se află tot la adresa *a*

- dezalocăm zona

Alocare dinamică tablouri 1D

- În cazul tablourilor unidimensionale alocarea dinamică sau alocarea statică (pe stivă) ambele produc un pointer la începutul zonei continue alocate
- Funcții pentru alocare prin return și prin parametru

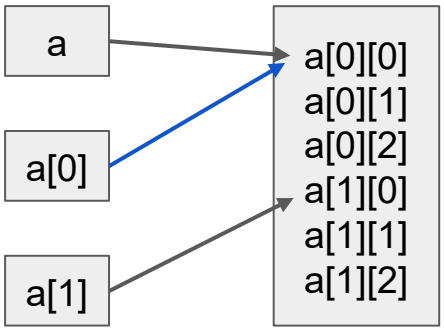
```
int* aloca_prin_return(int n){
    return malloc(n * sizeof(int));
}
void aloca_prin_parametru(int n, int** pp){
    *pp = malloc(n * sizeof(int));
}
```

Alocare dinamică tablouri 2D

- În cazul tablourilor bidimensionale alocarea dinamică și alocarea statică (pe stivă) produc rezultate diferite și incompatibile

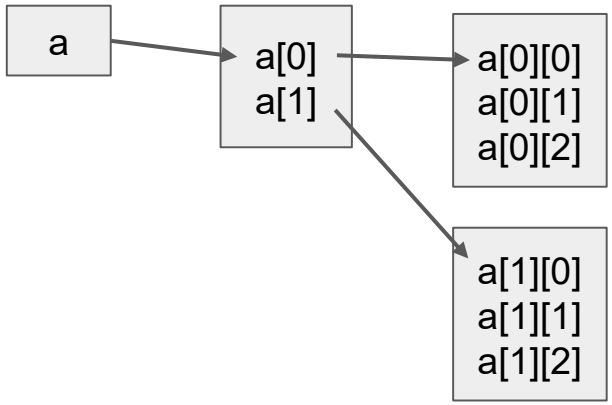
alocare statică (pe stivă)

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



alocare dinamică (pe heap)

```
int** a = malloc(2 * sizeof(int*));
a[0] = malloc(3 * sizeof(int));
a[1] = malloc(3 * sizeof(int));
```



Alocare dinamică tablouri 2D - șablon alocare

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

int main(){
    int n = 2, m = 3;
    char** a = calloc(n, sizeof(char*));
    for(int i=0; i<n; i++)
        a[i] = calloc(m, sizeof(char));

    for(int i=0; i<n; i++)
        for(int j=0; j<m; j++)
            a[i][j] = '#';
    *(*(a+1)+1) = '.';

    for(int i=0; i<n; i++)
        free(a[i]);
    free(a);
    return 0;
}
```

- șablon pentru tablou bidimensional cu *n* rânduri și *m* coloane
- se alocă *n* rânduri de `char*`
- se alocă fiecare rând de *m* `char`-uri
- accesarea se poate face atât cu [] cât și cu operatorul de dereferențiere *
- la dezalocare prima data dezalocăm rândurile
- apoi șirul pentru rânduri

Pointeri la funcții

- Definiție - este un pointer care arată la adresa corespunzătoare unei funcții
- Oferă o modalitate de a trimite funcții ca argumente la funcții
- Declarație

```
tip (*nume_f)(tip1, ...);
```

- unde `tip` este tipul returnat de funcție
- `nume_f` este numele pointerului la funcție
- `tip1` este tipul primului parametru al funcției
    - poate lipsi dacă funcția nu acceptă parametri
    - poate fi urmărit de mai mulți parametri
    - urmărește forma antetului funcției la care pointează

Pointeri la funcții - exemple de declarații

- Pointer la o funcție care nu returnează nimic și nu are nici un parametru de intrare

```
void (*f)();
```

- Pointer la o funcție care returnează un `int` și așteaptă două `int`-uri ca parametri

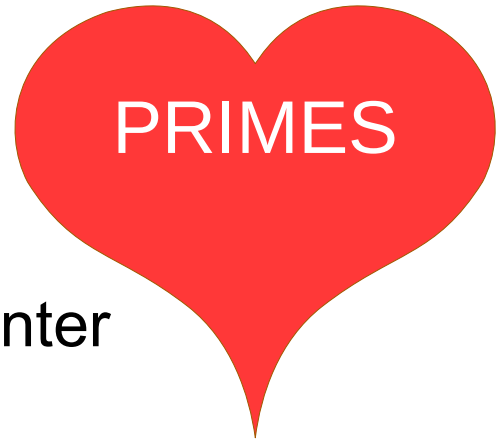
```
int (*g)(int, int);
```

- Pointer la o funcție care returnează un pointer `int` și așteaptă un pointer la un `int` (sau un tablou) și un `int`

```
int* (*h)(int*, int);
```

Pointeri la funcții

- Un pointer la o funcție
  - Conține adresa acelei funcții
  - Este similar cu numele unui tablou care reprezintă adresa primului element din tablou
  - Numele unei funcții este adresa de început a secțiunii de cod care definește funcția
- Pointerii la funcții pot fi
  - Asignați la alți pointeri la funcții
  - Trimiși ca argumente la apelul funcțiilor
  - Memorați în tablouri
- Apelul unei funcții prin intermediul unui pointer
  - Se poate face și fără a dereferenția pointer-ul
  - Se poate face și cu dereferențierea pointer-ului
    - De obicei pentru a sublinia faptul că se face apelul unei funcții prin intermediul unui pointer



Pointeri la funcții - exemplu simplu

```c
#include <stdio.h>
int add(int a, int b){ return a+b; }
int multi(int a, int b){ return a*b; }
int accum(int* a, int n, int (*f)(int, int)){
    int ret = a[0];
    for(int i=1; i<n; i++)
        ret = f(ret, a[i]);
    return ret;
}

int main(){
    int a[] = {1, 2, 4};
    int n = sizeof(a)/sizeof(a[0]);
    int s = accum(a, n, add);
    int p = accum(a, n, multi);
    printf("suma = %d\n", s);
    printf("produsul = %d\n", p);
    return 0;
}
```

- funcție simplă care adună
- funcție simplă care înmulțește
- funcție care acumulează elementele din șirul *a* aplicând operația din pointerul la funcție *f*
- se inițializează cu primul element
- se aplică funcția prin folosirea pointerului pe câte o pereche

- pentru sumă se trimite pointer la *add*
- pentru produs se trimite pointer la *multi*
- observăm că partea de cod pentru acumulare funcționează cu orice *f*, deci este generală

Pointeri la funcții - exemplu complex

```c
#include <stdio.h>
int comp_leq(int a, int b){ return a<=b; }
int comp_geq(int a, int b){ return a>=b; }
int comp_swap(int a, int b){ //91 <= 82
    int a2 = a%10 * 10 + a/10;
    int b2 = b%10 * 10 + b/10;
    return a2 <= b2;
}
void sortby(int* a, int n, int (*cmp)(int, int)){
    for(int i=0; i<n; i++){
        for(int j=i+1; j<n; j++){
            if (cmp(a[j], a[i])){
                int t = a[i];
                a[i] = a[j];
                a[j] = t;
            }
        }
    }
}//...
```

- returnează adevărat dacă a<=b
- returnează adevărat dacă a>=b
- returnează adevărat dacă după interschimbarea cifrelor a<=b
- intenționat pentru numere cu cel mult două cifre

- sortare prin selection sort pe baza unui comparator general
- mutăm cel mai mic element pe prima poziție, sortăm restul la fel
- timp pătratic în *n*

Pointeri la funcții - exemplu complex

```
//...
int main(){
    int a[] = {11, 91, 82, 13, 14, 7, 0};
    int n = sizeof(a)/sizeof(int);

    int (*comps[3])(int, int) = {
        comp_leq, comp_geq, comp_swap
    };

    for(int i=0; i<3; i++){
        sortby(a, n, comps[i]);
        for(int j=0; j<n; j++)
            printf("%02d ", a[j]);
        puts("");
    }

    return 0;
}
```

- un tablou cu pointeri la funcții
- fiecare are același antet

- sortăm tabloul folosind fiecare comparator
- apelăm funcția generală de sortare
- afișăm să vedem că ordinea respectă comparatorul

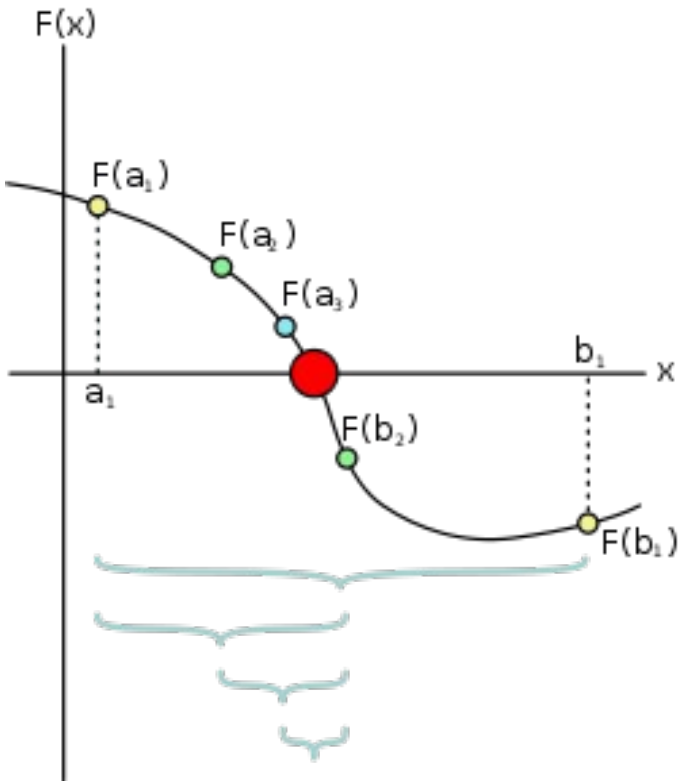
- pentru comp_swap avem ordinea:
- 00 11 91 82 13 14 07 pentru că după interschimbarea cifrelor ele sunt în ordine:
- 00 11 19 28 31 41 70

Analiză problemă - căutare rădăcină

- Se dă o ecuație matematică generală
  - polinom de ordin 3, funcții trigonometrice, exponențială
- Se cere găsirea unei rădăcini care se află într-un interval dat [a, b]
  - în acest interval se află exact o singură rădăcină
  - funcția este continuă
- Se cere răspunsul cu precizie de *x* cifre după virgulă
  - pe calculator oricum nu putem în general să reprezentăm soluția exactă

Analiză problemă - căutare rădăcină

- Metoda bisecției (căutare binară)
- precondiție f(a)*f(b) < 0
- eps = 10^{-x}
- Cât timp b-a > eps
  - c = (a+b)/2
  - dacă f(a) f(c) <= 0
    - b = c
  - altfel
    - a = c
- Rădăcina este aproximată ca a (sau b), b-a < eps



Analiză problemă - căutare rădăcină - implementare

```c
#include <stdio.h>
#include <math.h>
double f(double x){ return x*x*x - x - 2; }
double g(double x){ return exp(x)*x - 10; }
double bisection(double (*f)(double),
        double a, double b, int prec){
   double eps = pow(10, -prec);
   while(b-a > eps){
        double c = (a+b)/2;
        if (f(a)*f(c) <= 0)
            b = c;
        else
            a = c;
   }
   return a;
}
int main(){
   printf("%.8f\n", bisection(f, 1, 2, 7));
   printf("%.8f\n", bisection(g, 1, 2, 7));
   return 0;
}
```

- polinom de gradul 3
- funcție generală cu exponențială
- funcție care implementează metoda bisecției pe o funcție generală trimisă ca pointer

- când este produsul egal cu 0?

- putem apela cu orice funcție