Programarea Calculatoarelor

Cursul 7: Pointeri (II). Pointeri la pointeri. Alocarea dinamică a memoriei. Tablouri alocate dinamic. Pointeri la funcții

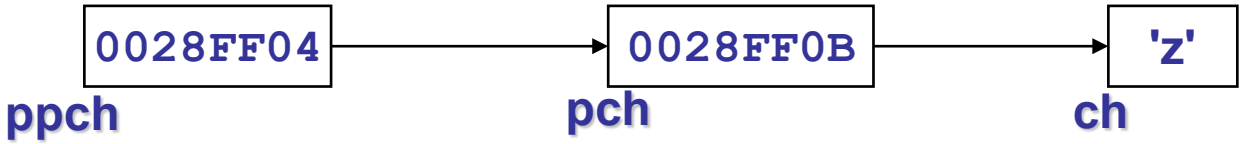
Ion Giosan

Universitatea Tehnică din Cluj-Napoca

Departamentul Calculatoare

Pointeri la pointeri

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

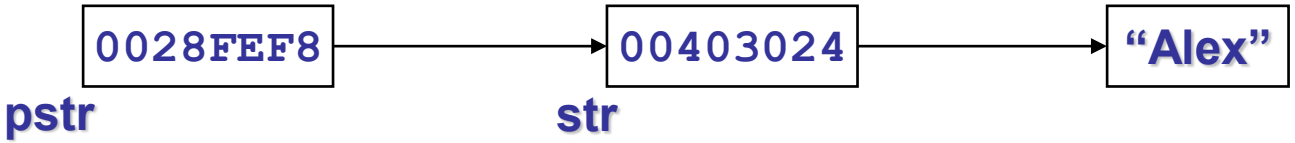



```
printf("%p %p %c",ppch,pch,ch); // 0028FF04 0028FF0B z
```

Pointer-ul de tip **`char *`** poate referi

- un singur caracter
- primul caracter dintr-un șir de caractere terminat prin caracterul `'\0'` (un *string*)

Pointer la *string*

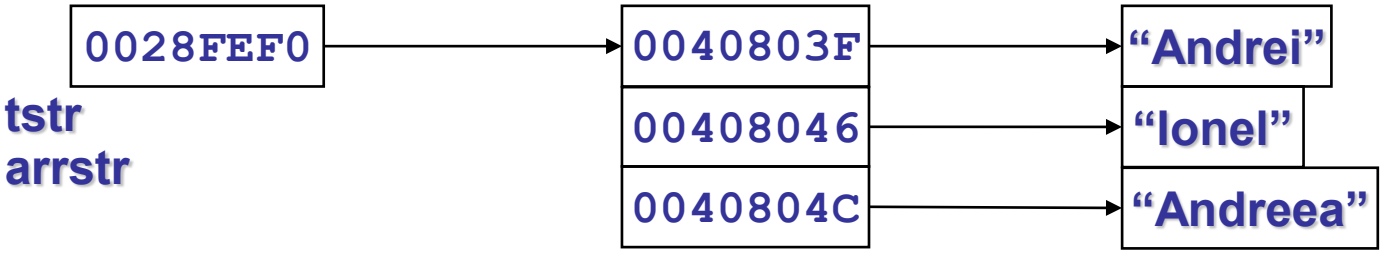



```
char *str="Alex";
char **pstr=&str;
printf("%p %p %s",pstr,str,str); // 0028FEF8 00403024 Alex
printf("%c%c",str[0],str[3]); // Ax
```

Pointer-ul de tip **`char **`** poate referi

- un singur *string*
- primul *string* dintr-un tablou de *string*-uri

Tablouri de *string-uri*




```
char * tstr[] = {"Andrei", "Ionel", "Andreea"};
char ** arrstr = tstr;
printf("%p %p %p %p",tstr,tstr[0],tstr[1],tstr[2]);
                    //0028FEF0 0040803F 00408046 0040804C
printf("%s %s",tstr[0],arrstr[2]); // Andrei Andreea
printf("%c%c%c",tstr[0][0],tstr[1][2],tstr[2][6]); // Ana
```

Pointeri utilizați în *string*-uri

- Un *string* nu este altceva decât o zonă de memorie ocupată cu un șir de caractere (un caracter pe un octet) terminată cu un octet de valoare zero (caracterul '\0')
- O variabilă care reprezintă un *string* nu este altceva decât un pointer la primul octet
- Un *string* nu poate conține o înșiruire de caractere '\0', întrucât un astfel de caracter reprezintă finalul unui *string*
- Nu se poate copia conținutul un *string* într-un alt *string* utilizând operatorul de atribuire **=**
  - Acesta copiază pointerul nu și conținutul!
  - Pentru a crea un *string* duplicat trebuie folosite funcții de procesare specifice
- Nu uitați de dimensiunea alocată pentru un *string* și nici că acesta trebuie să fie terminat cu un octet zero
  - Altfel se poate ca programul să acceseze caractere aflate în memorie în afara spațiului alocat *string*-ului

Pointeri utilizați în *string*-uri

```
char s1[] = {'I','m','i','n','e','n','t','\0'};
// echivalent cu:
char s1[] = "Iminent";

char * s2 = s1; // refera acelasi string
printf("%p %s %p %s",s1,s1,s2,s2);
    // 0028FF08 Iminent 0028FF08 Iminent
s2[0]='E';
printf("%s %s",s1,s2); // Eminent Eminent
s2[7]='a';
printf("%s", s1); // Eminenta.................
    /* afisarea caracterelor continua cu valori din
       memorie pana la intalnirea unui octet zero
       sau pana cand memoria poate fi accesata! */
```

Pointeri utilizați în *string*-uri

```
char s3[100] = "Iminent"; // aloca 100 de caractere!

char *s4 = s3; // refera acelasi string
s3[0]='E';
s3[7]='a';

s4[8]='\0'; // se scrie caracterul '\0'
// echivalent cu:
s4[8]=0; // se scrie octetul zero pe ultima pozitie

printf("%s",s3); // Eminenta
```

Pointeri - rezumat

- Un pointer reține o referință către o zonă de memorie
  - În acea zonă de memorie este stocat ceva folositor
- Operația de dereferențiere aplicată unui pointer permite accesul la zona de memorie referită
  - Dereferențierea este permisă numai după ce pointer-ului i-a fost asignată o referință validă
  - Pot apărea erori serioase dacă se dereferențiază un pointer căruia nu i-a fost asignată o referință validă
- Prin alocarea unui pointer nu i asignează automat o referință, acesta fiind de obicei neinițializat
- Atribuirea unui pointer la alt pointer face ca ambii să refere aceeași zonă de memorie
  - Are loc partajarea aceleiași informații prin intermediul lor

Alocarea/dezalocarea memoriei

- Variabilele **globale** și **statice** sunt alocate și trăiesc până la terminarea execuției programului
- Variabilele **locale** (**automate**) sunt alocate pe **stivă** și sunt distruse în momentul părăsirii funcției în care au fost alocate
- ***Heap***-ul este o zonă predefinită de memorie (de dimensiuni foarte mari) care poate fi accesată de program pentru a stoca date și variabile
- Datele și variabilele pot fi alocate pe *heap* prin apeluri speciale de funcții din biblioteca *stdlib.h*: **malloc**, **calloc**, **realloc**
- Zonele de memorie pot să fie dezalocate, la cerere, prin apelul funcției **free**
  - Este recomandat ca memoria să fie eliberată în momentul în care datele/variabilele respective nu mai sunt de interes!

Avantajele/dezavatajele *heap*-ului

- **Avantaje**
  - Durata de viață
    - Programatorul controlează exact momentele când are loc alocarea și dezalocarea memoriei
    - Este posibilă alocarea unei structuri de date în memorie și chiar returnarea adresei ei de către o funcție în locul unde aceasta este apelată
  - Dimensiuni
    - Dimensiunea memoriei alocate poate fi controlată în timpul execuției. De exemplu un *string* poate fi alocat astfel încât să aibă dimensiunea identică cu a altui *string* specific și cunoscut doar în timpul execuției programului
- **Dezavantaje**
  - Mai mult de lucru
    - Alocarea memoriei trebuie să fie făcută explicit în codul scris
  - Mai multe *bug*-uri
    - Neatenția la alocarea dimensiunilor zonelor respective de memorie
  - Memoria pe stivă este limitată dar întotdeauna este alocată corect

Funcții pentru alocarea/dezalocarea memoriei

- Cererea pentru alocarea unui bloc continuu de memorie pe *heap*:
- Prototipul funcției **malloc**

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

  - Funcția **malloc** returnează un pointer valid către blocul alocat pe *heap* sau pointer la NULL dacă cererea nu poate fi îndeplinită
  - Tipul **size_t** al parametrului formal este de fapt un tip **unsigned long**, iar **size** reprezintă dimensiunea blocului de memorie alocat, exprimată în octeți
  - Se încearcă astfel alocarea unui bloc de memorie continuu de **size** octeți
  - Tipul **void*** returnat de către funcție face obligatorie utilizarea unei conversii de tip atunci când respectivul pointer trebuie memorat într-un pointer de un tip obișnuit

Funcții pentru alocarea/dezalocarea memoriei

- Cererea pentru alocarea unui bloc continuu de memorie format din mai multe elemente, inițializate cu octeți de zero, pe *heap*:
- Prototipul funcției **calloc**

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

  - Funcția **calloc** returnează un pointer valid către blocul alocat pe *heap* sau NULL dacă cererea nu poate fi îndeplinită
  - Tipul **size_t** al parametrului formal este de fapt un tip **unsigned long**, iar **num** reprezintă numărul de elemente și **size** dimensiunea unui element exprimată în octeți
  - Se încearcă astfel alocarea unui bloc de memorie continuu de **num*size** octeți, toți inițializați cu zero
  - Tipul **void*** returnat de către funcție face obligatorie utilizarea unei conversii de tip atunci când respectivul pointer trebuie memorat într-un pointer de un tip obișnuit

Funcții pentru alocarea/dezalocarea memoriei

- Cererea pentru redimensionarea (creșterea/scăderea dimensiunii) unui bloc de memorie deja alocat
- Prototipul funcției **realloc**

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

  - Funcția **realloc** returnează un pointer valid către noul bloc re-alocat pe *heap* sau pointer la NULL dacă cererea nu poate fi îndeplinită
  - Parametrul formal **block** este un pointer de tip **void** către vechiul bloc de memorie existent
  - Tipul **size_t** al parametrului formal este de fapt un tip **unsigned long**, iar **size** reprezintă dimensiunea noului bloc de memorie ce trebuie re-alocat, exprimată în octeți
  - Se încearcă astfel re-alocarea blocului de memorie continuu având **size** octeți
  - Tipul **void*** returnat de către funcție face obligatorie utilizarea unei conversii de tip atunci când noul pointer trebuie memorat într-un pointer de un tip obișnuit

Funcții pentru alocarea/dezalocarea memoriei

- Dezalocarea memoriei alocate pe *heap* cu ajutorul funcțiilor **malloc**, **calloc, realloc**:
- Prototipul funcției **free**

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

  - Funcția **free** primește ca și argument un pointer de tip **void** către blocul de memorie valid, alocat pe *heap,* care a fost anterior alocat
  - Funcția dezalocă zona respectivă de memorie, marcând-o ca fiind disponibilă pentru a putea fi ulterior realocată (refolosită)
  - După apelul funcției **free** programul nu trebuie să mai acceseze nici măcar un octet din blocul care a fost dezalocat sau să prespună ca acesta mai este încă valid!
  - Un bloc de memorie nu trebuie eliberat de mai multe ori!

Exemplu: Alocarea/dezalocarea memoriei

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

void afiseaza(float *s, int nr)
{
    printf("Sirul de valori:\n");
    for (int i=0; i<nr; i++)
        printf("%g ",s[i]);
    printf("\n");
}

void citeste_elemente(float *s, int nr, int poz) {
    printf("Se vor citi %d valori:\n",nr);
    for (int i=0; i<nr; i++) {
        printf("Valoare[%d]=",poz+i);
        scanf("%f",s+poz+i);
    }
}
```

Exemplu: Alocarea/dezalocarea memoriei

```
int main() {
    int n;
    printf("Numarul de elemente al sirului: ");
    scanf("%d",&n);
    float *a = (float*)calloc(n, sizeof(float));
    if (a==NULL) {
        printf("Nu s-a putut aloca memorie!");
        exit(1);
    }
    citeste_elemente(a, n/2, 0);
    afiseaza(a,n);

    printf("Noul numar de elemente al sirului: ");
    int *r=(int*)malloc(sizeof(int));
    scanf("%d",r);
```

Exemplu: Alocarea/dezalocarea memoriei

```
    printf("Adresa blocului initial: %p\n",a);
    float *ra = (float*)realloc(a,(*r)*sizeof(float));
    if (ra==NULL) {
        printf("Nu s-a putut re-aloca memorie!");
        free(a);
        exit(2);
    }
    a=ra;
    printf("Adresa blocului realocat: %p\n",a);
    citeste_elemente(a, 1, *r-1);
    afiseaza(a,*r);

    free(a);
    free(r);
    return 0;
}
```

Exemplu: Alocarea/dezalocarea memoriei

Exemplu de execuție a programului:

```
Numarul de elemente al sirului: 6
Se vor citi 3 valori:
Valoare[0]=5.42
Valoare[1]=9.547
Valoare[2]=-1.41
Sirul de valori:
5.42 9.547 -1.41 0 0 0
Noul numar de elemente al sirului: 9
Adresa blocului initial: 00361560
Adresa blocului realocat: 00361560
Se vor citi 1 valori:
Valoare[8]=7.14
Sirul de valori:
5.42 9.547 -1.41 0 0 0 4.23518e-022 2.61062e-042 7.14
```

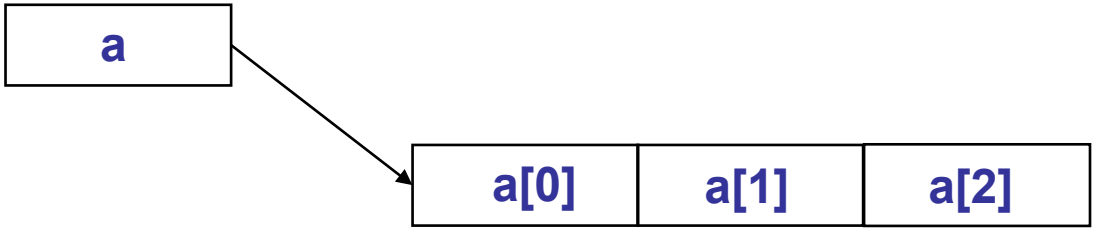
Alocarea dinamică a tablourilor unidimensionale

- Alocare pe stivă

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

- Alocare dinamică (pe heap)

```
int *a=(int*)malloc(3*sizeof(int));
```



În ambele situații **a** este pointer la primul element din șir (are valoarea adresei primului element din șir)

Alocarea dinamică a tablourilor unidimensionale - exemplu

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

void init(int n, int *x) {
    for (int i=0;i<n;i++)
        x[i]=i*i; // acces indexat la elementele tabloului
}
void afiseaza(int n, int *x) {
    for (int i=0;i<n;i++)
        printf("%d ", *(x+i)); // folosind operatii cu pointeri
    printf("\n");
}

int * aloca_prin_return(int n) {
    int *x = (int*)malloc(n*sizeof(int));
    return x; // returneaza adresa unui tablou alocat dinamic
}

void aloca_in_parametru(int n, int **x) {
    *x = (int*)malloc(n*sizeof(int));
} // aloca prin intermediul parametrului formal
```

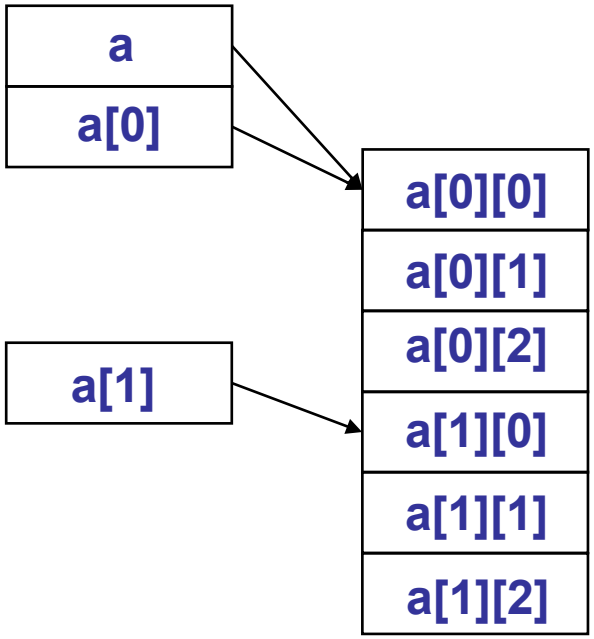
Alocarea dinamică a tablourilor unidimensionale - exemplu

```
int main() {
  int nr=6;
  int a[nr]; //tablou alocat pe stiva
  int *b = (int*)malloc(nr*sizeof(int)); // tablou alocat dinamic
  int *c = aloca_prin_return(nr); // tablou alocat dinamic
  int *d; //tablou alocat dinamic ulterior
  aloca_in_parametru(nr, &d);
  int * p[4]={a,b,c,d}; // sir de 4 pointeri la int (4 tablouri)
  printf("%d %d %d\n",sizeof(a),sizeof(b),sizeof(p));// 24 4 16
  for (int i=0; i<4; i++) {
      init(nr, p[i]);
      afiseaza(nr, p[i]); // 0 1 4 9 16 25
  }
  free(b); free(c);free(d);
  return 0;
}
```

Alocarea dinamică a tablourilor bidimensionale

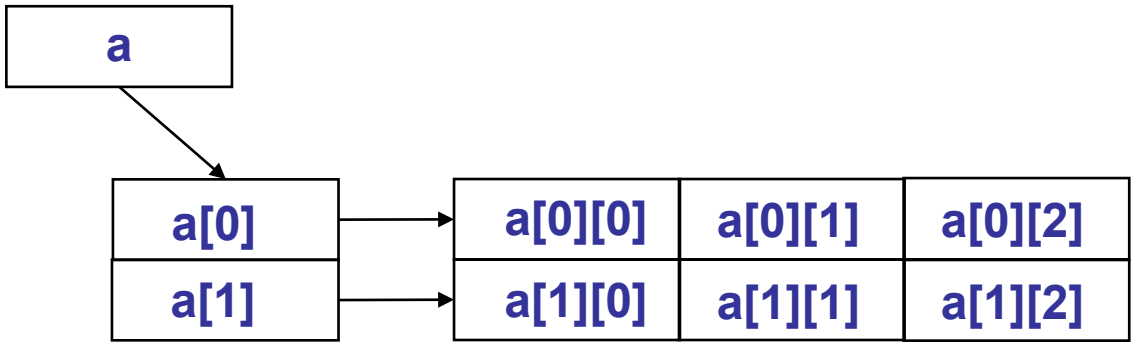
- Alocare pe stivă

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



- Alocare dinamică (pe heap)

```
int **a=(int**)malloc(2*sizeof(int*));
for (int i=0;i<2;i++)
   a[i]=(int*)malloc(3*sizeof(int));
```



Alocarea dinamică a tablourilor bidimensionale - exemplu

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

void init_tablou(int r, int c, int x[r][c]) {
    for (int i=0;i<r;i++)
        for (int j=0;j<c;j++)
             x[i][j]=i+j; // acces indexat la elemente
}// functia nu poate manipula un tablou alocat dinamic!

void afiseaza_tablou(int r, int c, int x[r][c]) {
    // afisare sub forma unei matrici
    for (int i=0;i<r;i++) {
        for (int j=0;j<c;j++)
            printf("%d ", x[i][j]); // acces indexat la elemente
        printf("\n");
    }
    printf("\n");
} // functia nu poate manipula un tablou alocat dinamic!
```

Alocarea dinamică a tablourilor bidimensionale - exemplu

```
void init(int r, int c, int **x)
{
    for (int i=0;i<r;i++)
        for (int j=0;j<c;j++)
            *(*(x+i)+j)=i+j; // acces cu operatii cu pointeri
} // functia nu poate manipula un tablou nealocat dinamic!


void afiseaza(int r, int c, int **x)
{
    // afisare sub forma unei matrici
    for (int i=0;i<r;i++) {
      for (int j=0;j<c;j++)
        printf("%d ", *(*(x+i)+j)); // acces cu operatii cu pointeri
      printf("\n");
    }
    printf("\n");
} // functia nu poate manipula un tablou nealocat dinamic!
```

Alocarea dinamică a tablourilor bidimensionale - exemplu

```
int ** aloca_prin_return(int r, int c) {
    int **x = (int**)malloc(r*sizeof(int*));
     for (int i=0;i<r;i++)
        x[i]=(int*)malloc(c*sizeof(int));
    return x; // returneaza adresa unui tablou alocat dinamic
}


void aloca_in_parametru(int r, int c, int ***x) {
    *x = (int**)malloc(r*sizeof(int*));
     for (int i=0;i<r;i++)
        (*x)[i]=(int*)malloc(c*sizeof(int));
} // aloca prin intermediul parametrului formal


void dezaloca(int r, int **x)
{
    for (int i=0;i<r;i++)
        free(x[i]); // dezaloca fiecare linie
    free(x); // dezaloca sirul de pointeri la linii
}
```

Alocarea dinamică a tablourilor bidimensionale - exemplu

```
int main() {
    int m=3; int n=2;
    int a[m][n]; //tablou alocat pe stiva; matrice cu m linii, n coloane
    int **b = (int**)malloc(m*sizeof(int*));//tablou alocat dinamic, mai
                                            // intai sirul de pointeri la linii
    for (int i=0;i<m;i++)
        b[i]=(int*)malloc(n*sizeof(int)); // alocarea fiecarei linii
    int **c = aloca_prin_return(m, n); // tablou alocat dinamic
    int **d; // tablou alocat dinamic ulterior
    aloca_in_parametru(m, n, &d);
    init_tablou(m, n, a);
    afiseaza_tablou(m, n, a);
    int ** p[3]={b,c,d}; // sir de 3 matrici alocate dinamic
    printf("%d %d %d\n\n", sizeof(a), sizeof(b), sizeof(p)); // 24 4 12
    for (int i=0; i<3; i++) {
        init(m, n, p[i]);                    // 0 1
        afiseaza(m, n, p[i]);                // 1 2
    }                                        // 2 3
    dezaloca(m, b); dezaloca(m, c); dezaloca(m, d);
    return 0;
}
```

Pointeri la funcții

- Declararea unui pointer la o funcție

```
tip_returnat (*nume_functie)();
```

- O astfel de funcție trebuie apelată cu atenție deoarece limbajul C nu verifică dacă s-au trimis argumente corespunzătoare!
- Exemple

```
int (*f1)(double); /* Pointer la o functie  care
                      primeste un double si
                      returneaza un int */
void (*f2)(char*); /* Pointer la o functie  care
                      primeste un pointer la char si
                      nu returneaza nimic */
double* (*f3)(int, int); /* Pointer la o functie
                      care primeste doi parametri de
                      tip int si returneaza
                      un pointer la double */
```

Pointeri la funcții

- Confuzia pointerilor la funcții cu funcții care returnează pointeri

```
int *f4(); /* Functie care returneaza pointer
              la int */
int (*f5)(); /* Pointer la functie care
                returneaza int */
int* (*f6)(); /* Pointer la functie care
                 returneaza pointer la int */
```

- Spațiul alb poate fi rearanjat. Următoarele două declarații sunt echivalente

```
int *f4();
int* f4();
```

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

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

double diferenta(int x, int y){
    return x-y;
}
double media(int x, int y){
    return (x+y)/2.0;
}
int main()
{
    double (*pf)(int,int); //Pointer la o functie
    double (*tpf[2])(int,int); //Tablou de pointeri la functii
    pf=diferenta; tpf[0]=pf;
    printf("%p\n",tpf); //0028ff04
    printf("%p %g\n", pf, tpf[0](17,8)); //0040135D 9
    pf=media; tpf[1]=pf;
    printf("%p %g\n", pf, tpf[1](17,8)); //00401377 12.5
    return 0;
}
```

Pointeri la funcții ca parametri la alte funcții

- O funcție `f` definită astfel

```
tip_f f(lista_parametri_formali_f)
```

poate fi trimisă la apelul unei funcții `g` definită astfel

```
tip_g g(...,
        tip_f (*p)(lista_parametri_formali_f),
        ...)
```

prin apelul

```
g(..., f, ...);
```

- Observație: numele unei funcții reprezintă un pointer la acea funcție

Pointeri la funcții ca parametri la alte funcții - exemplu

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

double diferenta(int x, int y){
    return x-y;
}
double media(int x, int y){
    return (x+y)/2.0;
}
double calcul(int a, int b, double (*f)(int,int))
{
    return f(a,b);
}
int main()
{
    double x = calcul(10,1,media);
    double y = calcul(10,1,diferenta);
    printf("%g %g", x, y); // 5.5 9
    return 0;
}
```