

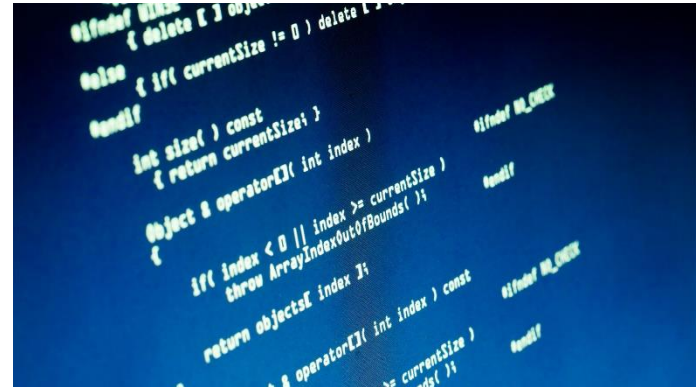
Tehnici de programare - TP



Cursul 2 – Pointeri la funcții

Ș.l. dr. ing. Cătălin Iapă

catalin.iapa@cs.upt.ro



Matrici alocate dinamic

Pointeri. Funcții

Pointeri la funcții

Algoritmi generici

qsort

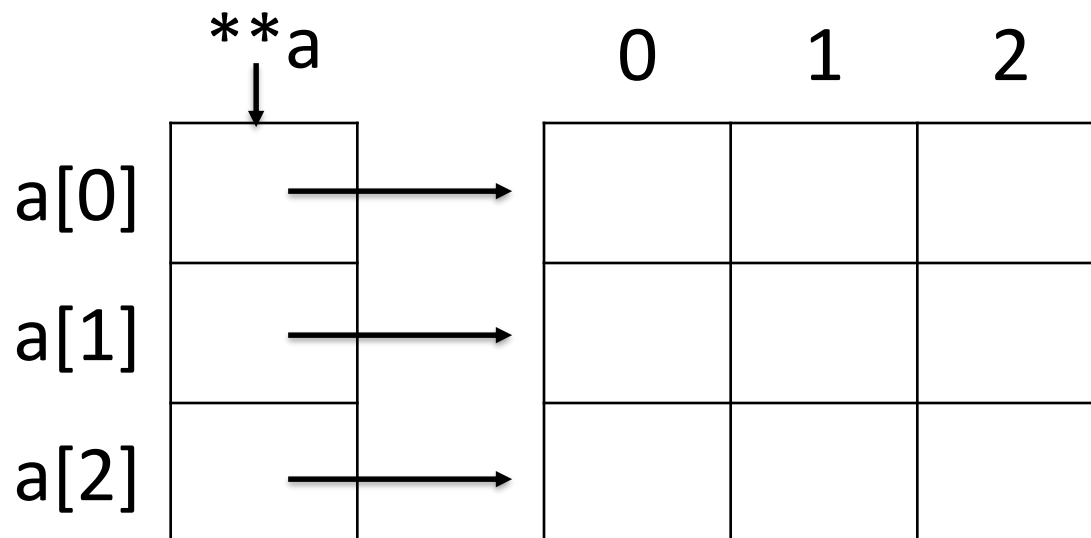
bsearch

Să ne amintim:

Crearea unei matrici alocate dinamic

Matricea va fi definită ca `int **a`

`a[i]` este pointerul (`int*`) de la indexul `i` din `a`, adică adresa memoriei care conține acea linie



Să ne amintim:

Crearea unei matrici alocate dinamic

```
int m,n,i,j;
```

```
int **a;
```

```
printf("m=");scanf("%d",&m);
```

```
printf("n=");scanf("%d",&n);
```

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

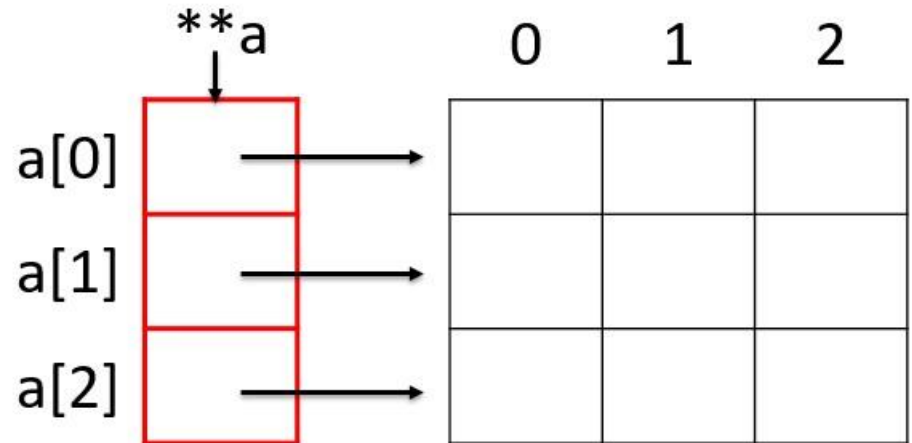
```
if(a==NULL)
```

```
{
```

```
    printf("memorie insuficienta\n");
```

```
    exit(EXIT_FAILURE);
```

```
}
```



Să ne amintim:

Crearea unei matrici alocate dinamic

// alocare linii din matrice

```
for(i=0;i<m;i++)  
{
```

```
a[i]=(int*)malloc(n*sizeof(int));  
if(a[i]==NULL)
```

```
{
```

```
for(i--;i>=0;i--)
```

```
free(a[i]);
```

// elibereaza liniile alocate anterior

```
free(a);
```

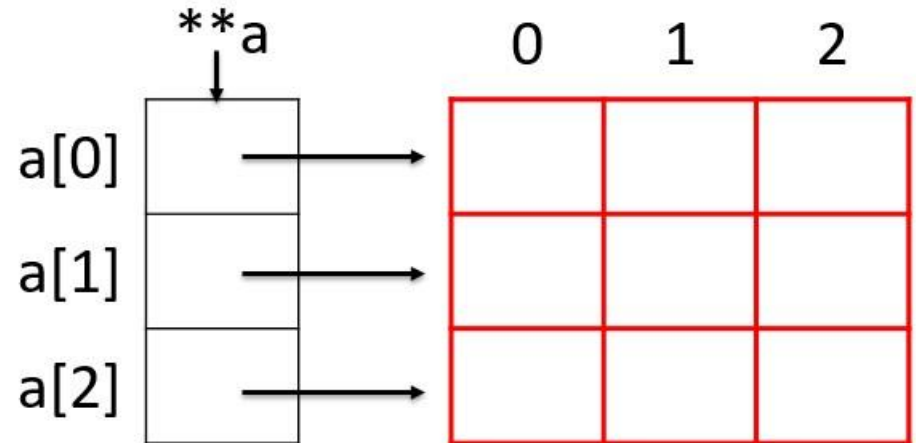
// elibereaza vectorul de pointeri

```
printf("memorie insuficienta\n");
```

```
exit(EXIT_FAILURE);
```

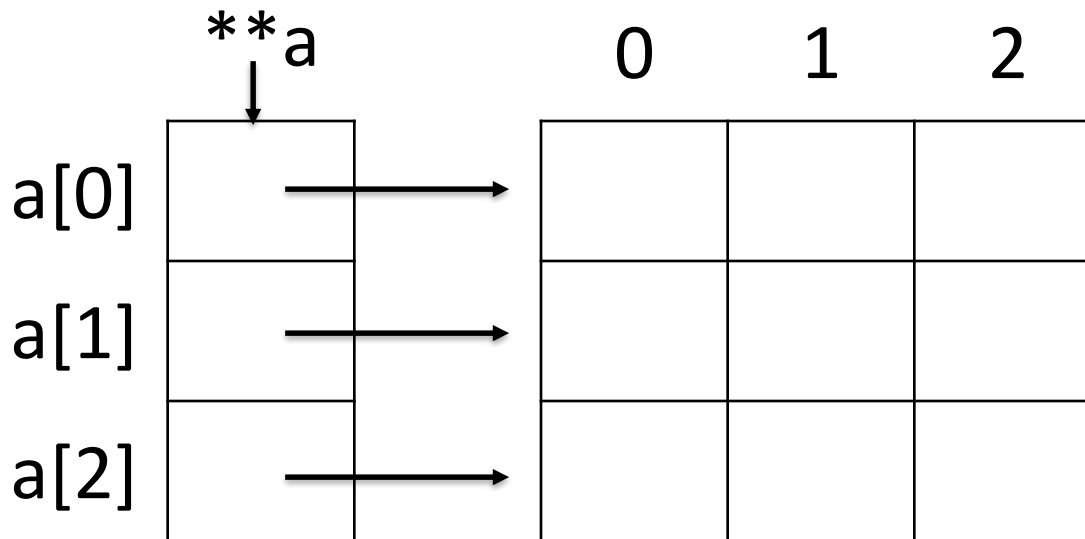
```
}
```

```
}
```



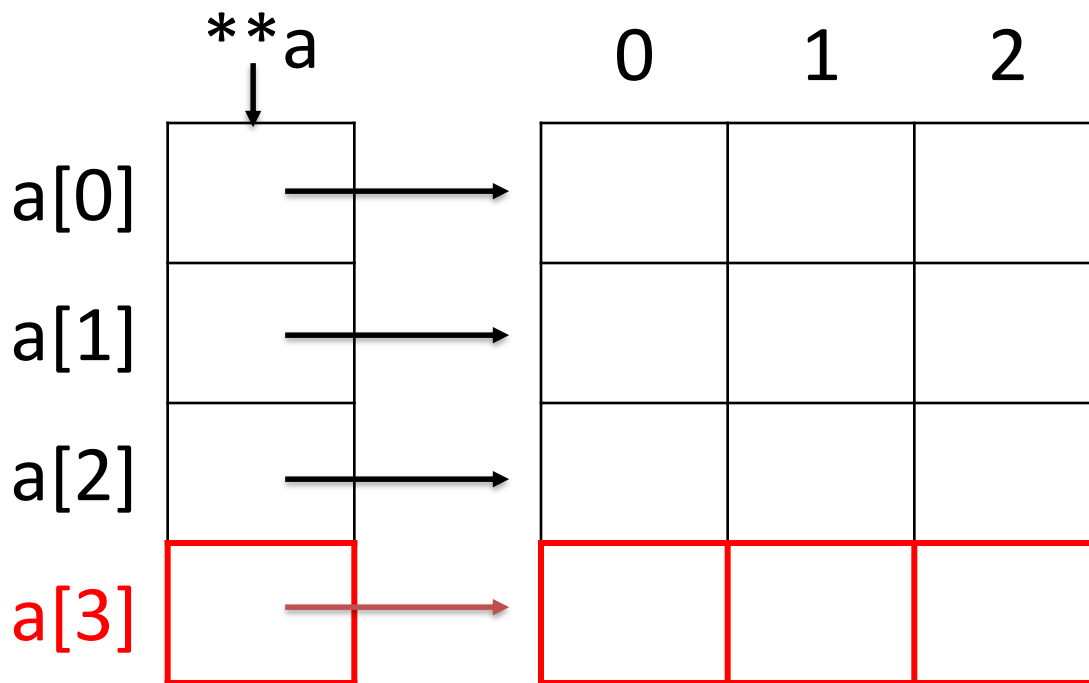
Modificarea dimensiunilor unei matrici alocate dinamic

Avantajul folosirii matricilor alocate dinamic (și a vectorilor alocați dinamic, în general) este că **putem redimensiona mărimea acestora**.



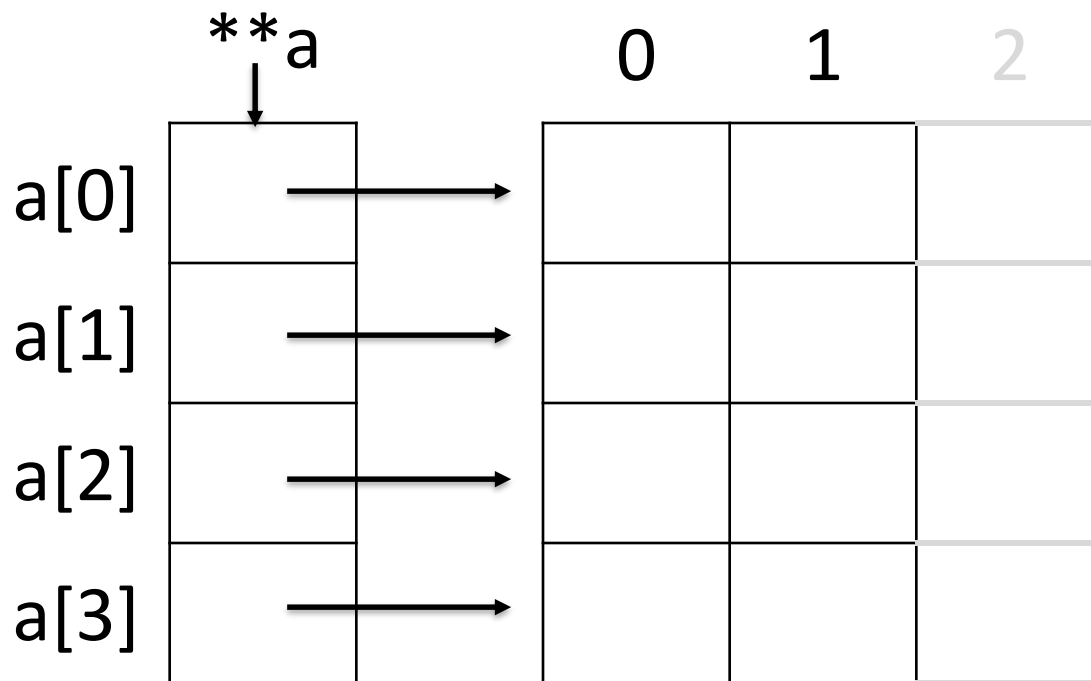
Modificarea dimensiunilor unei matrici alocate dinamic

Dacă vrem să adăugăm **o linie în plus** matricii va trebui să **redimensionăm vectorul de pointeri**, iar apoi **să alocăm dinamic memorie** pentru linia suplimentară.



Modificarea dimensiunilor unei matrici alocate dinamic

Dacă vrem să ștergem o coloană din matrice va trebui să redimensionăm fiecare linie din matrice (fiecare vector alocat dinamic).



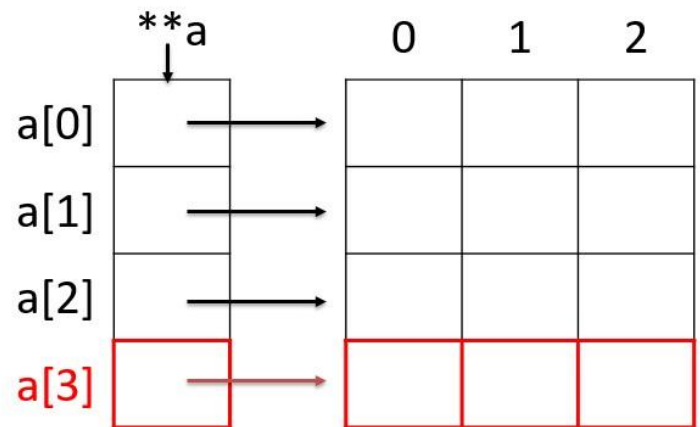
Modificarea dimensiunilor unei matrici alocate dinamic

Adăugarea unei linii suplimentare în matrice:

```
int ** adaugare_linie(int **a, int m, int n)
{
    int ** aux;
    aux = (int **) realloc(a,(m+1)*sizeof(int*));
    if(aux==NULL)
        free_mem(a,m,n);
    a=aux;

    a[m]= (int*)malloc(n*sizeof(int));
    if(a[m]==NULL)
        free_mem(a,m,n);

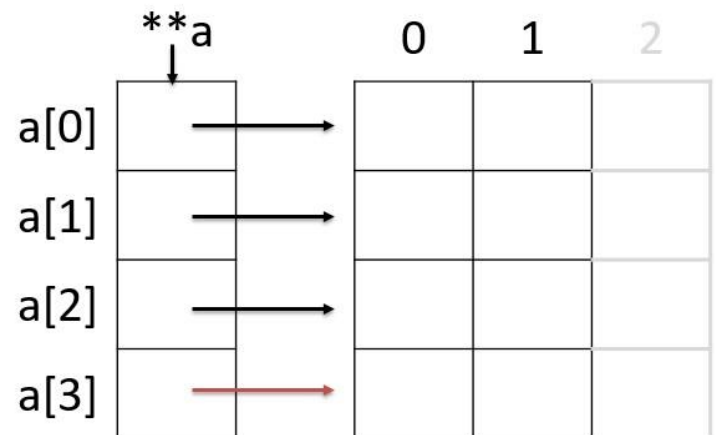
    return a;
}
```

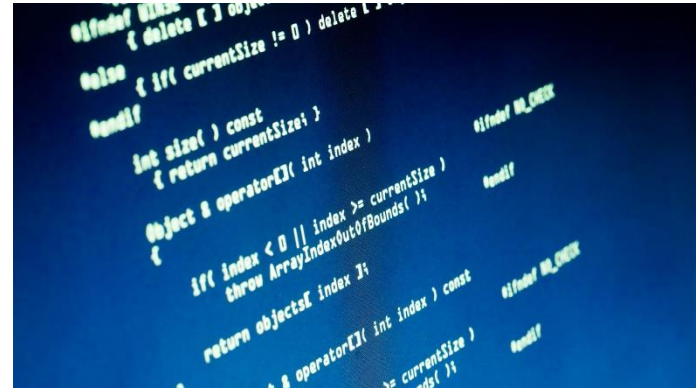


Modificarea dimensiunilor unei matrici alocate dinamic

Ștergerea unei coloane din matrice:

```
void stergere_coloana(int **a, int m, int n)
{
    int *aux;
    for(int i=0;i<m;i++)
    {
        aux=(int*)realloc(a[i],(n-1)*sizeof(int));
        if(aux==NULL)
            free_mem(a,m,n);
        a[i]=aux;
    }
}
```





Matrici alocate dinamic

Pointeri. Funcții

Pointeri la funcții

Algoritmi generici

qsort

bsearch

Funcții

Funcțiile au rolul de a grupa o secțiune de cod care îndeplinește o anumită sarcină sau funcționalitate.

Ele permit scrierea de cod modular, care **poate fi apelat de mai multe ori** dintr-un program.

Scopul principal al funcțiilor este de a face programarea mai **ușoară**, mai **organizată** și mai **ușor de întreținut**.

Funcții cu pointeri

Funcții cu argumente de tip pointer

Problema: în limbajul C argumentele unei funcții de transmit doar prin valoare

– se copiază valoarea argumentelor de la apel

→ consecință: un parametru local modificat în corpul unei funcții nu modifică și valoarea variabilei cu care a fost apelată

Problema: Se se scrie o funcție care interschimbă valorile a 2 argumente (swap)

```
#include <stdio.h>

void swap(int x, int y)
{
    int aux;
    aux = x;
    x = y;
    y = aux;
}

int main(void)
{
    int a = 3;
    int b = 5;
    printf ("%d %d\n", a, b);
    swap(a,b);
    printf ("%d %d\n", a, b);
    return 0;
}
```

```
valy@staff:~/teaching$ ./p
3 5
3 5
valy@staff:~/teaching$
```

- la apelul swap se copiază valorile lui a și b pe stivă iar parametri locali vor primi aceste valori
- intern, în funcție valorile parametrilor x și y se modifică dar în afara funcției a și b nu se modifică – modificarea nu se vede și în exteriorul funcției – este normal și corect fiind doar o copie de valori

Funcții cu pointeri

Funcții cu argumente de tip pointer

Problema: Se se scrie o funcție care interschimbă valorile a 2 argumente (swap)

SOLUȚIE: folosirea în funcție a argumentelor de tip pointer și trimiterea adreselor în apelul swap

```
#include <stdio.h>

void swap(int *x, int *y)
{
    int aux;
    aux = *x;
    *x = *y;
    *y = aux;
}

int main(void)
{
    int a = 3;
    int b = 5;
    printf ("%d %d\n", a, b);
    swap(&a,&b);
    printf ("%d %d\n", a, b);
    return 0;
}
```

```
valy@staff:~/teaching$ ./p
3 5
5 3
valy@staff:~/teaching$
```

- la apelul swap se copiază valorile adreselor lui a și b pe stivă iar parametri locali vor primi aceste valori, ce reprezintă adresele variabilelor a și b
- În continuare adresele lor, valorile pointerilor (ce sunt adrese) nu pot fi modificate, fiind situația precedentă
- pe de altă parte, având acces la adresă se poate modifica conținutul de la acea adresă folosind pointeri cu dereferențiere

Pointeri

În C există următoarele **categorii de pointeri**:

- **pointeri de date**, care conțin adresa unei variabile
- **pointeri generici**, pointeri void *, pot conține adresa unui obiect oarecare, de orice tip.
 - se utilizează atunci când nu se cunoaște precis tipul entității care va fi instanțiată în mod dinamic la execuție
- **pointeri de funcții**, care punctează adresa codului executabil al unei funcții.

Zone de memorie ale unui program C

segmentul de cod (.text)

- conține codul executabil al programului precum și constantele
- are dimensiune fixă egală cu dimensiunea codului executabil
- este read-only (protejat la scriere) din punct de vedere al programului (programul nu își poate singur rescrie codul)

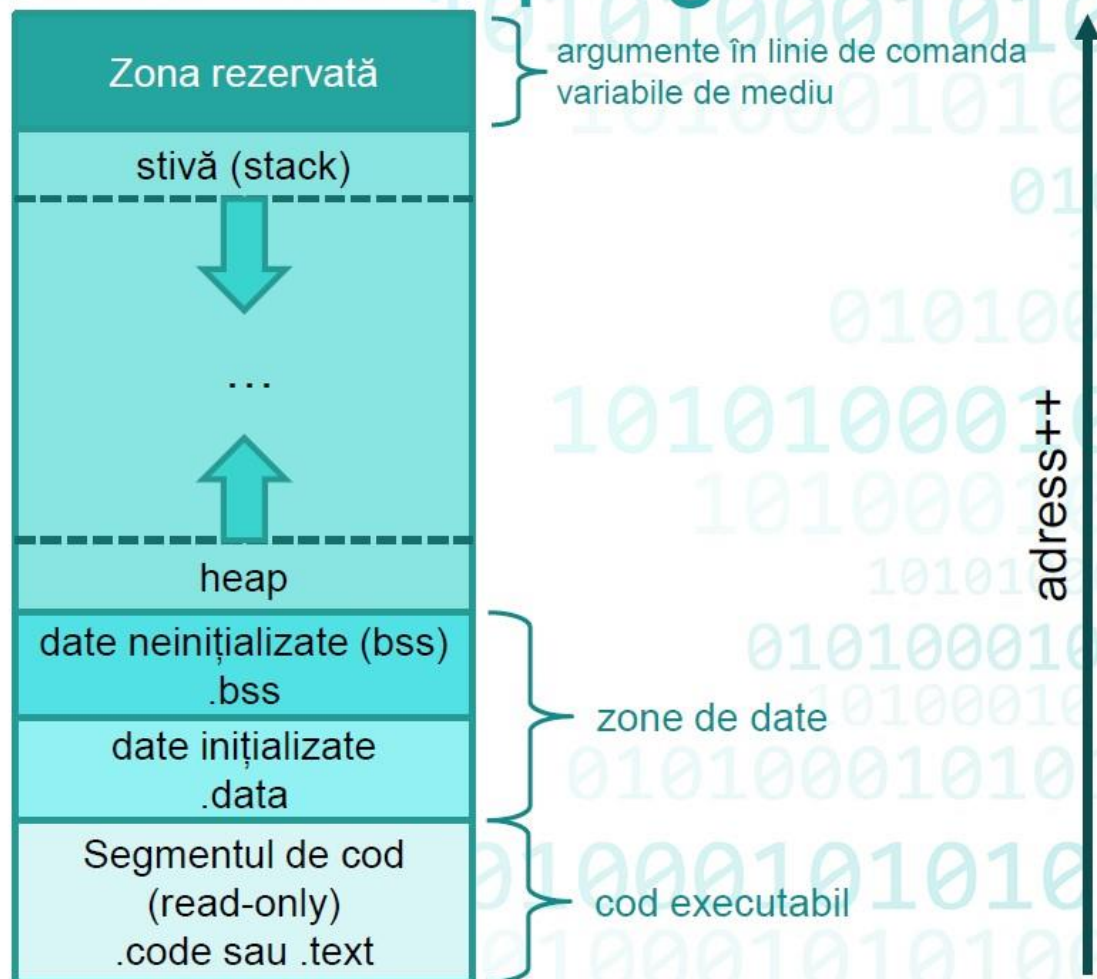
segmentul de date (.data și .bss)

segmentul .data

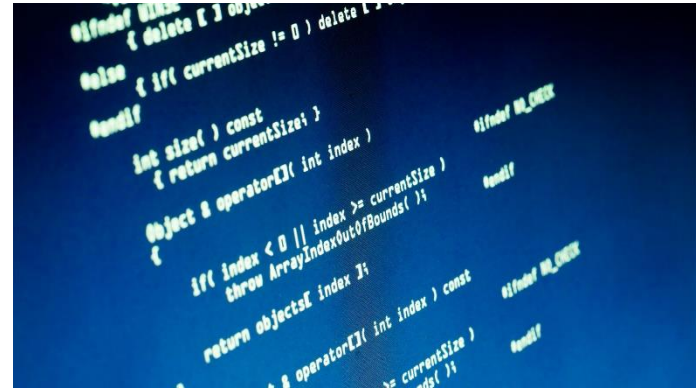
- conține date statice, declarate global sau cu modificatorul static în funcție, ce sunt inițializate de programator în cod
- este în politică read-write dar structura ei nu poate fi modificată în timpul execuției programului (la runtime)

segmentul .bss (block started by symbol)

- conține date statice, declarate global sau cu modificatorul static în funcție, ce NU sunt inițializate de programator în cod
- această zonă este inițializată de sistemul de operare cu 0 chiar înainte de lansarea în execuție a programului → dar NU E O REGULĂ
- este în politică read-write dar structura ei nu poate fi modificată la runtime



• Schema adaptare după: Cristina Stângaciu, curs "Limbaie de Programare", ETC, 2020



Matrici alocate dinamic

Pointeri. Funcții

Pointeri la funcții

Algoritmi generici

qsort

bsearch

Pointeri la funcții

În C, o funcție nu este o variabilă, dar este posibil să se definească **pointeri la funcții care se comportă similar unor variabile**: pot fi *atribuiți*, *plasați în tablouri*, *transmiși ca argumente* altor funcții, *returnați* de funcții etc.

Pointeri la funcții

Pointerii la funcții sunt variabile care stochează adresele funcțiilor în C.

În C, funcțiile sunt de fapt **adrese de memorie** care indică locația începutului funcției în memorie.

Pointerii la funcții sunt variabile care pot stoca aceste adrese și **pot fi utilizate pentru a apela funcții** sau pentru **a le trimite ca argumente către alte funcții**.

Pointeri la funcții

Cu pointerii la funcții **nu vom putea alocă sau elibera** spațiul de memorie, cum o putem face cu un pointer obișnuit către date.

Ca și la pointeri obișnuiți către date, putem avea și **vectori de pointeri la funcții**.

Dacă se implementează un **menu în program**, se pot folosi pointeri la funcții în loc de switch().

Pointeri la funcții

Pointerii la funcții **se declară** astfel:

*tip_bază (*ptr_fn)(argumente_funcție);*

Exemple de utilizare:

*int (*p)(int, int);*
*void (*pt)(int**,int, int);*
*double (*p3)(double);*

Pointeri la funcții

Declarație obișnuită de **funcție**:

tip_rez f(.....);

Declarație de **pointer la funcție**:

*tip_rez (*pf)(.....);*

Atribuiri echivalente: $pf = f;$ $pf = \&f;$

Apeluri echivalente: $f(...);$ $pf(...);$ $(*pf)(...);$

Pointeri la funcții

Cand declarăm un pointer la o funcție e important să punem între **paranteze rotunde** numele pointerului împreună cu *

```
int (*f)(void);  
//pointer la o funcție ce returnează un int
```

Dacă nu se pun paranteze, compilatorul va interpreta expresia ca **o funcție care returnează un pointer la int**:

```
int *f(void);  
//funcție care returnează un pointer către int
```

Pointeri la functii – exemplu de utilizare (1)

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

```
int f1(int a,int b)
{ return a+b; }
```

```
int f2(int a,int b)
{ return a-b; }
```

```
int main(void)
{ int a=7,b=5;
  int (*pf)(int,int);
  pf=&f1;
  printf("op(%d,%d)=>%d\n",a,b,(*pf)(a,b));
  pf=&f2;
  printf("op(%d,%d)=>%d\n",a,b,(*pf)(a,b));
  return 0;
}
```

```
op(7,5)=>12
op(7,5)=>2
```

// pf - pointer la o functie

// se seteaza pf cu adresa functiei f1

// se apelează functia pointata

Pointeri la funcții – exemplu de utilizare (2)

Pointer la funcție

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

```
int main()
```

```
{
```

```
void (*pointer_la_functie)(const char*) = &printf;
```

```
(*pointer_la_functie)("Pointer la functie\n");
```

```
return 0;
```

```
}
```



Matrici alocate dinamic

Pointeri. Funcții

Pointeri la funcții

Algoritmi generici

qsort

bsearch

Algoritmi generici

Un exemplu pentru utilizarea pointerilor la funcții ar putea fi **în programarea în jocuri video**.

În multe jocuri, este necesar să se definească comportamente specifice pentru personaje, obiecte sau evenimente în joc.

În loc să se scrie funcții separate pentru fiecare comportament posibil, se poate utiliza o funcție generală care acceptă **un pointer la o funcție de comportament specific**.

Astfel, programatorii pot crea comportamente personalizate și le pot utiliza în joc, permițând o mare flexibilitate în designul jocului.

Algoritmi generici

Una dintre aplicațiile pointerilor la funcții o constituie implementarea *algoritmilor generici*.

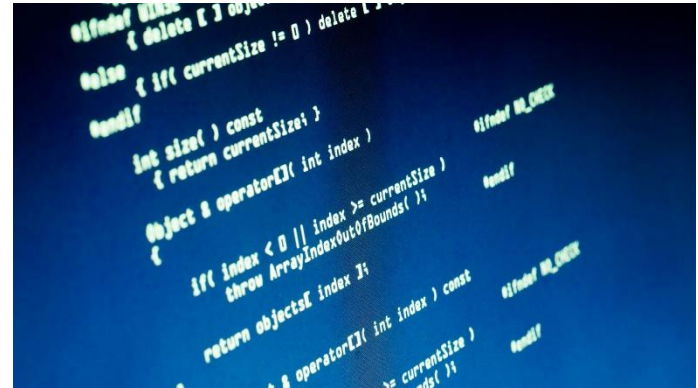
De exemplu, testarea dacă toate elementele unui vector îndeplinesc o anumită condiție, presupune iterarea vectorului și testarea fiecărui element conform condiției date.

Dacă dorim să testăm diverse condiții, *partea de iterare rămâne identică* și se modifică doar condiția de testat.

Algoritmi generici

Pentru algoritmi simpli nu este neapărat o îmbunătățire folosirea pointerilor la funcții dar, dacă algoritmul este foarte complex, conținând sute sau chiar mii de linii de cod, atunci **implementarea unei variante generice** a sa, care permite refolosirea unei mari părți din algoritm, reprezintă o **îmbunătățire de performanță**.

În biblioteca standard C există doi algoritmi generici, ***qsort* (quick sort)** și ***bsearch* (binary search)**, ambii declarați în antetul ***stdlib.h***.



Matrici alocate dinamic

Pointeri. Funcții

Pointeri la funcții

Algoritmi generici

qsort

bsearch

qsort()

Funcția *qsort* este o implementare performantă a algoritmului *quick sort* și poate fi folosită pentru *sortarea oricăror tipuri de vectori*.

qsort este declarat astfel:

```
void qsort(void *vector, size_t nElem, size_t dimensiuneElem,  
int (*compar)(const void *pElem1, const void *pElem2));
```

qsort are următorii parametri:

- *vector* - un vector de elemente
- *nElemente* - numărul elementelor din vector
- *dimensiuneElement* - dimensiunea unui element din vector, exprimată în octeți
- *compar* - o funcție care primește pointeri la două elemente din vector (transfer prin adresă).

qsort()

- *Funcția compar* din *qsort*- o funcție care primește pointeri la două elemente din vector (transfer prin adresă).

Parametrii sunt de tipul “*const void **”, adică pointeri generici la valori constante.

Funcția *compar* va fi apelată de *qsort* cu perechi de elemente, primul element fiind în vector în stânga celui de al doilea.

compar trebuie să compare elementele și să returneze un întreg cu următoarele semnificații:

- <0 - ordinea elementelor este corectă, deci vor fi lăsate de *qsort* la pozițiile lor prezente
- 0 - elementele sunt egale, deci ordinea nu contează
- >0 - ordinea este incorectă, deci elementele trebuie inversate

qsort() – exemplu

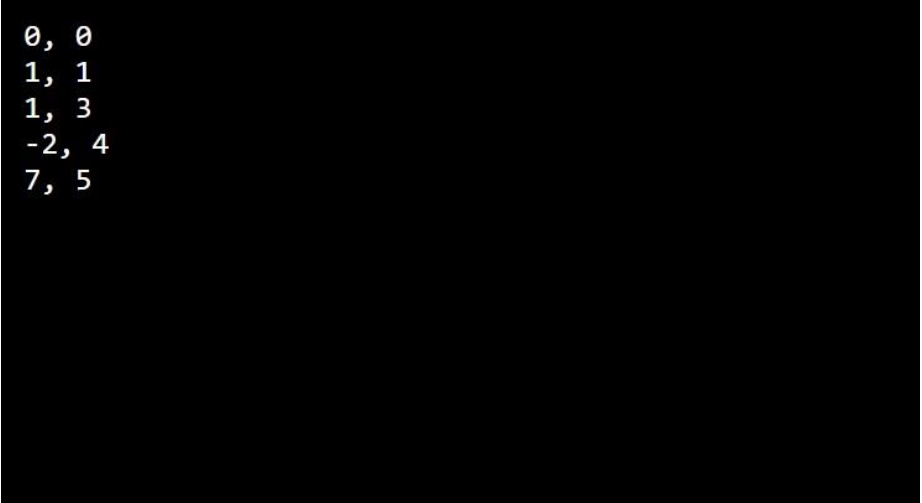
Se dă un vector de puncte în plan, având coordonatele (x,y) de tipul *double*. Se cere să se sorteze acest vector în ordinea distanțelor punctelor față de origine.

qsort() – exemplu

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

```
typedef struct{
    double x,y;
}Pt;
```

```
int main(void)
{
    Pt puncte[5]={{1,3},{7,5},{0,0},{-2,4},{1,1}};
    int i,n=5;
    qsort(puncte,n,sizeof(Pt),cmpDist);
    for(i=0;i<n;i++){
        printf("%g, %g\n",puncte[i].x,puncte[i].y);
    }
    return 0;
}
```



```
0, 0
1, 1
1, 3
-2, 4
7, 5
```

qsort() – exemplu

```
double dist(const Pt *pt) // distanta fata de origine
{ return sqrt(pt->x*pt->x+pt->y*pt->y); }
```

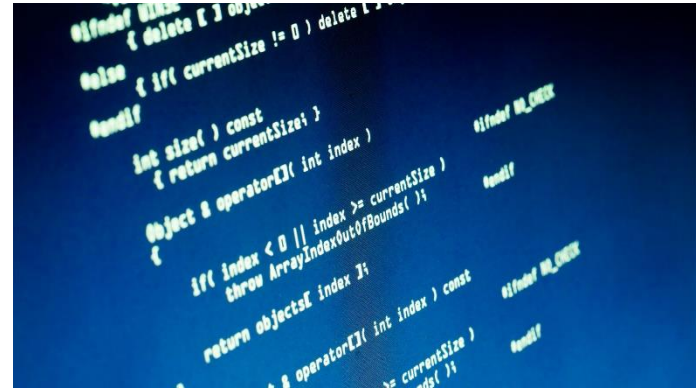
// deoarece qsort transmite functiei de comparare adresele elementelor, functia va primi pointeri la elemente

// in acest caz, deoarece vectorul are elemente de tipul Pt, functia va primi parametri de tipul Pt*

```
int cmpDist(const void *elem1,const void *elem2)
```

```
{
    const Pt *p1=(const Pt*)elem1;
    const Pt *p2=(const Pt*)elem2;
    double d1=dist(p1);
    double d2=dist(p2);
    if(d1<d2) return -1;
    if(d1>d2) return 1;
    return 0;
}
```

```
0, 0
1, 1
1, 3
-2, 4
7, 5
```



Matrici alocate dinamic

Pointeri. Funcții

Pointeri la funcții

Algoritmi generici

qsort

bsearch

bsearch()

Funcția **bsearch()** este definită în biblioteca standard C și este utilizată pentru căutarea unei chei într-un array sortat.

```
void *bsearch(const void *key, const void *base, size_t nmemb,  
              size_t size, int (*compar)(const void *, const void *));
```

Rezultatul returnat de funcția `bsearch()` este un pointer la elementul din array care corespunde cheii căutate sau NULL în cazul în care cheia nu a fost găsită.

bsearch()

Parametrii bsearch():

- **key**: pointer la cheia căutată. Acesta poate fi un pointer la orice tip
- **base**: pointer la începutul array-ului sortat în care se face căutarea.
- **nmemb**: numărul de elemente din array.
- **size**: dimensiunea fiecărui element din array.
- **compar**: pointer la o funcție care compară două elemente din array.

Această funcție trebuie să primească două argumente de tip `const void *` și să returneze un întreg negativ, zero sau pozitiv, în funcție de rezultatul comparației, la fel ca și la `qsort()`.

bsearch() – exemplu utilizare

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

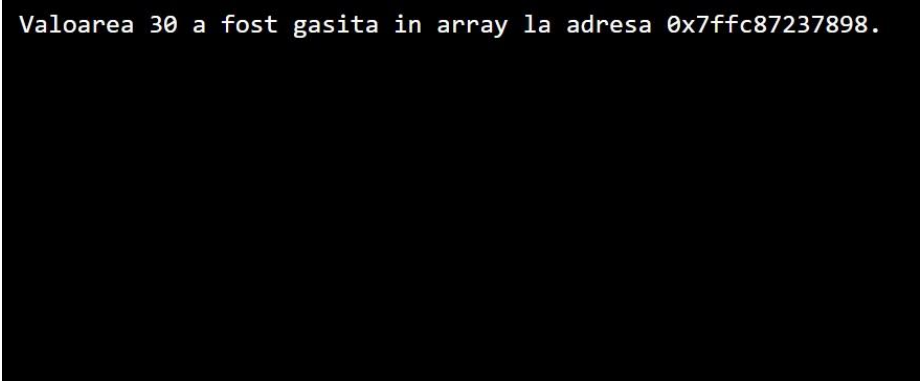
```
int cmpfunc(const void *a, const void *b) {
    return (*(int*)a - *(int*)b);
}
```

```
int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int n = 5;
    int key = 30;
```

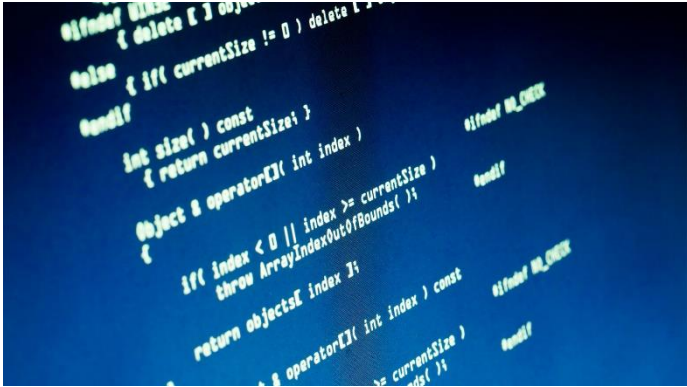
```
// Căutarea valorii 30 în array
```

```
int *ptr = (int*) bsearch(&key, arr, n, sizeof(int), cmpfunc);
```

```
if (ptr != NULL) {
    printf("Valoarea %d a fost gasita in array la adresa %p.\n", key, ptr);
} else {
    printf("Valoarea %d nu a fost gasita in array.\n", key);
}
return 0;
}
```



Valoarea 30 a fost gasita in array la adresa 0x7ffc87237898.



Vă mulțumesc!