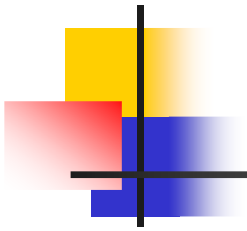
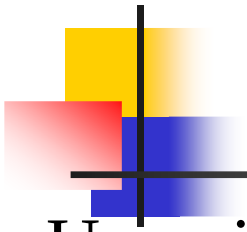


Pointeri şi tablouri. Alocarea dinamică a memoriei

1. Pointerii şi adresele
2. Pointerii şi tablourile
3. Aritmetica adreselor
4. Pointerii spre caractere
5. Alocarea dinamică



Pointeri

- Un pointer este o *variabilă* care *conține adresa unui obiect(altă variabilă sau funcție).*
- Orice variabilă are două elemente caracteristice: valoarea conținută în variabilă (*rvalue*) şi valoarea adresei locației de memorie (*lvalue*).

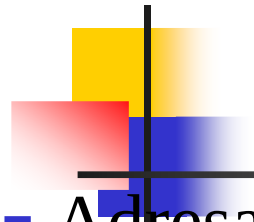
  Exemplu: `a = a + b;`
- Pointerii sunt folosiți mult în C deoarece:

  - permit exprimarea unor operații la un nivel mai scăzut;

  - conduc la un cod mai compact şi mai eficient.
- Trebuie folosiți cu grijă pentru a nu afecta claritatea şi simplitatea programelor.

Pointeri

Adrese de memorie



- Adresa locației de memorie în care este stocată o variabilă se poate obține aplicând *operatorul de adresă* (operatorul **&**) înaintea numelui variabilei (operație întâlnită frecvent în exemplele anterioare, în cazul apelării funcției scanf).
- Operatorul de adresă poate fi utilizat pentru obținerea valorii adresei oricărei variabile. Secvența următoare ilustrează acest lucru: se tipăresc valorile adreselor a două variabile, ca numere întregi fără semn, în reprezentare hexazecimală:

```
int a;
float x;
...
printf("adresa lui a este %x \n", &a);
printf("adresa lui x este %x \n", &x);
```

Pointeri
Adrese de memorie

- Valorile şi formatul adreselor de memorie depind de arhitectura calculatorului şi de sistemul de operare sub care rulează. => Din motive de portabilitate a programelor, dacă se doreşte declararea unei variabile care să conțină o adresă de memorie, nu se vor utiliza tipurile întregi.
- În C s-a definit un specificator de format special: ***%p***, pentru tipărirea valorilor reprezentând adrese de memorie.

```
int a;
float x;
printf("adresa lui a este %p \n", &a);
printf("adresa lui x este %p \n ", &x);
```

- Se recomandă utilizarea specificatorului de format %p pentru tipărirea adreselor (este o variantă portabilă).

Pointeri

Variabile pointer

- *Un pointer este o variabilă care are ca valoare o adresă de memorie.*

  Spunem că variabila pointer indică locația de memorie de la adresa pe care o conține (ceea ce am numit anterior, *lvalue).*

- Cel mai comun caz este acela în care valoarea unei variabile pointer este adresa unei alte variabile.

- În C, pointerii se referă la un anumit tip: tipul datelor conținute în locația de memorie indicată de variabila pointer.

=> O variabilă pointer în C este de regulă *pointer la un anumit tip*.

Pointeri

Variabile pointer

- Declararea variabilelor pointer se face în felul următor:

  *tip * nume_variabila_pointer;*

  variabila *nume_variabila_pointer* poate conține adrese de zone de memorie alocate unor date de tipul *tip*.

  * semnifică faptul că variabila este *pointer la tipul respectiv*.

- De exemplu,

  int *p;

definește o variabilă *p* pointer la întreg. Variabila *p* poate conține adrese de locații în care se memorează valori întregi.

Pointeri

Variabile pointer

- Declarațiile variabilelor pointer la un anumit tip pot să apară în aceeaşi listă cu variabilele obişnuite de acel tip:

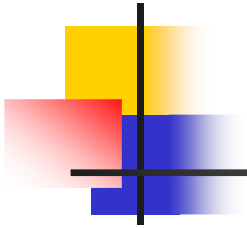
```
int x, *p;
```

definește o variabilă *x* de tip întreg şi o variabilă *p* de tip pointer la întreg.

- Se pot declara şi *pointeri generici*, de tip *void* * (tipul datelor indicate de astfel de pointeri nu este cunoscut).

Pointeri

Variabile pointer



- Un pointer de tip void reprezintă doar o adresă de memorie a unui obiect oarecare:

  - dimensiunea zonei de memorie indicate şi interpretarea informației conținute, nu sunt definite;

  - poate apare în atribuiri, în ambele sensuri, cu pointeri de orice alt tip;

  - folosind *o conversie explicită de tip* cu operatorul *cast* : ***(tip *)***, el poate fi convertit la orice alt tip de pointer; această conversie este posibilă deoarece, pe acelaşi calculator, toate adresele au aceeaşi lungime.

Pointeri

Utilizare – operatorii de adresare şi dereferenţiere

- Ca şi în cazul variabilelor de orice alt tip, şi variabilele pointer declarate şi neinițializate conțin valori aleatoare.
- Pentru a atribui variabilei *p* valoarea adresei variabilei *x*, se foloseşte operatorul de adresă într-o expresie de atribuire de forma:

```
p=&x;
```

În urma acestei operații, se poate spune că pointerul *p* indică spre variabila *x*.

- Pentru a obține valoarea obiectului indicat de un pointer (ceea ce am numit anterior, *rvalue)* se utilizează *operatorul de indirectare (dereferențiere), notat* *****.

Dacă *p* este o variabilă de tip pointer care are ca valoare adresa locației de memorie a variabilei întregi *x* (*p* indică spre *x*) atunci expresia **p* reprezintă o valoare întreagă (valoarea variabilei *x*).

Pointeri

Utilizare– operatorii de adresare şi dereferenţiere

```
int x, *p;
  x=3;
  p=&x;
  printf("%d", *p);
  *p=5;
  printf("%d", x);
```

Expresia **p* care se afişează cu primul printf reprezintă valoarea obiectului indicat de *p*, deci valoarea lui *x*. Se va afişa valoarea *3*.

Atribuirea *p=5; modifică valoarea obiectului indicat de *p*, adică valoarea lui *x*. Ultimul printf din secvență afişează *5* ca valoare a lui *x*.

Pointeri

Utilizare– operatorii de adresare şi dereferenţiere

- Întotdeauna când se aplică operatorul * asupra unui pointer ptr declarat ca şi tip * ptr;

expresia obținută prin dereferențiere (*ptr) este de tipul tip.

- În cazul unui pointer generic (de tip void *), dereferențierea trebuie precedată de *cast* (altfel nu ştim ce tip de valori indică pointerul).
- Când se utilizează operatorul de dereferențiere trebuie avut grijă ca variabila pointer asupra căreia se aplică să fi fost *inițializată*, adică să conțină adresa unei locații de memorie valide. O secvență ca cea de mai jos este incorectă şi poate genera erori grave la execuție: int *p; *p=3;

Pentru a indica adresă inexistentă, se utilizează ca valoare a unui pointer, constanta NULL.

Pointeri

Utilizare– operatorii de adresare şi dereferenţiere

- Variabilele pointer pot fi utilizate în expresii şi direct, (fără indirectare):

```
int x, *p1, *p2;
 x=3;
 p1=&x;
 p2=p1;    /* atribuire de pointeri */
```

- Atribuirea p2=p1; copiază conținutul lui *p1* în *p2*, deci *p2* va fi făcut să indice spre acelaşi obiect ca şi *p1*. În contextul secvenței de mai sus, *p2* va indica tot spre *x*.

Pointeri

Pointerii şi adresele

- *Exemple:*

```
int  x = 1, y = 2, z[10];
int *ip;         /* ip este un pointer la int */
ip = &x;         /* ip indica acum spre x */
y = *ip;         /* y este acum 1 */
*ip = 0;         /* x este acum 0 */
ip = &z[0];      /* ip indica acum spre z[0]  */
```

- Dacă ip indică spre x, atunci *ip poate apărea în orice context în care ar putea apărea x, deci

```
*ip  = *ip + 10;
```

îl incrementează pe x cu 10.

Pointeri

Pointerii şi adresele

- Operatorii unari * şi & au o precedență mai mare decât operatorii aritmetici, în consecință:

```
y = *ip + 1
```

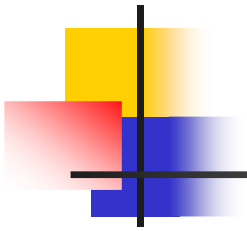
preia obiectul spre care indică ip, îl adună cu 1 şi atribuie rezultatul lui y, iar

```
*ip += 1
```

incrementează obiectul spre care indică ip, ca şi

```
++ *ip      şi
(*ip )++
```

- Operatorii unari precum * şi ++ se asociază de la dreapta la stânga.



Pointeri

Pointerii şi adresele

- Pointerii fiind variabile, pot fi folosiți şi fără a fi dereferențiați.

Dacă iq este un alt pointer spre tipul int,

iq = ip

copiază conținutul lui ip în iq, făcând astfel ca iq să indice spre ceea ce indică ip.

- Operatorul de *adresă* (&) se aplică numai obiectelor din memorie: variabile şi elemente de tablou. Operatorul & nu poate fi aplicat expresiilor, constantelor sau variabilelor de tip register.

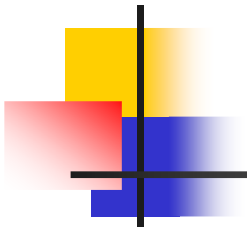
Tablouri
Exemple de utilizare

- Prelucrarea unui tablou unidimensional:

```
#define N 10
int t[N];  /* tablou de N elemente */
int i;  /* variabila contor */
for (i = 0; i < n; i++)
    *prelucrare t[i]
```

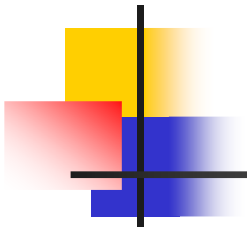
- *Exemplul 1*: Să se citească N numere întregi şi să se memorează în şirul t1. Să se construiască apoi şirul t2 cu elementele lui t1 în ordine inversă.

Tablouri
Exemple de utilizare



```
#include <stdio.h>
#define N 20

void main (void) {
   int t1[N], t2[N];
   int i;

 /* citeste elementele sirului t1 */
   printf("Introduceti cele %d elemente ale sirului \n", N);
   for (i = 0; i < N; i++)
      scanf("%d", &t1[i]);
```

Tablouri
Exemple de utilizare



```
   /* construieste t2 */
   for (i = 0; i < N; i++)
      t2[i] = t1[N – 1 - i];

   /* afiseaza elementele sirului t2 */
   printf(“Sirul in ordine inversa este: \n”);
   for(i = 0; i < N; i++)
      printf(“%d ”, t2[i]);
   printf(“\n”);
}
```

Tablouri
Exemple de utilizare

- *Citirea* elementelor unei matrici cu N linii şi M coloane:

```
printf("Introduceti elementele matricii: \n");
for(i = 0; i < N; i++)
   for(j = 0; j< M; j++)
      scanf("%d", &a[ i ][ j ]);
```

- *Afişarea* elementelor unei matrici:

```
for(i = 0; i < N; i++)  {
   for(j = 0; j < M; j++)
      printf("%4d", a[ i ][ j ]);
    printf("\n");
}
```

Tablouri
Exemple de utilizare

- *Exemplul 2*: Să se calculeze suma elementelor de sub diagonala principală a unei matrici pătrate cu n ($n \leq 20$) linii şi coloane.

```
#include <stdio.h>
void main(void)
{
    int a[20][20];
    int suma, i, j;
    printf("Introduceti dimensiunea matricii (<=20):");
    scanf("%d", &n);
```

Tablouri
Exemple de utilizare

```
/* citirea elementelor matricii */
for(i = 0; i < n; i++)
   for(j = 0; j< n; j++)  {
      printf("a[%d][%d] = ", i+1, j+1);
      scanf("%d", &a[ i ][ j ]);
    }
/* calculul si afisarea sumei */
for(i = 0, suma = 0; i < n; i++)
   for(j = 0; j< i; j++)
       suma += a[ i ][ j ];
 printf("Suma calculata este: %d\n", suma);
}
```

Tablouri
Exemple de utilizare

- Transmiterea tablourilor ca parametri la funcții.
- *Exemplul 3*: Să se scrie un program care calculează suma a două matrici, S1=A+B precum şi suma S2=S1+A.

```
#include <stdio.h>
int n, m;

void citire (int a[10][10])
{
   int i, j;
   printf("Introduceti elementele:");
```

Tablouri
Exemple de utilizare

```
 for(i = 0; i < n; i++)
   for(j = 0; j< m; j++)  {
      printf("a[%d][%d] = ", i+1, j+1);
      scanf("%d", &a[ i ][ j ]); }
}
void suma(int a[10][10], int b[10][10], int s[10][10])   {
   int i, j;
   for(i = 0; i < n; i++)
      for(j = 0; j< m; j++)
         s[ i ][ j ] = a[ i ][ j ] + b[ i ][ j ];
 }
```

Tablouri
Exemple de utilizare

```
void tiparire(int a[10][10])   {
   int i, j;
   for(i = 0; i < n; i++)   {
         for(j = 0; j< m; j++)
             printf("%8d", a[ i ][ j ]);
         printf("\n");
}
void main(void)  {
   int a[10][10], b[10][10], s1[10][10], s2[10][10];
    printf("Introduceti dimensiunile matricilor:");
    scanf("%d%d", &n, &m);
```

Tablouri
Exemple de utilizare

```
/* Citirea matricilor */
printf(“Introduceti elementele matricii a \n”);
citire(a);
printf(“Introduceti elementele matricii b \n”);
citire(b);

/* Tiparirea matricilor */
printf(“Matricea a este :\n”);
tiparire(a);
printf(“Matricea b este :\n”);
tiparire(b);
```

Tablouri
Exemple de utilizare

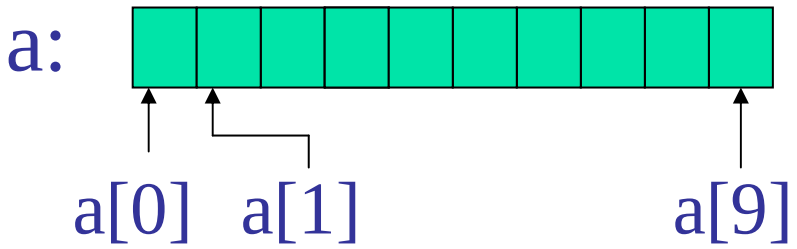
```
   /* Calculul si afisarea matricilor suma */

   suma(a, b, s1);
   printf("Matricea suma dintre a si b este:\n");
   tiparire(s1);

   suma(s1, a, s2);
   printf("Matricea suma dintre s1 si a este:\n");
   tiparire(s2);
}
```

Pointeri şi tablouri

- Orice operație care poate fi realizată cu ajutorul tablourilor (variabilelor cu indici) poate fi, de asemenea, efectuată cu ajutorul pointerilor.
- Declarația

```
int a[10];
```

definește un tablou de dimensiune 10, adică un bloc de 10 elemente consecutive notate a[0], a[1], ..., a[9].



Pointeri şi tablouri

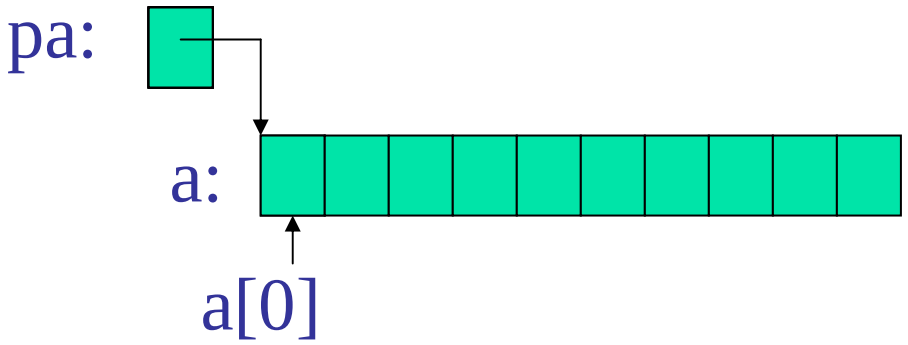
- Dacă pa este un pointer spre un întreg, declarat ca

```
int *pa;
```

atunci atribuirea

```
pa = &a[0];
```

îl setează pe pa să indice spre elementul zero al lui a; mai precis, pa conține adresa lui a[0].



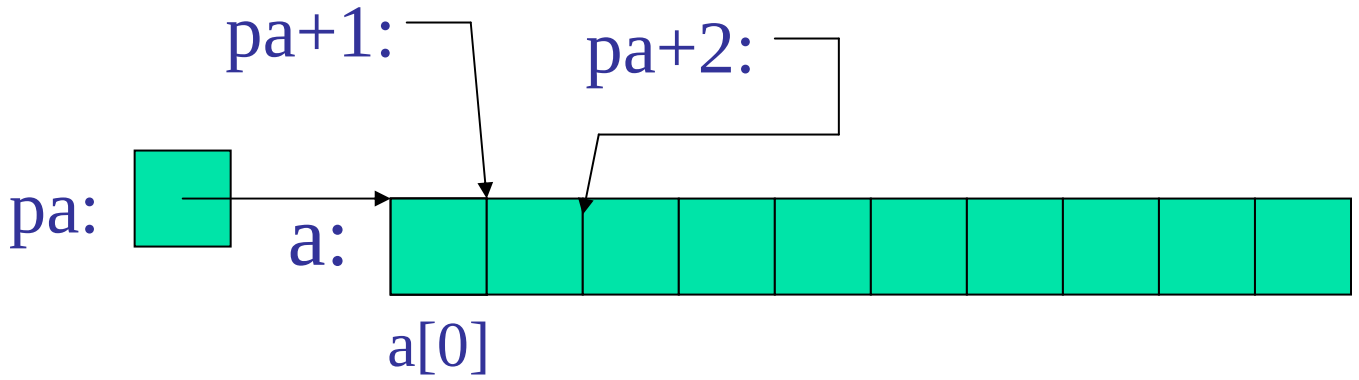
Pointeri şi tablouri

- Instrucțiunea

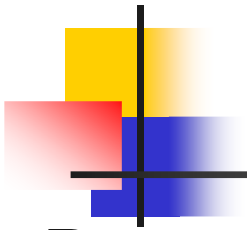
```
x = *pa;
```

va copia conținutul lui a[0] în x.

- Dacă pa indică spre un anumit element de tablou, atunci, prin definiție, pa + 1 indică spre următorul element, pa + i indică spre o locație aflată la i elemente după pa, iar pa – i indică spre o locație aflată cu i elemente înainte.



Pointeri şi tablouri



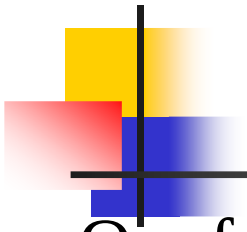
- Remarcile anterioare sunt valabile indiferent de tipul sau mărimea variabilelor din tabloul a.
- Înțelesul expresiei "*adună 1 la un pointer*" şi, prin extensie, întreaga *aritmetică a pointerilor* este că pa + 1 indică spre următorul obiect, iar pa + i indică spre al i-lea obiect de după pa.
- Prin definiție, *valoarea unei variabile* sau *expresii de tip tablou este adresa elementului zero al tabloului*.

```
pa = &a[0];
```

poate fi scrisă

```
pa = a;
```

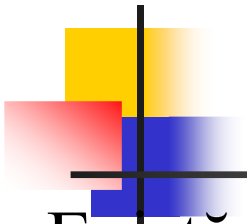
Pointeri şi tablouri



- O referință la a[i] poate fi scrisă şi ca *(a+i); cele două forme sunt echivalente.
- Aplicând operatorul **&** ambelor părți ale echivalenței, rezultă că &a[i] şi a+i sunt forme identice.
- Dacă pa este un pointer, în expresii acesta poate fi folosit cu un indice; pa[i] este identic cu *(pa+i).

- Concluzie: ***o expresie cu tablou şi indice este echivalentă cu una scrisă ca pointer şi distanță de deplasare.***

Pointeri şi tablouri



- Există totuşi o *diferență* între un nume de tablou şi un pointer.

  Un pointer este o variabilă:

  `pa=a` şi `pa++` sunt *expresii legale*

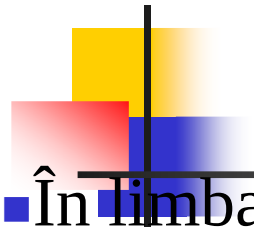
  Un nume de tablou nu este o variabilă:

  `a=pa` şi `a++` sunt *expresii ilegale*

- Când un nume de tablou este *transmis*, *ca argument*, *unei funcții*, ceea ce se transmite este *o adresă* reprezentând *locația elementului inițial*.

- În interiorul funcției apelate, acest argument este o variabilă locală; deci un **nume de tablou sub formă de parametru este un pointer**, adică o variabilă care conține o adresă.

Pointeri şi tablouri



- În limbajul C, numele unui tablou este echivalent cu un pointer care conține adresa primului său element.
- Fie următoarele declarații: *tab* un tablou de elemente de tip *T* şi *p* un pointer la tipul *T*:

```
T tab[N];
T * p;
```

Cele 3 atribuiri următoare sunt echivalente, si au ca efect faptul că *p* va indica adresa primului element al tabloului *tab*:

```
p=tab;  p=&tab;  p=&tab[0];
```

- Atribuirea `tab=p;` nu este permisă.
- Numele tabloului indică o adresă constantă: adresa primului element al tabloului este o valoare constantă din momentul în care tabloul a fost alocat static, deci nu mai poate fi modificată printr-o atribuire ca cea de mai sus.

Pointeri şi tablouri

```
/* strlen: returneaza lungimea sirului s */
int strlen(char *s)  {
   int n;
   for (n = 0; *s !='\0'; s++)
      n++;
   return n;   }
```

- Sunt corecte toate apelurile următoare:

```
strlen("buna ziua");     /* constanta sir */
strlen(tablou);          /* char tablou[100]; */
strlen(ptr);             /* char *ptr; */
```

Pointeri şi tablouri

- Ca *parametri formali* din definiția unei funcții, declarațiile

  char s[ ]; şi char *s; sunt echivalente.

  Se preferă **char *s**; deoarece precizează într-un mod mai explicit faptul că parametrul este un pointer.

- Când numele unui tablou este transmis unei funcții, *funcția poate considera*, după cum preferă, că i-a fost transmis *fie un tablou*, *fie un pointer*. Poate folosi *chiar ambele notații* dacă acest lucru este considerat potrivit şi clar.

- Se poate transmite funcției doar o parte dintr-un tablou:

  f(&a[2]) sau f(a+2)

  În interiorul lui f declarația parametrului este:

  f(int tabl[ ]) { ... } sau f(int *tabl) { ... }

Pointeri şi tablouri
Aritmetica adreselor

- Toate operațiile de manipulare a pointerilor iau în considerare în mod automat dimensiunea obiectului spre care se indică.

- *Operațiile permise cu pointeri* sunt:
  - atribuirea pointerilor de acelaşi tip;
  - adunarea unui pointer cu un întreg sau scăderea unui întreg;
  - scăderea sau compararea a doi pointeri care indică spre elemente ale aceluiaşi tablou;
  - atribuirea valorii zero (NULL) sau compararea cu aceasta.

Pointeri şi tablouri

Aritmetica pointerilor-incrementare şi decrementare

- Operatorii unari de *incrementare* şi *decrementare* se pot aplica asupra variabilelor de tip pointer, în format prefix şi postfix.

```
T  *p;
p++;  p--;
++p;  --p;
```

- Operatorul de incrementare aplicat asupra unui operand de tip pointer la tipul *T* măreşte adresa conținută de operand cu numărul de octeți necesar pentru a păstra o dată de tipul *T* (se adună *sizeof(T)):*

```
T  tab[N];
T  *p;
int i;
p=&tab[i];
p++;
```

p va contine adresa elementului *tab[i+1];*

Pointeri şi tablouri

Aritmetica pointerilor – Adunarea şi scăderea unui întreg

- Dacă *p* este un pointer la tipul *T* şi *n* o valoare întreagă, expresia *p+n* măreşte valoarea lui *p* cu *n*sizeof(T)*. Analog, expresia *p-n* micşorează valoarea lui *p* cu *n*sizeof(T);*
- Dacă se declară un tablou *tab* şi se utilizează ca în secvența de mai jos:

```
T  tab[N];
T  * p;
p=tab;
```

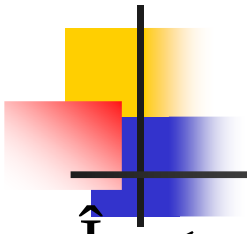
expresia *p+n* va indica spre elementul *tab[n]*.

Valoarea elementului *tab[n]* poate fi reprezentată şi prin expresia **(p+n)*.

- ***Variabilele cu indici se pot înlocui prin expresii cu pointeri.***

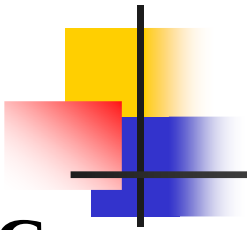
Pointeri şi tablouri

Aritmetica pointerilor - iniţializare



- În general, un *pointer poate fi iniţializat* ca orice altă variabilă deşi, în mod normal, singurele valori care au sens sunt zero sau o expresie care implică adresele unor date definite anterior.

- *Constanta zero* poate fi atribuită unui pointer şi un pointer poate fi comparat cu constanta *zero*. *Constanta simbolică NULL* (definită în <stdio.h>) este adesea folosită în locul lui *zero*, ca o modalitate de a indica mai clar că aceasta este o valoare specială pentru un pointer.



Pointeri şi tablouri

Aritmetica pointerilor

Compararea a doi pointeri

- Doi pointeri care indică spre elementele aceluiaşi tablou pot fi comparați, folosind operatorii de relație şi de egalitate.

- Dacă *p* şi *q* sunt doi pointeri spre elementele *tab[i]* respectiv *tab[j]* ale unui tablou, comparația *p<q* este adevărată dacă şi numai dacă *i<j*.

Diferența a doi pointeri

- Doi pointeri care indică spre elementele aceluiaşi tablou pot fi scăzuți. Dacă *p* şi *q* sunt doi pointeri spre elementele *tab[i]* şi respectiv *tab[i+n]*, diferența *q-p* are valoarea egală cu *n*.

Pointeri şi tablouri

Aritmetica pointerilor

- Dacă p şi q indică spre elemente ale aceluiaşi tablou şi p<q, atunci q-p+1 reprezintă numărul de elemente de la p la q inclusiv.

```
/* strlen: returneaza lungimea sirului s */
int strlen(char *s)
{
   char *p = s;  /* initializare cu primul element al sirului*/
   while (*p != '\0')
      p++;
   return p-s;  /* returneaza nr. de caractere parcurse*/
}
```

Pointeri şi tablouri
Aritmetica pointerilor

- Sunt *ilegale* următoarele operații aritmetice asupra pointerilor:
  - adunarea a doi pointeri;
  - înmulțirea sau împărțirea pointerilor;
  - deplasarea sau aplicarea unor măşti;
  - adunarea cu valori de tip float sau double ;
  - atribuirea a doi pointeri de tipuri diferite fără a folosi operatorul cast (cu excepția tipului void *).

Pointeri şi tablouri

Prelucrarea tablourilor

- Exemplu: parcurgerea elementelor unui tablou.

```
void tiparire1(int tab[ ], int N) {
    int i;
    for (i=0; i<N; i++)
      printf("%d  ",tab[ i ]);  }
void tiparire2(int tab[ ],int N) {
    int * ptr;
    for (ptr=tab; ptr<tab + N; ptr++)
      printf("%d  ", *ptr);   }
void tiparire3(int *tab, int N) {
    int * ptr;
    for (ptr=tab; ptr<tab + N; ptr++)
      printf("%d  ", *ptr);   }
```

Pointeri şi tablouri

Prelucrarea tablourilor

```
void tiparire4(int *tab, int N) {
        int i;
        for (i=0; i<N; i++, tab++)
          printf("%d  ", *tab);
      }
```

Apelurile celor 4 funcții au acelaşi efect:

```
void main(void) {
        int a[5]={1,2,3,4,5};
        tiparire1(a,5);
        tiparire2(a,5);
        tiparire3(a,5);
        tiparire4(a,5);
      }
```

Pointeri şi tablouri

Pointerii spre caractere

- O *constantă şir*, scrisă sub forma:

  “Eu sunt un sir de caractere”

este un tablou de caractere. În reprezentarea internă, tabloul se termină cu caracterul ‘\0’, pentru ca în program să se poată găsi sfârşitul şirului. *Lungimea spațiului de stocare este de aceea cu unu mai mare decât decât numărul de caractere* aflate între ghilimele.

- Poate cea mai frecventă utilizare a constantelor şir este ca argumente ale funcțiilor:

  printf(“Buna ziua\n”);

Accesul la acest şir se face printr-un pointer la tipul caracter; funcția printf primeşte un pointer către începutul tabloului de caractere. Pointerul accesează deci primul element al şirului.

Pointeri şi tablouri

Pointerii spre caractere

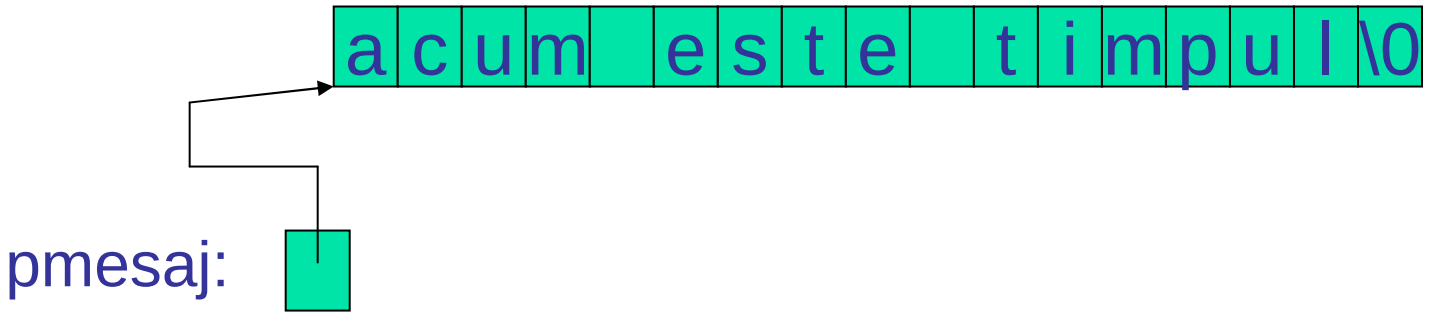
- Constantele şir nu trebuie să fie neapărat argumente de funcţii. Dacă pmesaj este declarat ca:

```
char *pmesaj;
```

atunci instrucţiunea

```
pmesaj = “acum este timpul”;
```

îi atribuie lui pmesaj un pointer către tabloul de caractere.

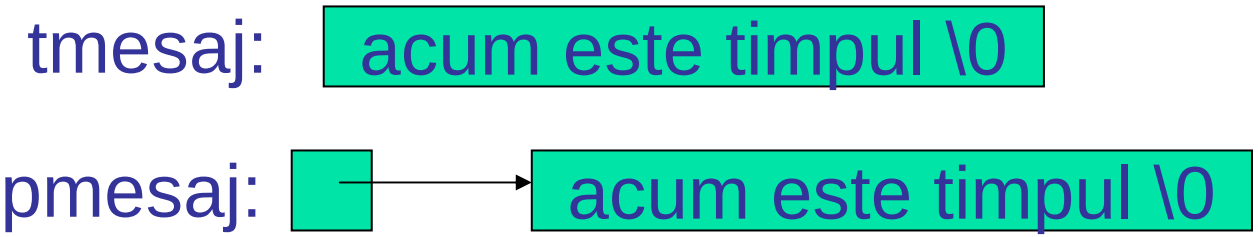


Pointeri şi tablouri

Pointerii spre caractere

- Există o *deosebire importantă* între următoarele definiții:

```
char tmesaj[ ] =  “acum este timpul”;   /* un tablou */
char *pmesaj = “acum este timpul”;      /* un pointer */
```



- Caracterele din cadrul tabloului pot fi modificate individual, dar tmesaj va indica întotdeauna către acelaşi spațiu de stocare.

- pmesaj este un pointer inițializat să indice către o constantă şir; poate fi modificat ulterior astfel încât să indice altă locație, dar *rezultatul este nedefinit dacă se modifică conținutul şirului.*

Pointeri şi tablouri

Pointerii spre caractere

- *Exemplul 1*: funcția strcpy, care copiază un şir în alt şir.

```
/* strcpy: copiaza t in s; versiunea cu indici de tablou */
void strcpy(char *s, char *t)
{
   int i;
   i = 0;
   while ((s[ i ] = t[ i ]) != '\0')
       i++;
}
```

Pointeri şi tablouri

Pointerii spre caractere

```
/* strcpy: copiaza t in s; versiunea cu pointeri */
void strcpy(char *s, char *t)
{
   while ((*s = *t) != '\0')  {
       s++;
       t++;
   }
}
```

- Deoarece argumentele sunt transmise prin valoare, strcpy poate folosi parametrii s şi t cum doreşte.

Pointeri şi tablouri

Pointerii spre caractere

- În practică, strcpy nu ar fi scrisă ca în exemplele anterioare. Programatorii C experimentați ar prefera:

```
/* strcpy: copiaza t in s; versiunea cu pointeri 2*/
void strcpy(char *s, char *t)
{
   while ((*s++ = *t++) != '\0')
         ;
}
```

- Această versiune mută incrementarea lui s şi t în partea de testare a ciclului. În final se copiază în s inclusiv terminatorul '\0'.

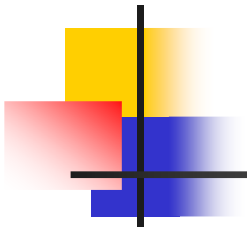
Pointeri şi tablouri

Pointerii spre caractere

- Putem realiza încă o compactare, observând că nu este necesară comparația cu '\0' pentru a controla ciclul (întrebarea este doar dacă expresia este zero). Efectul final este copierea caracterelor din t în s, până la şi inclusiv terminatorul '\0'.

```
/* strcpy: copiaza t in s; versiunea cu pointeri 3*/
void strcpy(char *s, char *t)
{
   while (*s++ = *t++)
        ;
}
```

- *Convenția de notație este demnă de luat în seamă, iar stilul din acestă variantă trebuie însuşit (se întâlneşte frecvent).*



Pointeri şi tablouri

Pointerii spre caractere

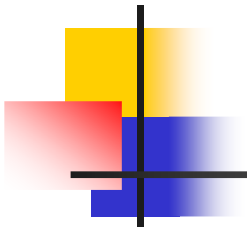
- *Exemplul 2*: funcția strcmp, care compară lexicografic două şiruri de caractere (s şi t) şi returnează o valoare:

- negativă, dacă s < t
- zero, dacă s == t
- pozitivă, dacă s > t.

Valoarea se obține prin scăderea caracterelor de la prima poziție în care t şi s diferă.

Pointeri şi tablouri

Pointerii spre caractere



```
/* strcmp: compara sirurile de caractere s si t;
                    versiunea cu indici de tablouri  */
int strcmp (char *s, char *t)
{
   int;
   for (i = 0; s[ i ] == t[ i ]; i++)
      if (s[ i ] == '\0')
          return 0;
   return s[ i ] – t[ i ];
}
```

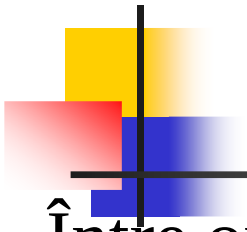
Pointeri şi tablouri

Pointerii spre caractere

```
/* strcmp: compara sirurile de caractere s si t;
                    versiunea cu pointeri  */
int strcmp (char *s, char *t)
{
   for ( ; *s == *t; s++, t++)
      if (*s == '\0')
          return 0;
   return *s – *t;
}
```

Pointeri şi tablouri

Pointerii spre caractere



- Între operatorii *, ++ şi -- pot să apară următoarele combinații. De exemplu:

```
*--p
```

îl decrementează pe p înainte de a prelua caracterul spre care indică p.

Perechea de expresii:

```
*p++ = val;   /* introdu val in stiva */
val = *--p;  /* scoate varful stivei si copiaza-l in val */
```

constituie instrucțiunile standard pentru introducerea şi scoaterea unui element din stivă.

Alocarea dinamică a memoriei

- Multe aplicații pot fi optimizate dacă memoria necesară stocării datelor lor este alocată *dinamic* în timpul execuției programului.
- Alocarea dinamică de memorie înseamnă alocarea de zone de memorie şi eliberarea lor în timpul execuției programelor.
- Zona de memorie în care se alocă, la cerere, datele dinamice este diferită de zona de memorie în care se alocă datele statice(stiva de date) şi se numeşte HEAP.
- Spațiul de memorie alocat dinamic, la cerere, este accesibil programatorului printr-un pointer care conține adresa sa. Pointerul care indică un obiect din HEAP este de regulă stocat într-o variabilă alocată static.

Pointeri şi tablouri
Alocarea dinamică

- Momentele alocării şi cel al eliberării zonei de memorie pot fi aleatorii pe durata execuției programului şi sunt stabilite de programator.
- Pentru alocarea şi eliberarea memoriei alocate dinamic există funcții oferite de biblioteca C standard.
- Funcțiile de gestionare a memoriei au prototipurile în fişierele alloc.h şi stdlib.h. Cele mai utilizate dintre acestea sunt:

malloc – alocarea zonei de memorie

free – eliberarea zonei de memorie ocupată cu malloc

Pointeri şi tablouri
Alocarea dinamică

Funcția de alocare dinamică a memoriei (malloc)

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

Funcția alocă în HEAP un bloc de dimensiune size; dacă operația reuşeşte returnează un pointer la blocul alocat, altfel returnează NULL.

- Este posibil ca cererea de alocare să nu poată fi satisfăcută dacă nu mai există suficientă memorie în zona de HEAP sau nu există un bloc compact de dimensiune cel puțin egală cu size.

- Dimensiunea size se specifică în octeți.

Pointeri şi tablouri
Alocarea dinamică

- Rezultatul funcției malloc este un pointer de tip void.

- Se va folosi operatorul cast pentru conversii de tip la atribuirea rezultatului funcției către un pointer de un anumit tip.

- Alocarea dinamică a unei zone pentru o valoare de tip int :

```
int * ip;

ip=(int * )malloc(sizeof(int));
*ip=5;
```

Pointeri şi tablouri
Alocarea dinamică

- Chiar dacă variabila pointer care indică o astfel de zonă de memorie nu mai există (şi-a depăşit durata de viață) zona alocată rămâne în continuare ocupată, devenind însă inaccesibilă pentru program.
- O eroare frecventă de acest gen poate apare la utilizarea variabilelor dinamice în cadrul funcțiilor:

```
void fct()
{
    int * p;
    p=(int *)malloc(10* sizeof(int));
}
```

Pointeri şi tablouri
Alocarea dinamică

Funcția de eliberare dinamică a memoriei (free)

```
void free(void *p);
```

Funcția eliberează un bloc alocat anterior cu malloc; adresa de început a blocului este transmisă ca argument, la apelul funcției.

- Transferul unei valori invalide (un pointer la un bloc de memorie care nu a fost rezultatul unui apel al funcției malloc) poate compromite funcționarea sistemului de alocare.

Pointeri şi tablouri
Alocarea dinamică

- Un tablou poate fi alocat dinamic printr-o secvență ca cea de mai jos:

```
TIP * p;
p= (TIP *) malloc(N*sizeof(TIP));
```

Pointerul p va indica un bloc suficient de mare pentru a conține N elemente de tipul TIP.

În continuare, variabila p poate fi utilizată ca şi cum ar fi fost declarată ca un tablou de forma:

```
TIP p[ N ];
```

- Avantajul alocării dinamice a unui tablou este că dimensiunea sa poate fi specificată doar în timpul execuției.

Pointeri şi tablouri
Alocarea dinamică

- *Exemplul 1*: Alocarea dinamică de memorie pentru un tablou de n numere întregi:

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

main(void)
{
   int n;
   int * tab;
   int i;
   printf("Introduceti numarul de elemente: \n");
   scanf("%d", &n);
```

Pointeri şi tablouri
Alocarea dinamică

```
  if ((tab=(int *)malloc(n * sizeof(int)))==NULL) {
     printf("Eroare alocare dinamica memorie !\n");
     exit(1);
   }
   for (i=0; i<n; i++)
      scanf("%d", &tab[ i ]);
   for (i=0; i<n; i++)
      printf("%d ", tab[ i ]);
   free(tab);
}
```

Pointeri şi tablouri
Alocarea dinamică

- *Exemplul 2:* Să se citească de la tastatură un şir de caractere sir1. Să se creeze o copie dinamică a acestuia, notată sir2*:*

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
main(void) {
    char sir1[40], *sir2;
    printf("introduceti un sir de caractere: \n");
    scanf("%40s", sir1);
    if ((sir2=(char *)malloc(strlen(sir1)+1))==NULL) {
        printf("eroare aloc. dinamica memorie !\n");exit(1);}
      strcpy(sir2, sir1);
        printf("copia sirului este: %s \n", sir2);   .....  }
```

Pointeri şi tablouri
Alocarea dinamică

- *Exemplul 3:* Definirea si utilizarea unui vector alocat dinamic (varianta 2):

```
main( ) {
        int n,i; int * a;

        printf ("n="); scanf ("%d", &n);
        a=(int *) calloc (n, sizeof(int));
        // sau: a=(int*) malloc (n*sizeof(int));
        printf ("componentele vectorului: \n");
        for (i=0; i<n; i++)
            scanf ("%d", &a[ i ]); // sau scanf ("%d", a+i);
        for (i=0;i; i<n; i++)
            printf ("%d ",a[ i ]);  // sau printf ("%d ", *(a+i));
}
```

Pointeri şi tablouri
Alocarea dinamică

- *Exemplul 4*: Alocarea dinamică a unei matrici:

```
main () {
        int ** a; int i, j, nl, nc;
        printf ("nr. linii="); scanf ("%d",&nl);
        printf ("nr. col. ="); scanf ("%d",&nc);
        // memorie pentru vectorul de pointeri la linii
        a = (int**) malloc (nl*sizeof(int*));
        for (i=0; i < nl; i++)
        // aloca memorie pentru fiecare linie i
        a[i] = (int*) calloc (nc, sizeof(int)); // o linie
           .........................
}
```

Pointeri şi tablouri
Alocarea dinamică

- Pentru o matrice alocată dinamic, cu elemente întregi, notația a[i][j] este interpretată astfel :

  - a[i] conține un pointer (o adresa *b*)

  - b[j] sau b+j conține întregul din poziția "j" a vectorului cu adresa "*b*".

Pointeri şi tablouri
Alocarea dinamică

- *Exemplul 5*: Se citeşte un număr necunoscut de valori întregi într-un vector extensibil:

```
#define INCR 100 //cu cat creste vectorul la fiecare realocare
main() {
        int n,i,m ;
        float x, * v; // v = adresa vector
        n=INCR; i=0;
        v = (float *)malloc (n*sizeof(float)); //alocare initiala
        while ( scanf("%f",&x) != EOF) {
        if (++i == n) { // daca este necesar
        n= n+ INCR; //creste dimensiunea vectorului
        v=(float *) realloc (v, n*sizeof(float)); }
        v[i]=x; // memorarea lui x in vector  }.....
}
```

Pointeri şi tablouri

Alocarea dinamică - concluzii

- Variabilele alocate în HEAP sunt **dinamice**. Ele se numesc astfel pentru apar şi pot să dispară pe parcursul execuției programului, în mod dinamic. Alocarea spațiului necesar pentru o asemenea variabilă ca şi relocarea (eliberarea) lui, se efectuează în mod explicit în program, prin apelurile malloc şi free.
- Timpul scurs din momentul creării locației unei variabile şi până la desființarea acestei locații se numeşte **durata de viață a variabilei**. Pentru variabilele obişnuite (automatice), durata de viață coincide cu durata de activare a blocului de care aparțin. Pentru variabilele dinamice apariția şi dispariția lor are loc independent de structura textului programului, iar durata de viață este intervalul de timp scurs între apelurile funcțiilor malloc şi free pentru acele variabile.