# Capitolul IB.07. Pointeri. Pointeri şi tablouri. Pointeri şi funcţii

*Cuvinte cheie*
Pointer, referenţiere, dereferenţiere, indirectare, vectori şi pointeri, tablouri ca argumente ale funcţiilor, pointeri în funcţii, pointeri la funcţii, funcţii generice, tipul referinţă

## IB.07.1. Pointeri

**Pointerii** reprezintă cea mai puternică caracteristică a limbajului C/C++, permiţând programatorului să acceseze direct conţinutul memoriei, pentru a eficientiza astfel gestiunea memoriei; programul şi datele sale sunt păstrate în memoria RAM ( *Random Access Memory* ) a calculatorului.
În acelaşi timp, pointerii reprezintă şi cel mai complex şi mai dificil subiect al limbajului C/C++, tocmai datorită libertăţilor oferite de limbaj în utilizarea lor.
Prin utilizarea corectă a pointerilor se paote îmbunătăţi drastic eficienţa şi performanţele programului. Pe de altă parte, utilizarea lor incorectă duce la apariţia multor probleme, de la cod greu de citit şi de întreţinut la greşeli penibile de genul pierderi de memorie sau depăşirea unei zone de date. Utilizarea incorectă a pointerilor poate expune programul atacurilor externe (hacking). Multe limbaje noi (Java, C#) au eliminat pointerii din sintaxa lor pentru a evita neplăcerile cauzate de aceştia.

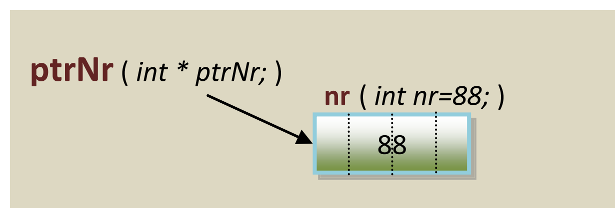
O locaţie de memorie are o adresă şi un conţinut. Pe de altă parte, o variabilă este o locaţie de memorie care are asociat un nume şi care poate stoca o valoare de un tip particular. În mod normal, fiecare adresă poate păstra 8 biţi (1 octet) de date. Un întreg reprezentat pe 4 octeţi ocupă 4 locaţii de memorie. Un sistem 'pe 32 de biţi' foloseşte în mod normal adrese pe 32 de biţi. Pentru a stoca adrese pe 32 de biţi sunt necesare 4 locaţii de memorie.

**Definiţie:**

**O variabilă pointer (pe scurt vom spune un pointer) este o variabilă care păstrează adresa unei date, nu valoarea datei.**
**Cu alte cuvinte, o variabilă pointer este o variabilă care are ca valori adrese de memorie. Aceste adrese pot fi:**

- **Adresa unei valori de un anumit tip (pointer la date)**
- **Adresa unei funcţii (pointer la o funcţie)**
- **Adresa unei zone cu conţinut necunoscut (pointer la *void*).**

În figura următoare, s-a reprezentat grafic un pointer *ptrNr* care este un pointer la o variabilă *nr* (de tip *int*); cu alte cuvinte, *ptrNr* este o variabilă pointer ce stochează adresa lui *nr*. În general, vom reprezenta grafic pointerii prin săgeţi.



Un pointer poate fi utilizat pentru referirea diferitelor date şi structuri de date, cel mai frecvent folosindu-se pointerii la date. Schimbând adresa memorată în pointer, pot fi manipulate informații situate la diferite locații de memorie, programul şi datele sale fiind păstrate în memoria RAM a calculatorului.
Deşi adresele de memorie sunt de multe ori numere întregi pozitive, tipurile pointer sunt diferite de tipurile întregi şi au utilizări diferite. Unei variabile pointer i se pot atribui constante întregi ce reprezintă adrese, după conversie .

**În limbajul C tipurile pointer se folosesc în principal pentru:**

- **declararea şi utilizarea de vectori, mai ales pentru vectori ce conțin şiruri de caractere;**
- **parametri de funcții prin care se transmit rezultate (adresele unor variabile din afara funcției);**
- **acces la zone de memorie alocate dinamic şi care nu pot fi adresate printr-un nume;**
- **parametri de funcții prin care se transmit adresele altor funcții.**

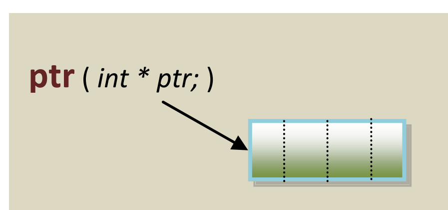
## IB.07.2. Declararea pointerilor

Ca orice variabilă, pointerii trebuie declarați înainte de a putea fi utilizați.
În sintaxa declarării unui pointer se foloseşte caracterul * înaintea numelui pointerului. Declararea unei variabile (sau parametru formal) de un tip pointer include declararea tipului datelor (sau funcției) la care se referă acel pointer. Sintaxa declarării unui pointer la o valoare de tipul "tip" este:

**Sintaxa:**

```
tip * ptr;        // sau
tip* ptr;         //sau
tip  *ptr;
```

Reprezentarea grafică corespunzătoare acestei declarații este următoarea:




Exemple de variabile şi parametri pointer:

```
int * pi;              // pi - adresa unui intreg sau vector de int
void * p;              // p - adresa de memorie
int * * pp;            // pp - adresa unui pointer la un intreg
char* str;  // str - adresa unui şir de caractere
```

Atunci când se declară mai multe variabile pointer de acelaşi tip, nu trebuie omis asteriscul care arată că este un pointer.

*Exemple:*

```
int *p, m;          // m de tip "int", p de tip "int *"
int *a, *b ;        // a şi b de tip pointer
```

Convenția de nume pentru pointeri sugerează să se pună un prefix sau sufix cu valoarea "p" sau "ptr". Exemplu: *iPtr, numarPtr, pNumar, pStudent*.

Dacă se declară un tip pointer cu *typedef* atunci se poate scrie astfel:

```
typedef int* intptr;          // intptr este nume de tip
intptr  p1, p2, p3;           // p1, p2, p3 sunt pointeri
```

Tipul unei variabile pointer este important pentru că determină câți octeți vor fi folosiți de la adresa conținută în variabila pointer şi cum vor fi interpretați. Un pointer la *void* nu poate fi utilizat pentru a obtine date de la adresa din pointer, deoarece nu se ştie câți octeți trebuie folosiți şi cum.

Există o singură constantă de tip pointer, cu numele *NULL* şi valoare zero, care este compatibilă la atribuire şi comparare cu orice tip pointer.

**Observații:**
Totuşi, se poate atribui o constantă întreagă convertită la un tip pointer unei variabile pointer:

```
char * p = (char*)10000;                // o adresa de memorie
```

## IB.07.3. Operații cu pointeri la date

### IB.07.3.1 Inițializarea pointerilor, operația de referențiere (&)

Atunci când declarăm un pointer, el nu este inițializat. Cu alte cuvinte, el are o valoare oarecare ce reprezintă o adresă a unei locații de memorie oarecare despre care bineînțeles că nu ştim dacă este validă (acest lucru este foarte periculos, pentru că poate fi de exemplu adresa unei alte variabile!). Trebuie să inițializăm pointerul atribuindu-i o adresă validă.
Aceasta se poate face în general folosind operatorul de referențiere - de luare a adresei - (&).

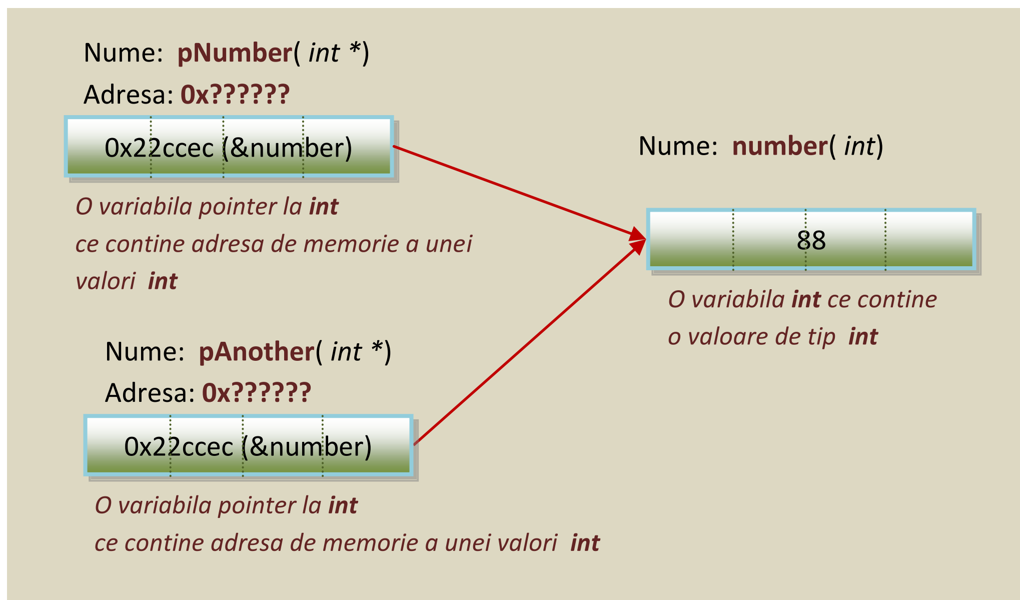
**Sintaxa:**

**Operatorul unar & aplicat unei variabile are ca rezultat adresa variabilei respective**

De exemplu, dacă ***nr*** este o variabilă de tip ***int***, ***&nr*** returnează adresa lui ***nr***. Această adresă o putem atribui unei variabile pointer:

```
int number = 88;        // o variabila int cu valoarea 88
int *pNumber;           // declaratia unui pointer la un intreg
pNumber = &number;      // atribuie pointerului adresa variabilei int
pNumber int *pAnother = &number;       /* declaratia unui pointer la un
                         intreg si initializare cu adresa variabilei int */
```

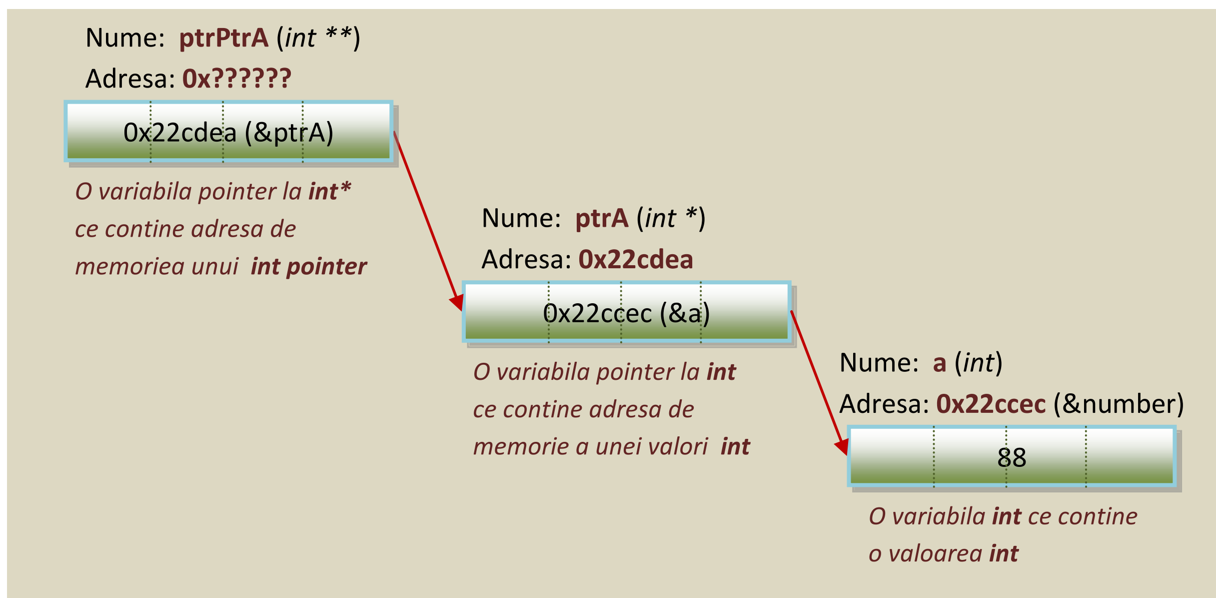
În figură, variabila *number*, memorată începând cu adresa 0x22ccec, conține valoarea întreagă 88. Expresia *&number* returnează adresa variabilei *number*, adresă care este 0x22ccec. Această adresă este atribuită pointerului *pNumber*, ca valoare a sa inițială, dar şi pointerului *pAnother*, ca urmare cei doi pointeri vor indica aceeaşi celulă de memorie!




*Exemplu:*

```
int **ptrPtrA, *ptrA, a=1;
// ptrPtrA - pointer la un pointer la un intreg
// ptrA - pointer la un intreg
// a - variabila int cu valoarea 1
ptrA = &a;           // atribuie pointerului ptrA adresa variabilei int a
ptrPtrA = & ptrA;    /* atribuie pointerului ptrPtrA adresa variabilei
                                        pointer la int ptrA */
```

În figură, variabila *a*, memorată începând cu adresa 0x22ccec, conține valoarea întreagă 88. Expresia *&a* returnează adresa variabilei *a*, adresă care este 0x22ccec. Această adresă este atribuită pointerului *ptrA*, ca valoare a sa inițială, iar pointerul *ptrPtrA* va avea ca valoare adresa pointerului *ptrA*!




## IB.07.3. 2 Indirectarea sau operația de dereferențiere(*)

Indirectarea printr-un pointer (diferit de *void* *), pentru acces la datele adresate de acel pointer, se face prin utilizarea operatorul unar *, *operator de dereferențiere (indirectare)*.

**Sintaxa:**

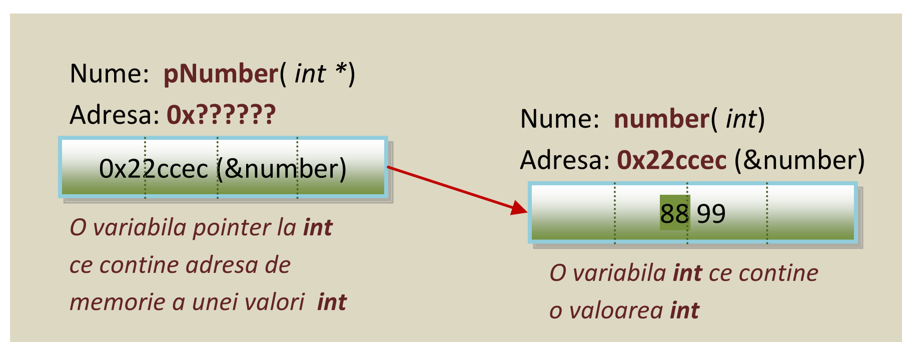
**Operatorul unar * - operator de dereferențiere (indirectare) - returnează valoarea păstrată la adresa indicată de pointer.**

*Exemple:*

```
int *p, m;  // p poinetr la int, m variabila int
m=*p;       // m ia valoarea indicata de pointerul p


int number = 88;
int *pNumber = &number;   /* Declara şi atribuie adresa variabilei number
             pointer-ului pNumber (acesta poate fi de exemplu 0x22ccec)*/
printf("%p\n", pNumber);   // Afiseaza pointerul (0x22ccec)
printf("%d\n", *pNumber);  /* Afiseaza valoarea indicata de pointer,
                                valoare care este de tip int (88)*/
*pNumber = 99;             /* Atribuie o valoare care va fi stocata la
            adresa indicata de pointer. Atentie! NU variabilei pointer!*/
printf("%d\n", *pNumber); /* Afiseaza noua valoare indicata de pointer,
                                                                99*/
printf("%d\n", number);    /* Valoarea variabilei number s-a schimbat de
                           asemenea (99)*/
```

pNumber stochează adresa unei locaţii de memorie; *pNumber se referă la valoarea păstrată la adresa indicată de pointer, sau altfel spus la valoarea indicată de pointer:




Putem spune că o variabilă face referire directă la o valoare, în timp ce un pointer face referire indirectă la o valoare, prin adresa de memorie pe care o stochează. Referirea unei valori în mod indirect printr-un pointer se numeşte **indirectare.**

**Observaţie:**

Simbolul * are înţelesuri diferite. Atunci când este folosit într-o declaraţie (int *pNumber), el denotă că numele care îi urmează este o variabilă de tip pointer. În timp ce, atunci când este folosit într-o expresie/instrucţiune (ex. *pNumber = 99;  printf("%d\n", *pNumber); ), se referă la valoarea indicată de variabila pointer.

### IB.07.3.3 Operaţia de atribuire

În partea dreaptă poate fi un pointer de acelaşi  tip (eventual cu conversie de tip), constanta *NULL* sau o expresie cu rezultat pointer.

*Exemple:*

```
int *p, *q=NULL;
float x=1.23;
p=q;
p=&x;
```

Unei variabile de tip *void** i se poate atribui orice alt tip de pointer fără conversie de tip explicită şi un argument formal de tip *void** poate fi înlocuit cu un argument efectiv de orice tip pointer. Atribuirea între alte tipuri pointer se poate face numai cu conversie de tip explicită (*cast*) şi permite interpretarea diferită a unor date din memorie. De exemplu, putem extrage cei doi octeţi dintr-un întreg scurt astfel:

```
short n;
char * p = (char*) &n;
c1= *p;
c2 = *(p+1);
```

### IB.07.3.4 Operaţii de comparaţie

**Compararea a doi pointeri (operaţii relaţionale cu pointeri) se poate face utilizând operatorii cunoscuţi:**
**==      !=      <      >      <=      >=**

### IB.07.3.5 Aritmetica pointerilor

Adunarea sau scăderea unui întreg la (din) un pointer, incrementarea şi decrementarea unui pointer se pot face astfel:

**Sintaxa:**
**p++; ↔ p=p+sizeof(tip);**
**p--; ↔ p=p-sizeof(tip);**
**p=p+c; ↔ p=p+c*sizeof(tip);**
**p=p-c; ↔ p=p-c*sizeof(tip);**

*Exemplu:*

```
void printVector( int a[], int n) {    // afişarea unui vector
   while (n--)
        printf ("%d ", *a++);
 }
```

Trebuie observat că incrementarea unui pointer şi adunarea unui întreg la un pointer nu adună întotdeauna întregul 1 la adresa conţinută în pointer; valoarea adăugată (scăzută) depinde de tipul variabilei pointer şi este egală cu produsul dintre constantă şi numărul de octeţi ocupat de tipul adresat de pointer.

Această convenţie permite referirea simplă la elemente succesive dintr-un vector folosind indirectarea printr-o variabilă pointer.

O altă operaţie este cea de **scădere** a două variabile pointer de acelaşi tip (de obicei adrese de elemente dintr-un acelaşi vector), obţinându-se astfel „distanţa" dintre două adrese, atenţie, nu în octeţi ci în blocuri de octeţi, în funcţie de tipul pointerului.
Exemplu de funcţie care întoarce indicele în şirul s1 a şirului s2 sau un număr negativ dacă s1 nu conţine pe s2:

```
int pos ( char* s1, char * s2) {
  char * p =strstr(s1,s2); //p va fi adresa la care se găseşte s2 in s1
  if (p) return p-s1;
  else  return -1;
}
```

### IB.07.3.6 Dimensiunea

Spaţiul ocupat de o variabilă pointer se determină utilizând operatorul *sizeof*: Valoarea expresiei este 2 (în modelul small); oricare ar fi tip_referit, expresiile de mai jos conduc la aceeaşi valoare 2:

```
sizeof( &var )
sizeof( tip_referit * )
```

### IB.07.3.7 Afişarea unui pointer

Tipărirea valorii unui pointer se face folosind funcţia *printf* cu formatul *%p*, valoarea apărând sub forma unui număr în hexa.

**Observaţii:**

- Adresa unei variabile pointer este un pointer, la fel ca adresa unei variabile de orice alt tip: &var_pointer
Un pointer este asociat cu un tip şi poate conţine doar o adresă de tipul specificat.

```
int i = 88;
double d = 55.66;
int *iPtr = &i;    // pointer int ce contine adresa unei variabile int
double *dPtr = &d; // pointer double ce indica spre o valoare double

iPtr = &d;      // EROARE, nu poate contine o adresa de alt tip
dPtr = &i;      // EROARE, nu poate contine o adresa de alt tip
iPtr = i;       /* EROARE, pointerul pastreaza adresa unui int,
                        NU o valoare int */

int j = 99;
iPtr = &j;      // putem schimba adresa continuta de un pointer
```

- O eroare frecventă este utilizarea unei variabile pointer care nu a primit o valoare (adică o adresă de memorie) prin atribuire sau prin iniţializare la declarare. Iniţializarea unui pointer se face prin atribuirea adresei unei variabile, prin alocare dinamică, sau ca rezultat al executării unei funcţii.
Compilatorul nu generează eroare sau avertizare pentru astfel de greşeli.
*Exemple incorecte:*

```
int * a;          // declarata dar neiniţializata    !!
scanf ("%d",a) ;  // citeşte la adresa conţinuta in variabila a
int *iPtr;        // declarata dar neiniţializata!!
*iPtr = 55;
print("%d\n",*iPtr);
```

Putem iniţializa un pointer cu valoarea 0 sau NULL, aceasta însemnând că nu indică nicio adresă – pointer *null*. Dereferenţierea unui pointer nul duce la o excepţie de genul STATUS_ACCESS_VIOLATION, şi un mesaj de eroare 'segmentation fault', eroare foarte des întâlnită!

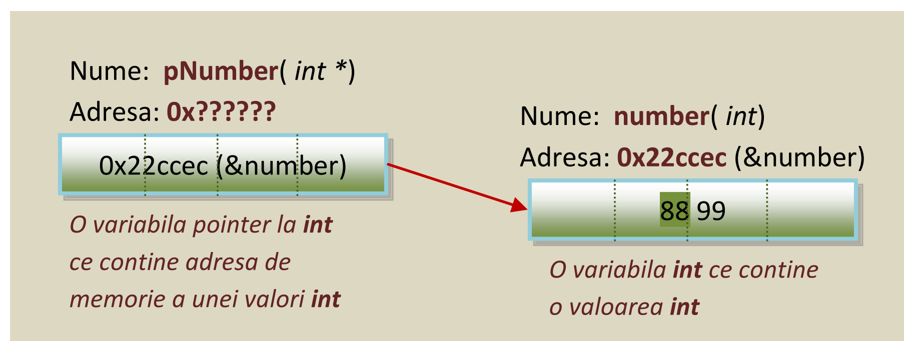
```
int *iPtr = 0;          // Declara un pointer int si-l initializeaza cu 0
print("%d\n",*iPtr);  // EROARE! STATUS_ACCESS_VIOLATION!
int *p = NULL;        // declara tot un pointer NULL
```

Inițializarea unui pointer cu NULL la declarare este o practică bună, deoarece elimină posibilitatea "uitării" inițializării cu o valoare validă!

- void * înseamnă un pointer de tip neprecizat şi utilizarea acestui tip de pointeri ne permite păstrarea gradului de generalitate al unui program la maximum.
- Atenție însă: Nu putem face operații aritmetice asupra acestor pointeri sau asupra pointerului nul.

*Exemplu:*

```
/* Test pentru declarare si utilizare pointeri */
int number = 88;          // number - intreg cu valoarea initiala 88
int *pNumber = &number;    /* Declara şi atribuie adresa variabilei number
        pointer-ului pNumber (acesta poate fi de exemplu 0x22ccec)*/
printf("%p\n", pNumber);  // Afiseaza pointerul (0x22ccec)
printf("%p\n", &number);  // Afiseaza adresa lui number (0x22ccec)
printf("%d\n", *pNumber); /* Afiseaza valoarea indicata de pointer,
                               valoare care este de tip int (88)*/
*pNumber = 99;            /* Atribuie o valoare care va fi stocata la
         adresa indicata de pointer. Atentie! NU variabilei pointer!*/
printf("%p\n", pNumber);  // Afiseaza pointerul (0x22ccec)
printf("%p\n", &number);  // Afiseaza adresa lui number (0x22ccec)
printf("%d\n", *pNumber);  // Afiseaza noua valoare indicata de pointer 99
printf("%d\n", number);   /*Valoarea variabilei number s-a schimbat de
                                                      asemenea (99)*/
printf("%p\n", &pNumber);  //Afiseaza adresa pointerului pNumber 0x22ccf0
```




Notă: Valoarea pe care o veți obține pentru adresă este foarte puțin probabil să fie cea din acest exemplu!

*Exemplu:*

```
int *p, n=5, m;
p=&n;
m=*p;       // m este 5
m=*p+1;     // m este 6

int *p;
float x=1.23, y;
p=&x;
y=*p;             // valoare eronata pentru y!

int *a,**b, c=1, d;
a=&c;
b=&a;
d=**b;            // d este 1
```

## IB.07.4 Vectori şi pointeri

**Convenţie!**
***Numele unui tablou* este un pointer constant spre primul element (index 0) din tablou.**

Cu alte cuvinte, o variabilă de tip tablou conţine adresa de început a acestuia (adresa primei componente) şi de aceea este echivalentă cu un pointer la tipul elementelor tabloului. Aceasta echivalenţă este utilizată de obicei în argumentele de tip tablou şi în lucrul cu tablouri alocate dinamic.
Expresiile de mai jos sunt deci echivalente:

**nume_tablou ⇔ &nume_tablou ⇔&nume_tablou[0]**
**şi:**
***nume_tablou ⇔ nume_tablou[0]**
***( nume_tablou +i) ⇔ nume_tablou [i]**

În concluzie, există următoarele echivalenţe de notaţie pentru un vector *a*:

| a[0] | *a |
|---|---|
| &a[0] | a |
| a[1] | *(a+1) |
| &a[1] | a+1 |
| a[k] | *(a+k) |
| &a[k] | a+k |

**Declaraţii echivalente pentru tablouri:**
**tip v [dim1] [dim2]…[dimn];**
**⇔**
**tip *…*v;**

*Exemple:*

```
int v[10]; // vector cu dimensiune fixă
                                                           int
*v=(int *)malloc(10*sizeof(int));        // vector alocat dinamic

// Referire elemente pentru ambele variante de declarare:
   v[i]            // sau:
   *(v+i)
```

```
int i;
double v[100], x, *p;
p=&v[0];          // corect, neelegant
p=v;
x=v[5];
x=*(v+5);
v++;              // incorect
p++;              //corect
```

```
Obs:
p[4]=2.5          //corect sintactic, dar nu e alocată memorie pentru p!!!
```

## IB.07.5 Transmiterea tablourilor ca argumente ale funcţiilor

Un tablou este trimis ca parametru unei funcţii folosind pointerul la primul element al tabloului. În declararea funcţiei putem folosi notaţia specifică tabloului (ex. int[]) sau notaţia specifică pointerilor (ex. int*). Compilatorul îl tratează întotdeauna ca pointer (ex. int*). De exemplu, pentru declararea unei funcţii care primeşte un vector de întregi şi dimensiunea lui avem următoarele declaraţii echivalente:

```
void printVec (int a [ ], int n);
void printVec (int * a, int n);
void printVec (int a [50 ], int n);
```

Ele vor fi tratate ca int* de către compilator. Dimensiunea din parantezele drepte este ignorată. Numărul de elemente din tablou trebuie trimis separat, sub forma unui al doilea parametru de tip *int*. Compilatorul nu va lua în calcul acest parametru ca dimensiune a tabloului şi ca urmare nu va verifica dacă această dimensiune se încadrează în limitele specificate (>0 şi <dim_maximă_declarată).
Elementele unui tablou pot fi modificate în funcţie, deoarece se transmite adresa acestuia – prin referinţă (vezi Transmiterea parametrilor – cap IB.06).

În interiorul funcţiei ne putem referi la elementele vectorului *a* fie prin indici (cu *a[i]*), fie prin indirectare (**(a+i)*), indiferent de felul cum a fost declarat vectorul *a*.

```
// prin indexare
void printVec (int a[ ], int n) {
   int i;
   for (i=0;i<n;i++)
      printf (%6d",a[i]);
}

// prin indirectare
void printVec (int *a, int n) {
   int i;
   for (i=0;i<n;i++)
      printf (%6d", *a++);
}
//Citirea elementelor unui vector se poate face asemănător:
for (i=0;i<n;i++)
   scanf ("%d", a+i);         // echivalent cu &a[i] şi cu a++
```

Apelul funcţiei se poate face astfel:

```
int main()
{
   int v[10], n;
   …
   printVec(v,n);
   …
}
      // SAU:
int main()
{
   int *v, n;     /* Atentie! Vectorul v nu are alocata memorie, va trebui
                     sa ii alocam dinamic!!*/
```

```
   …
   printVec(v,n);
    …
}
```

```
#include <stdio.h>
#include <stdlib.h>
#define N 5

int citire1(int tab[]){
  //citeste elementele lui tab prin accesarea indexata a elementelor
  int i=0;
      printf("Introduceti elementele tabloului:\n");
      while(scanf("%d",&tab[i]!=EOF) i++;
      return i;
}

void tiparire1(int *tab, int n){
  //tipareste elementele tabloului prin accesarea indexata a elementelor
      int i;
      printf("Elementele tabloului:\n");
      for(i=0;i<n;i++)
          printf("%d ",tab[i]);
      printf("\n");
}

int citire2(int tab[]){
  /* citeste elementele lui tab - accesarea fiecarui element se
  face printr-un pointer la el */
      int *pi;
      pi=tab;
      printf("Introduceti elementele tabloului:\n");
      while(scanf("%d",pi)!=EOF) pi++;
      return pi-tab;
}

void tiparire2 (int tab[], int n){
   // tipareste elementele lui tab prin accesare prin pointeri
       int *pi;
       printf("Elementele tabloului:\n");
       for (pi=tab; pi<tab+n; pi++)
         printf("%d ",*pi);
       printf("\n");
}

int main(){
  int tab1[N], tab2[N], n, m;
  n=citire1(tab1);
  tiparire1(tab1,n);
  m=citire2(tab2);
  tiparire2(tab2,m);
      return 0;
}
```

Program pentru citirea unui vector de întregi şi extragerea elementelor distincte într-un al doilea vector, care se va afisa. Se vor utiliza funcţii. Ce funcţii trebuie definite?

```
#define MAX 30
#include <stdio.h>
```

```
/* cauta pe x în vectorul a*/
int gasit(int *v, int n, int x){
   int m=0,i;
   for (i=0;i<n; i++)
      if (v[i]==x)  return i;
   return -1;
}

int main () {
   int a[MAX];           // un vector de intregi
   int b[MAX];           // aici se pun elementele distincte din a
   int n,m,i,j;          // n=dimensiune vector a, m=dimensiune vector b
   printf("Numar de elemente vector n="); scanf("%d",&n);
   printf ("Introducere %d numere intregi:\n",n);

   // citire vector:
   for (i=0;i<n;i++) scanf("%d",&a[i]);

   m=0;
   for (i=0;i<n;i++)
      if(gasit(b,m,a[i])==-1) b[m++]=a[i];

   // afiseaza elemente vector b:
   printf("Elementele distincte sunt:");
   for (j=0;j<m;j++)
      printf ("%5d",b[j]);

       return 0;
}
```

***Pentru declararea unei funcții care primeşte o matrice ca parametru*** avem următoarele posibilități:

- **int min( int t[][NMAX], int m, int n);**
- **int min( int *t [NMAX], int m, int n);**
- **int min( int **t, int m, int n);**

Apelul funcției se va face astfel:

```
int a[NMAX][NMAX], m, n;
…..
int mimim = min( a, m, n);
```

**Observații:**

- În aplicațiile numerice se preferă argumentele de tip vector şi adresarea cu indici, iar în funcțiile cu şiruri de caractere se preferă argumente de tip pointer şi adresarea indirectă prin pointeri.
- O funcție poate avea ca rezultat un pointer dar nu şi rezultat vector.
- Diferența majoră dintre o variabilă pointer şi un nume de vector este aceea că un nume de vector este un pointer constant (adresa este alocată de compilatorul C şi nu mai poate fi modificată la execuție). Un nume de vector nu poate apare în stânga unei atribuiri, în timp ce o variabilă pointer are un conținut modificabil prin atribuire sau prin operații aritmetice.

*Exemple:*

```
int a[100], *p;
p=a; ++p;                          // corect
a=p; ++a;                          // ambele instrucțiuni produc erori
```

- Când un nume de vector este folosit ca argument, se transmite un pointer cu aceeaşi valoare ca numele vectorului, iar funcţia poate folosi argumentul formal în stânga unei atribuiri.
- Declararea unui vector (alocat la compilare) nu este echivalentă cu declararea unui pointer, deoarece o declaraţie de vector alocă memorie şi iniţializează pointerul ce reprezintă numele vectorului cu adresa zonei alocate (operaţii care nu au loc automat la declararea unui pointer).

```
int * a;  a[0]=1;       // greşit !
int *a={3,4,5};         // echivalent cu: int a[]={3,4,5}
```

- Operatorul *sizeof* aplicat unui nume de vector cu dimensiune fixă are ca rezultat numărul total de octeţi ocupaţi de vector, dar aplicat unui argument formal de tip vector (sau unui pointer la un vector alocat dinamic) are ca rezultat mărimea unui pointer:

```
float x[10];
float * y=(float*)malloc (10*sizeof(float));  /*vector caruia i s-a
                     alocat dinamic memorie pentru 10 numere float */
printf ("%d,%d \n",sizeof(x), sizeof(y));           // scrie 40, 4
```

- Numărul de elemente dintr-un vector alocat la compilare sau iniţializat cu un şir de valori se poate afla prin expresia: sizeof (x) / sizeof(x[0]).

```
int x[10];
printf ("%d\n",sizeof(x));               // scrie 40
printf ("%d\n",sizeof(x[0]));            // scrie 4
printf ("%d\n",sizeof(x)/sizeof(x[0]); // scrie 10
```

## IB.07.6 Pointeri în funcţii

Reamintim că în C/C++, parametrii efectivi sunt transmişi prin valoare - valorile parametrilor actuali sunt depuse pe stivă, fiind apoi prelucrate ca parametri formali de către funcţie. Ca urmare, modificarea valorii lor de către funcţie nu este vizibilă în exterior! Un exemplu clasic este o funcţie care *încearcă* să schimbe între ele valorile a două variabile, primite ca argumente:

```
void swap (int a, int b) {
   int aux;
   aux=a;
   a=b;
   b=aux;
}

int main () {
   int x=3, y=7;
   swap(x,y);
   printf ("%d,%d \n", x, y);       // scrie 3, 7  nu e ceea ce ne doream!
   return 0;
}
```

Pentru a înţelege mai bine mecanismul transmiterii parametrilor prin valoare oferim următoarea detaliere a paşilor efectuaţi în cazul funcţiei de interschimbarea a valorilor a două variabile:

....

În multe situații, se doreşte modificarea parametrilor unei funcții. Acest lucru poate fi realizat prin transmiterea ca parametru al funcției a unui pointer la obiectul a cărui valoare vrem să o modificăm, modalitate cunoscută sub numele de transmitere prin referință.

**Observație:** Se pot modifica valorile de la adresele trimise ca parametri! Nu se pot modifica adresele trimise!

**O funcție care:**
- **trebuie să modifice mai multe valori primite prin argumente, sau**
- **care trebuie să transmită mai multe rezultate calculate de funcție**

**trebuie să folosească argumente de tip pointer.**

Versiunea corectă pentru funcția *swap* este următoarea:

```
void swap (int * pa, int * pb) {          // pointeri la intregi
   int aux;
   aux=*pa;
   *pa=*pb;
   *pb=aux;// Adresare indirecta pt a accesa valoarile de la adresele pa, pb
}

// apelul acestei funcţii foloseşte argumente efective pointeri:

int main(void)
{
   int x=5, y=7;
   swap(&x, &y);
       //transmitere prin adresă
   printf("%d %d\n", x, y);
   /*valorile sunt inversate adică se va afişa 7 5*/
```

```
   return 0;
}
```

*Exemple:*

```
/* calcul nr2   */
#include <stdio.h>

void square(int *pNr) {
   *pNr *= *pNr; //Adresare indirecta pt a accesa valoarea de la adresa pNr
}

int main() {
   int nr = 8;
   printf("%d\n", nr); // 8
   square(&number);    // transmitere prin referinta explicita - pointer
   printf("%d\n", nr); // 64
   return 0;
}
```

Pentru a înţelege mai bine mecanismul transmiterii parametrilor prin pointeri oferim următoarea detaliere a paşilor efectuaţi în cazul funcţiei de interschimbarea a valorilor a două variabile:

...

**Observaţii:**

- O funcţie care primeşte două sau mai multe numere pe care trebuie să le modifice va avea argumente de tip pointer sau un argument vector care reuneşte toate rezultatele (datele modificate).

- Dacă parametrul este un tablou, funcția poate modifica valorile elementelor tabloului, primind adresa tabloului. A se observa că trebuie să se transmită ca parametri şi dimensiunea/dimensiunile vectorului/matricii. Dacă parametrul este şir de caractere, dimensiunea vectorului de caractere nu trebuie să se transmită, sfârşitul şirului fiind indicat de caracterul terminator de şir, *0*!
- O funcție poate avea ca rezultat un pointer, dar acest pointer nu trebuie să conțină adresa unei variabile locale, deoarece o variabilă locală are o existență temporară, garantată numai pe durata executării funcției în care este definită (cu excepția variabilelor locale statice) şi de aceea adresa unei astfel de variabile nu trebuie transmisă în afara funcției, pentru a fi folosită ulterior. Un rezultat pointer este egal cu unul din argumente, eventual modificat în funcție, fie o adresă obținută prin alocare dinamică ( care rămâne valabilă si după terminarea funcției). Pentru detalii a se vedea **IB.10.**
  Exemplu corect:

```
int *plus_zece( int *a ) {
        *a=*a+10;
   return a;
}
```

  Exemplu de programare greşită:

```
// Vector cu cifrele unui nr intreg
int *cifre (int n) {
      int k , c[5]; // Vector local
      for (k=4; k>=0; k--) {
            c[k]=n%10;
            n=n/10;
      }
      return c; // Gresit
}
```

- Pointerii permit:
  - să realizăm modificarea valorilor unor variabile transmise ca parametri unei funcții;
  - să accesăm mult mai eficient tablourile;
  - să lucrăm cu zone de memorie alocate dinamic;
  - să se acceseze indirect o valoare a unui tip de date.

*Exemple:*
*Program pentru determinarea elementelor minim şi maxim dintr-un vector într-o aceeaşi funcţie. Funcţia nu are tip (void)!*

```
#include<stdio.h>
void minmax ( float x[], int n, float* pmin, float* pmax) {
        float xmin, xmax;
        int i;
        xmin=xmax=x[0];
        for (i=1;i<n;i++) {
            if (xmin > x[i])  xmin=x[i];
            if (xmax < x[i]) xmax=x[i];
         }
         *pmin=xmin;
         *pmax=xmax;
      }

// utilizare funcţie
```

```
int main () {
        float a[]={3,7,1,2,8,4};
        float a1,a2;
        minmax (a,6,&a1,&a2);
        printf("%f %f \n",a1,a2);
        getchar();
        return 0;
}

// NU!!!
int main () {
        float a[]={3,7,1,2,8,4};
        float *a1, *a2; // pointeri neinitializati !!!
        minmax (a,6,a1,a2);
        printf("%f %f \n",*a1,*a2);
        getchar();
        return 0;
}
```

Să se scrie o funcție care calculează valorile unghiurilor unui triunghi, în funcție de lungimile laturilor. Funcția va fi scrisă în două variante:

- cu 6 argumente: 3 date şi 3 rezultate
- cu 2 argumente de tip vector.

```
//varianta 1 cu 6 argumente: 3 date şi 3 rezultate

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

void unghiuri(float ab, float ac, float bc, float *a, float *b, float *c)
{
   *a=acos((ab*ab+ac*ac-bc*bc)/(2*ab*ac))*180/M_PI;
   *b=acos((ab*ab+bc*bc-ac*ac)/(2*ab*bc))*180/ M_PI;
   *c=acos((bc*bc+ac*ac-ab*ab)/(2*bc*ac))*180/ M_PI;
}

int main(){
   float a, b, c, ab, ac, bc;
   scanf("%f%f%f", &ab, &ac, &bc);
   unghiuri(ab, ac ,bc, &a ,&b, &c);
   printf("%f %f %f" , a, b, c);
   return 0;
}

//varianta 2 cu 2 argumente de tip vector

#include <stdio.h>
#include <math.h>
void unghiuri(float *L , float *U )
{
   U[0]=acos((L[0]*L[0]+L[1]*L[1]-L[2]*L[2])/(2*L[0]*L[1]))*180/ M_PI;
   U[1]=acos((L[0]*L[0]+L[2]*L[2]-L[1]*L[1])/(2*L[0]*L[2]))*180/ M_PI;
   U[2]=acos((L[2]*L[2]+L[1]*L[1]-L[0]*L[0])/(2*L[2]*L[1]))*180/ M_PI;
}

int main(){
   float L[2],U[2];
   int i;
   for (i=0;i<=2;i++)
       scanf("%f",&L[i]);

   unghiuri(L,U);
```

```
   for (i=0;i<=2;i++)
       printf("%f ",U[i]);
   return 0;
}
```

## IB.07.7 Pointeri la funcții

Anumite aplicații numerice necesită scrierea unei funcții care să poată apela o funcție cu nume necunoscut, dar cu prototip şi efect cunoscut. De exemplu, o funcție care:

- să sorteze un vector ştiind funcția de comparare a două elemente ale unui vector
- să calculeze integrala definită a oricărei funcții cu un singur argument
- să determine o rădăcină reală a oricărei ecuații (neliniare).

Aici vom lua ca exemplu o funcție *listf* care poate afişa (lista) valorile unei alte funcții cu un singur argument, într-un interval dat şi cu un pas dat. Exemple de utilizare a funcției *listf* pentru afişarea valorilor unor funcții de bibliotecă:

```
int main () {
   listf (sin,0.,2.*M_PI, M_PI/10.);
   listf (exp,1.,20.,1.);
   return 0;
}
```

Problemele apar la definirea unei astfel de funcții, care primeşte ca argument numele (adresa) unei funcții.

**Convenție C:**
**Numele unei funcții neînsoțit de o listă de argumente este adresa de început a codului funcției şi este interpretat ca un pointer către funcția respectivă.**

Deci *sin* este adresa funcției *sin(x)* în apelul funcției *listf*.

Declararea unui parametru formal (sau unei variabile) de tip pointer la o funcție are forma următoare:

**Sintaxa:**

**tip_returnat (*pf) (lista_param_formali);**

**unde:**

- ***pf*** **este numele paramatrului (variabilei) pointer la funcție,**
- ***tip_returnat*** **este tipul rezultatului funcției.**
- ***lista_param_formali*** **include doar tipurile parametrilor.**

**Observație:**
Parantezele sunt importante, deoarece absența lor modifică interpretarea declarației. Astfel, declarația:

```
tip * f (lista_param_formali)
```

introduce o funcție cu rezultat pointer, nu un pointer la funcție!!

De aceea, o eroare de programare care trece de compilare şi se manifestă la execuție, este apelarea unei funcții fără paranteze; compilatorul nu apelează funcția şi consideră că programatorul vrea să folosească adresa funcției!

*Exemplu:*

```
if ( test ) break;      /* gresit, echiv. cu if (1) break; deoarece e luat
                        in calcul pointerul la functia test */

if ( test() ) break;    // aici se testeaza valoarea intoarsa de functie
```

În concluzie, definirea funcției *listf* este:

```
void listf (double (*fp)(double), double min, double max, double pas) {
   double x,y;
   for (x=min; x<=max; x=x+pas) {
     y=(*fp)(x);                                     // sau:  y=fp(x);
     printf ("\n%20.10lf %20.10lf", x,y);
   }
}
```

Pentru a face programele mai explicite se pot defini nume de tipuri pentru tipuri pointeri la funcții, folosind declarația ***typedef***.

```
typedef double (* ftype) (double);
void listf (ftype fp, double min, double max, double pas) {
   double x, y;
   for (x=min; x<=max; x=x+pas) {
             y = fp(x);
             printf ("\n%20.10lf %20.10lf", x,y);
   }
}
```

*Exemple:*

1. Program cu meniu de opțiuni; operatorul alege una dintre funcțiile realizate de programul respectiv.

```
#include<stdio.h>
#include<stdio.h>
typedef void (*funPtr) (); /* defineste tipul funPtr, care este pointer la
o functie de tip void fara argument */

// funcții pentru operatii realizate de program
void  unu () {
   printf ("unu\n");
}

void  doi () {
   printf ("doi\n");
}

void  trei () {
   printf ("trei\n");
}

// selectare şi apel funcție
int main () {
  funPtr tp[ ]= {unu,doi,trei};     // vector de pointeri la funcții
  short option=0;
  do{
     printf("Optiune (1/2/3):");
     scanf ("%hd", &option);
     if (option >=1 && option <=3)
                 tp[option-1]();         // apel funcție (unu/doi/trei)
             else break;
  }while (1);
```

```
  return 0;
}

//Secvența echivalentă, fara a folosi pointeri la functii, este:
do {
   printf("Optiune (1/2/3):");
   scanf ("%hd", &option);
   switch (option) {
     case 1: unu(); break;
     case 2: doi(); break;
     case 3: trei(); break;
   }
} while (1);
```

2. Program pentru operații aritmetice între numere întregi (doar adunare şi scădere, se poate completa):

```
/* Test pointeri la functii (TestFunctionPointer.cpp) */
#include <stdio.h>

int aritmetica(int, int, int (*)(int, int));
    /* int (*)(int, int) este un pointer la o functie,
      care primeste doi intregi si returneaza un intreg */
int add(int, int);
int sub(int, int);

int add(int n1, int n2) { return n1 + n2; }
int sub(int n1, int n2) { return n1 - n2; }

int aritmetica(int n1, int n2, int (*operation) (int, int)) {
   return (*operation)(n1, n2);
}

int main() {
   int number1 = 5, number2 = 6;

   // adunare
   printf("%d\n", aritmetica(nr1, nr2, add));
   // scadere
   printf("%d\n", aritmetica(nr1, nr2, sub));
   return 0;
}
```

## IB.07.8 Funcții generice

În fişierul *stdlib.h* sunt declarate patru funcții generice pentru sortarea, căutarea liniară şi căutarea binară într-un vector cu componente de orice tip, care ilustrează o modalitate simplă de generalizare a tipului unui vector. Argumentul formal de tip vector al acestor funcții este declarat ca *void** şi este înlocuit cu un argument efectiv pointer la un tip precizat (nume de vector). Un alt argument al acestor funcții este adresa unei funcții de comparare a unor date de tipul celor memorate în vector, funcție furnizată de utilizator şi care depinde de datele folosite în aplicația sa.
Pentru exemplificare urmează declarațiile pentru trei din aceste funcții ("lfind" este la fel cu "lsearch"):

**void *bsearch (const void *key, const void *base, size_t nelem, size_t width, int (*fcmp)(const void*, const void*));**

**void *lsearch (const void *key, void *base, size_t * pnelem, size_t width, int (*fcmp)(const void *, const void *));**

**void qsort (void *base, size_t nelem, size_t width, int (*fcmp)(const void *, const void *));**

- *base* este adresa vectorului
- *key* este cheia (valoarea) căutată în vector (de acelaşi tip cu elementele din vector)
- *width* este dimensiunea unui element din vector (ca număr de octeți)
- *nelem* este numărul de elemente din vector
- *fcmp* este adresa funcției de comparare a două elemente din vector.

Exemplul următor arată cum se poate ordona un vector de numere întregi cu funcția *qsort* :

```
// functie pentru compararea a doua numere intregi
int intcmp (const void * a, const void * b) {
  return *(int*)a-*(int*)b;
}

int main () {
   int a[]= {5,2,9,7,1,6,3,8,4};
   int i, n=9;
             // n=dimensiune vector
   qsort ( a,9, sizeof(int),intcmp);          // ordonare vector
   for (i=0;i<n;i++)                          // afişare rezultat
     printf("%d ",a[i]);
}
```

## IB.07.9 Anexă. Tipul referință în C++

În C++ s-a introdus tipul referință, folosit în primul rând pentru parametri modificabili sau de dimensiuni mari, dar şi funcțiile care au ca rezultat un obiect mare pot fi declarate de un tip referință, pentru a obține un cod mai performant.

**Sintaxa:**

**Caracterul ampersand (&) folosit după tipul şi înaintea numelui unui parametru formal sau al unei funcții arată compilatorului că pentru acel parametru se primeşte adresa şi nu valoarea argumentului efectiv.**

Spre deosebire de un parametru pointer, un parametru referință este folosit de utilizator în interiorul funcției la fel ca un parametru transmis prin valoare, dar compilatorul va genera automat indirectarea prin pointerul transmis (în programul sursă nu se foloseşte explicit operatorul de indirectare *).

*Exemplu:*

```
void schimb (int & x, int & y) {    // schimba intre ele doua valori
   int t = x;
   x = y;
   y = t;
}
void sort ( int a[], int n ) {                   // ordonare vector
   ...
   if ( a[i] > a[i+1])  schimb ( a[i], a[i+1]);
   ...
}
```

Sintaxa declarării unui tip referinţă este următoarea:

**Sintaxa:**

**tip & nume**

**unde nume poate fi:**

- **numele unui parametru formal**
- **numele unei funcţii (urmat de lista argumentelor formale)**
- **numele unei variabile (mai rar).**

Efectul caracterului *&* în declaraţia anterioară este următorul: compilatorul creează o variabilă *nume* şi o variabilă pointer la variabila *nume*, iniţializează variabila pointer cu adresa asociată lui *nume* şi reţine că orice referire ulterioară la *nume* va fi tradusă într-o indirectare prin variabila pointer anonimă creată.

O funcţie poate avea ca rezultat o referinţă la un vector dar nu poate avea ca rezultat un vector. O funcţie nu poate avea ca rezultat o referinţă la o variabilă locală, aşa cum nu poate avea ca rezultat un pointer la o variabila locală.

Referinţele simplifică utilizarea unor parametri modificabili de tip pointer, eliminând necesitatea unui pointer la pointer.
Exemplu de funcţie care primeşte adresa unui şir şi are ca rezultat adresa primei litere din acel şir:

```
void skip (char * & p){
   while ( ! isalpha(*p))
      p++;
}
```

Parametrul efectiv transmis unei funcţii pentru un parametru referinţă trebuie să fie un pointer modificabil şi deci nu poate fi numele unui vector alocat la compilare:

```
int main () {
    char s[]=" 2+ana -beta "; // un sir
    char *p=s;           // nu se poate scrie:  skip(s);
    skip(p);
    puts(p);
       return 0;
}
```

Există riscul modificării nedorite, din neatenţie, a unor parametri referinţă, situaţie ce poate fi evitată prin copierea lor în variabile locale ale funcţiei.