POINTERI

5

5.1.Variabile pointer
5.1.1. Declararea variabilelor pointer
5.1.2. Iniţializarea variabilelor pointer
5.1.3. Pointeri generici
5.2. Operaţii cu pointeri
5.3. Pointeri şi tablouri
5.3.1. Pointeri şi şiruri de caractere
5.3.2. Pointeri şi tablouri bidimensionale
5.4. Tablouri de pointeri
5.5. Pointeri la pointeri
5.6. Modificatorul const în declararea pointerilor

5.1. VARIABILE POINTER

Pointerii sunt variabile care au ca valori sunt adresele altor variabile (obiecte). Variabila este un nume simbolic utilizat pentru un grup de locaţii de memorie. Valoarea memorată într-o variabilă pointer este o *adresă*.
Din punctul de vedere al conţinutului zonei de memorie adresate, se disting următoarele categorii de pointeri:

- pointeri ***de date (obiecte)*** - conţin adresa unei variabile din memorie;
- pointeri ***generici (numiţi şi pointeri void)*** - conţin adresa unui obiect oarecare, de tip neprecizat;
- pointeri ***de funcţii*** (prezentaţi în capitolul 6.11.)- conţin adresa codului executabil al unei funcţii.

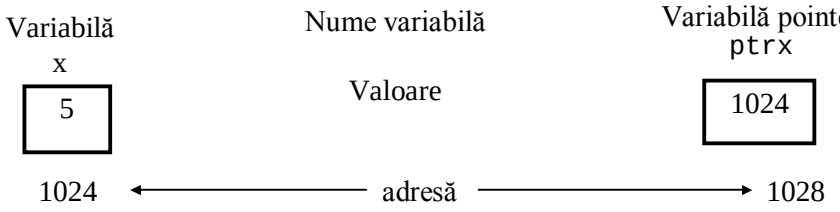


Figura 5.1. Variabile pointer

În figura 5.1, variabila `x` este memorată la adresa 1024 şi are valoarea 5. Variabila `ptrx` este memorată la adresa de memorie 1028 şi are valoarea 1024 (adresa variabilei x). Vom spune că `ptrx` pointează către `x`, deoarece valoarea variabilei `ptrx` este chiar adresa de memorie a variabilei `x`.

5.1.1. DECLARAREA VARIABILELOR POINTER

Sintaxa ***declaraţiei unui pointer de date*** este:

```
tip *identificator_pointer;
```

Simbolul `*` precizează că `identificator_pointer` este numele unei variabile pointer de date, iar `tip` este tipul obiectelor a căror adresă o va conţine.

Exemplu:

```
int u, v, *p, *q;            // *p, *q sunt pointeri de date (către int)
double a, b, *p1, *q1;       // *p1, *q1 sunt pointeri către date de tip double
```

Pentru ***pointerii generici***, se foloseşte declaraţia:

```
void *identificator_pointer;
```

Exemplu:

```
void *m;
```

Aceasta permite declararea unui pointer generic, care nu are asociat un tip de date precis. Din acest motiv, în cazul unui pointer vid, dimensiunea zonei de memorie adresate şi interpretarea informaţiei nu sunt definite, iar proprietăţile diferă de ale pointerilor de date.

5.1.2. INIŢIALIZAEA VARIABILELOR POINTER

Există doi ***operatori unari*** care permit utilizarea variabilelor pointer:
- **&** - *operatorul adresă (de referențiere)* - pentru aflarea adresei din memorie a unei variabile;
- * - *operatorul de indirectare (de deferențiere)* - care furnizează valoarea din zona de memorie spre care pointează pointerul operand.

În exemplul prezentat în figura 5.1, pentru variabila întreagă x, expresia **&x** furnizează *adresa variabilei* x. Pentru variabila pointer de obiecte int, numită ptr, expresia ***ptr** înseamnă *conținutul locației de memorie a cărei adresă este memorată în variabila* ptr. Expresia *ptr poate fi folosită atât pentru aflarea valorii obiectului spre care pointează ptr, cât şi pentru modificarea acesteia (printr-o operație de atribuire).

Exemplu:
```
int x, y, *ptr;
// ptr- variabilă pointer către un int; x,y-variabile predefinite, simple, de tip int
x=5; cout<<"Adresa variabilei x este:"<<&x<<'\n';
cout<<"Valoarea lui x:"<<x<<'\n';
ptr=&x;                 // atribuire: variabila ptr conține adresa variabilei x
cout<<"Variabila pointer ptr are valoarea:"<<ptr;
cout<<" si adreseaza obiectul:"<< *ptr<<'\n';
y=*ptr; cout<<"y="<<y<<'\n';  // y=5
x=4; cout<<"x="<<x<<'\n'; cout<<"*ptr="<<*ptr<<'\n';
// x si *ptr reprezinta acelasi obiect, un intreg cu valoarea 4
x=70;                   // echivalenta cu *ptr=70;
y=x+10;                 // echivalenta cu y=*ptr+10
```

În exemplul anterior, atribuirea **ptr=&x** se execută astfel: operatorul & furnizează adresa lui x; operatorul = atribuie valoarea (care este o adresă) variabilei pointer ptr.
Atribuirea **y=*ptr** se realizează astfel: operatorul * accesează conținutul locației a cărei adresă este conținută în variabila ptr; operatorul **=** atribuie valoarea variabilei y.

Declarația **int *ptr;** poate fi, deci, interpretată în două moduri, ambele corecte:
- **ptr** este de tipul **int *** (ptr este de tip pointer spre int)
- ***ptr** este de tipul **int** (conținutul locației spre care pointează variabila ptr este de tipul int)

Construcția ***tip** *** este de tipul pointer către int.
Atribuirea **x=8;** este echivalentă cu **ptr=&x; *p=x;**

Variabilele pointer, alături de operatorii de referențiere şi de deferențiere, pot apare în expresii.
Exemple:
```
int x, y, *q; q=&x;
*q=8;          // echivalentă cu x=8;
q=&5;          // invalidă - constantele nu au adresă
*x=9;          // invalidă - x nu este variabilă pointer
x=&y;  //invalidă: x nu este variabilă pointer, deci nu poate fi folosită cu operatorul de indirectare
y=*q + 3;      // echivalentă cu y=x+3;
*q = 0;        // setează x pe 0
*q += 1;       // echivalentă cu (*q)++ sau cu x++
int *r; r = q;
/* copiază conținutul lui q (adresa lui x) în r, deci r va pointa tot către x (va conține tot adresa lui x)*/
double w, *r = &w, *r1, *r2; r1= &w; r2=r1;
cout<<"r1="<<r1<<'\n';          //afişează valoarea pointerului r1 (adresa lui w)
cout<<"&r1="<<&r1<<'\n';        // afişează adresa variabilei r1
cout<<"*r1= "<<*r1<<'\n';
double z=*r1;                   // echivalentă cu z=w
cout<<"z="<<z<<'\n';
```

5.1.3. POINTERI GENERICI

La declararea pointerilor generici (***void * nume;***) nu se specifică un tip, deci unui pointer void i se pot atribui adrese de memorie care pot conține date de diferite tipuri: int, float, char, etc. Aceşti pointeri pot fi folosiți cu mai multe tipuri de date, de aceea este necesară folosirea ***conversiilor explicite*** prin expresii de tip cast, pentru a preciza tipul datei spre care pointează la un moment dat pointerul generic.

Exemplu:

```
void *v1, *v2; int a, b, *q1, *q2;
q1 = &a; q2 = q1; v1 = q1;
q2 = v1;     // eroare: unui pointer cu tip nu i se poate atribui un pointer generic
q2 = (int *) v1; double s, *ps = &s;
int c, *l; void *sv;
l = (int *) sv; ps = (double *) sv;
*(char *) sv = 'a';     /*Interpretare: adresa la care se găseşte valoarea lui sv este interpretată ca fiind adresa zonei de memorie care conține o data de tip char. */
```

Pe baza exemplului anterior, se pot face observațiile:
1. Conversia tipului pointer generic spre un tip concret înseamnă, de fapt, precizarea tipului de pointer pe care îl are valoarea pointerului la care se aplică conversia respectivă.
2. Conversia tipului pointer generic asigură o flexibilitate mai mare în utilizarea pointerilor.
3. Utilizarea în mod abuziv a pointerilor generici poate constitui o sursă de erori.

5.2. OPERAȚII CU POINTERI

În afara operației de ***atribuire*** (prezentată în paragraful 5.1.2.), asupra variabilelor pointer se pot realiza operații de ***comparare***, ***adunare*** şi ***scădere*** (inclusiv *incrementare* şi *decrementare*).

- Compararea valorilor variabilelor pointer
  Valorile a doi pointeri pot fi ***comparate***, folosind *operatorii relaționali*, ca în exemplul:

  Exemplu:

```
int *p1, *p2;
if (p1<p2)
   cout<<"p1="<<p1<<"<"<<"p2="<<p2<<'\n';
else cout<<"p1="<<p1<<">="<<"p2="<<p2<<'\n';
```

  O operație uzuală este *compararea unui pointer cu valoarea nulă*, pentru a verifica *dacă acesta adresează un obiect*. Compararea se face cu constanta simbolică `NULL` (definită în header-ul `stdio.h`) sau cu valoarea 0.

  Exemplu:

```
if (!p1)              // sau if (p1 != NULL)
   . . . . . ;        // pointer nul
else . . . .  ;       // pointer nenul
```

- Adunarea sau scăderea
  Sunt permise operații de ***adunare*** sau ***scădere*** *între un pointer de obiecte şi un întreg*:
  Astfel, dacă *ptr* este un pointer către tipul *tip* (`tip *ptr;`), iar `n` este un întreg, expresiile

  `ptr + n` `şi` ***`ptr - n`***

  au ca valoare, valoarea lui `ptr` la care se adaugă, respectiv, se scade ***`n * sizeof(tip)`***.
  Un caz particular al adunării sau scăderii dintre un pointer de date şi un întreg (`n=1`) îl reprezintă incrementarea şi decrementarea unui pointer de date. În expresiile `ptr++`, respectiv `ptr--`, valoarea variabilei ptr devine ***`ptr+sizeof(tip)`***, respectiv, ***`ptr-sizeof(tip).`***
  Este permisă *scăderea a doi pointeri de obiecte de acelaşi tip*, rezultatul fiind o valoare întreagă care reprezintă diferența de adrese divizată prin dimensiunea tipului de bază.

Exemplu:
```
int a, *pa, *pb;
cout<<"&a="<<&a<<'\n'; pa=&a; cout<<"pa="<<pa<<'\n';
cout<<"pa+2"<<pa+2<<'\n'; pb=pa++; cout<<"pb="<<pb<<'\n';
int i=pa-pb; cout<<"i="<<i<<'\n';
```

5.3. POINTERI ŞI TABLOURI

În limbajele C/C++ există o strânsă legătură între pointeri şi tablouri, deoarece *numele unui tablou* este un *pointer* (constant!) care are ca valoare adresa primului element din tablou. Diferența dintre numele unui tablou şi o variabilă pointer este aceea că unei variabile de tip pointer i se pot atribui valori la execuție, lucru imposibil pentru numele unui tablou. Acesta are tot timpul, ca valoare, adresa primului său element. De aceea, se spune că *numele unui tablou* este un *pointer constant* (valoarea lui nu poate fi schimbată). *Numele unui tablou* este considerat ca fiind un ***rvalue*** (right value-valoare dreapta), deci nu poate apare decât în partea dreaptă a unei expresii de atribuire. *Numele unui pointer* (în exemplul următor, *ptr) este considerat ca fiind un ***lvalue*** (left value-valoare stânga), deci poate fi folosit atât pentru a obține valoarea obiectului, cât şi pentru a o modifica printr-o operație de atribuire.

Exemplu:
```
int a[10], *ptr;          // a este definit ca &a[0]; a este pointer constant
a = a + 1;                // ilegal
ptr = a ;          // legal: ptr are aceeaşi valoare ca şi a, respectiv adresa elementului a[0]
                   // ptr este variabilă pointer, a este constantă pointer.
int x = a[0];      // echivalent cu x = *ptr; se atribuie lui x valoarea lui a[0]
```

Deoarece numele tabloului a este sinonim pentru adresa elementului de indice zero din tablou, asignarea `ptr=&a[0]` poate fi înlocuită, ca în exemplul anterior, cu `ptr=a`.

5.3.1. POINTERI ŞI ŞIRURI DE CARACTERE

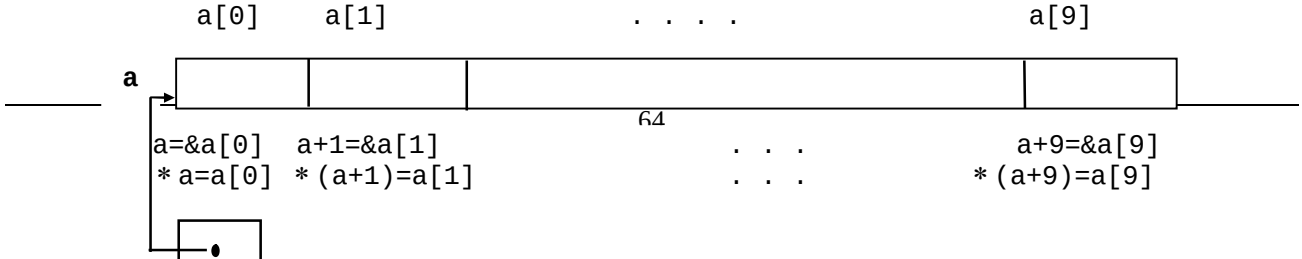
Aşa cum s-a arătat în capitolul 4, un şir de caractere poate fi memorat într-un vector de caractere. Spre deosebire de celelalte constante, constantele şir de caractere nu au o lungime fixă, deci numărul de octeți alocați la compilare pentru memorarea şirului, variază. Deoarece valoarea variabilelor pointer poate fi schimbată în orice moment, cu multă uşurință, este preferabilă utilizarea acestora, în locul tablourilor de caractere (vezi exemplul următor).

Exemplu:
```
char sir[10];     char *psir;
sir = "hello";    // ilegal
psir = "hello";   // legal
```
Operația de indexare a elementelor unui tablou poate fi realizată cu ajutorul variabilelor pointer.

Exemplu:
```
int a[10], *ptr;  // a este pointer constant; ptr este variabilă pointer
ptr = a;          // ptr este adresa lui a[0]
```
`ptr+i` înseamnă `ptr+(i*sizeof(int))`, deci: `ptr + i ⇔ & a[i]`

Deoarece numele unui tablou este un pointer (constant), putem concluziona (figura 5.2):

`a+i ⇔ & a[i]`
`a[i] ⇔ *(a+i)`



Exerciţiu: Să se scrie următorul program (care ilustrează legătura dintre pointeri şi vectori) şi să se urmărească rezultatele execuţiei acestuia.

```
#include <iostream.h>
void main(void)
{int a[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }; int  *pi1 = a ;
int  *pi2 = &a[0]; int  *pi3;
cout<<"a="<<a<<"&a="<<&a<<"*a="<<*a<<'\n';
cout<<"a+1="<<(a+1)<< " &a[1]="<< &a[1]<<'\n';
cout<<"a[1]="<<a[1]<< " *(a+1)="<< *(a+1)<<'\n';
cout<<"pi1="<<pi1<<"pi2="<<pi2<<'\n'; int x=*pi1;
/* x primeşte valoarea locaţiei a carei adresă se află în variabila pointer pi1, deci valoarea lui a[0] */
cout<<"x="<<x<<'\n'; x=*pi1++;              // echivalent cu *(pi1++)  x=1
cout<<"x="<<x<<'\n'; x=(*pi1)++;
/* x=0: întâi atribuirea, apoi incrementarea valorii spre care pointează pi1. În urma incrementării,
valoarea lui a[0] devine 1 */
cout<<"x="<<x<<'\n'; cout<<*pi1<<'\n';x=*++pi1;        //echivalent cu *(++pi1)
cout<<"x="<<x<<'\n'; x=++(*pi1); cout<<"x="<<x<<'\n'; pi1=a;
pi3=pi1+3;
cout<<"pi1="<<pi1<<"*pi1="<<*pi1<<"&pi1="<<&pi1<<'\n';
cout<<"pi3="<<pi3<<"*pi3="<<*pi3<<"&pi3="<<&pi3<<'\n';
cout<<"pi3-pi1="<<(pi3-pi1)<<'\n';  //pi3-pi1=3
}
```

Exerciţiu: Să se scrie următorul program (legătura pointeri-şiruri de caractere) şi să se urmărească rezultatele execuţiei acestuia.

```
#include <iostream.h>
void main(void)
{int a=-5, b=12, *pi=&a; double u=7.13, v=-2.24, *pd=&v;
char sir1[]="sirul 1", sir2[]="sirul 2", *psir=sir1;
cout<<"a="<<a<<" &a="<<&a<<" b="<<b<<" &b="<<&b<<'\n';
cout<<"*pi="<<*pi<<"pi="<<pi<<" &pi="<<&pi<<'\n';
cout<<"*pd="<<*pd<<"pd="<<pd<<" &pd="<<&pd<<'\n';
cout<<"*sir1="<<*sir1<<" sir1="<<sir1<<" &sir1="<<&sir1<<'\n';
// *sir1=s  sir1=sirul 1  &sir1=0xffd6
cout<<"*sir2="<<*sir2<<" sir2="<<sir2<<" &sir2="<<&sir2<<'\n';
// *sir2=s  sir2=sirul 2  &sir1=0xffce
cout<<"*psir="<<*psir<<" psir="<<psir<<" &psir="<<&psir<<'\n';
// *psir=s  psir=sirul 1  &sir1=0xffcc
cout<<"sir1+2="<<(sir1+2)<<" psir+2="<<(psir+2)<<'\n';
// sir1+2=rul 1  psir+2=rul 1
cout<<"*(sir1+2)="<< *(sir1+2)<<'\n';
// *(sir1+2)=r  valoarea elementului de indice 2
void *pv1, *pv2;
pv1=psir; pv2=sir1;
cout<<"pv1="<<pv1<<"&pv1="<<&pv1<<'\n';
cout<<"pv2="<<pv2<<"&pv2="<<&pv2<<'\n';
pi=&b; pd=&v; psir=sir2;
cout<<"*pi="<<*pi<<"pi="<<pi<<" &pi="<<&pi<<'\n';
cout<<"*pd="<<*pd<<"pd="<<pd<<" &pd="<<&pd<<'\n';
```

```
cout<<"*psir="<<*psir<<"psir="<<psir<<" &psir="<<&psir<<'\n';
}
```

Exerciţiu: Să se scrie un program care citeşte elementele unui vector de întregi, cu maxim 20 elemente şi înlocuieşte elementul maxim din vector cu o valoare introdusă de la tastatură. Se va folosi aritmetica pointerilor.

```
#include <iostream.h>
void main()
{ int a[20];
int n, max, indice; cout<<"Nr. elemente:"; cin>>n;
for (i=0; i<n; i++)
      { cout<<"a["<<i<<"]="; cin>>*(a+i);}
// citirea elementelor vectorului
max=*a; indice=0;
for (i=0; i<n; i++)
      if (max<=*(a+i))
            { max=*(a+i); indice=i;}
// aflarea valorii elementului maxim din vector şi a poziţiei acestuia
int val;
cout<<"Valoare de inlocuire:"; cin >> val;
*(a+indice)=val;
// citirea valorii cu care se va înlocui elementul maxim
for (i=0; i<n; i++)
       cout<<*(a+i);<<'\t';
cout<<'\n';
// afişarea noului vector
/* în acest mod de implementare, în situaţia în care în vector există mai multe elemente a căror
valoare este egală cu valoarea elementului maxim, va fi înlocuit doar ultimul dintre acestea (cel de
indice maxim).*/
}
```

5.3.2. POINTERI ŞI TABLOURI MULTIDIMENSIONALE

Elementele unui tablou bidimensional sunt păstrate tot într-o zonă continuă de memorie, dar inconvenientul constă în faptul că ne gândim la aceste elemente în termeni de rânduri (linii) şi coloane (figura 5.3). Un tablou bidimensional este tratat ca un tablou unidimensional ale cărui elemente sunt tablouri unidimensionale.

```
int M[4][3]={ {10, 5, -3}, {9, 18, 0}, {32, 20, 1}, {-1, 0, 8} };
```

Compilatorul tratează atât M, cât şi M[0], ca *tablouri* de mărimi diferite. Astfel:

```
cout<<"Marime M:"<<sizeof(M)<<'\n';                 // 24 = 2octeţi * 12elemente
cout<<"Marime M[0]"<<sizeof(M[0])<<'\n';            // 6 =  2octeţi * 3elemente
cout<<"Marime M[0][0]"<<sizeof(M[0][0])<<'\n';      // 4 octeţi (sizeof(int))
```

Matricea **M**

M[0]	10	5	-3
M[1]	9	18	0
M[2]	32	20	1
M[3]	-1	0	8

Matricea `M` are 4 linii şi 3 coloane.
Numele tabloului bidimensional, M, referă întregul tablou;
`M[0]` referă prima linie din tablou;
`M[0][0]` referă primul element al tabloului.

Figura 5.3. Matricea M

Aşa cum compilatorul pointer, un tablou bidimensional este referit într-o manieră similară. Numele tabloului bidimensional, `M`, reprezintă adresa (pointer) către primul element din tabloul bidimensional, acesta fiind prima linie, M[0] (tablou

unidimensional). `M[0]` este adresa primului element (M[0][0]) din linie (tablou unidimensional), deci M[0] este un pointer către int: `M = M[0] = &M[0][0]`. Astfel, `M` şi `M[linie]` sunt pointeri constanţi.

Putem concluziona:

- **`M`** este *un pointer către un tablou unidimensional* (de întregi, în exemplul anterior).
- **`*M`** este *pointer către int* (pentru că M[0] este pointer către int), şi **`*M = *(M + 0)` ⇔ `M[0].`**
- **`**M`** este *întreg*; deoarece **`M[0][0]`** este int, **`**M=*(*M)⇔*(M[0])=*(M[0]+0)⇔ M[0][0]`**.

Exerciţiu: Să se testeze programul următor, urmărind cu atenţie rezultatele obţinute.

```
#include <iostream.h>
#include <conio.h>
void main()
{int a[3][3]={{5,6,7}, {55,66,77}, {555,666,777}};
clrscr();
cout<<"a="<<a<<" &a="<<&a<<" &a[0]="<<&a[0]<<'\n';
cout<<"Pointeri catre vectorii linii\n";
for (int i=0; i<3; i++){
      cout<<" *(a+"<<i<<")="<<*(a+i);
      cout<<" a["<<i<<"]="<<a[i]<<'\n';
}
// afişarea matricii
for (i=0; i<3; i++){
      for (int j=0; j<3; j++)
            cout<<*(*(a+i)+j)<<'\t'; //sau:
            cout<<*(a[i]+j)<<'\t';
      cout<<'\n';
      }
}
```

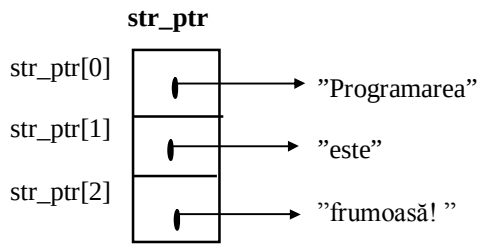
5.4. TABLOURI DE POINTERI

Un tablou de pointeri este un tablou ale cărui elemente sunt pointeri. Modul general de declarare a unui tablou de pointeri:

```
tip *nume_tablou[dim];
```

Să considerăm exemplul în care se declară şi se iniţializează tabloul de pointeri **`str_ptr`** (figura 5.4.):

```
char * str_ptr[3] = { "Programarea", "este", "frumoasă!" };
```



Deoarece operatorul de indexare `[ ]` are prioritate mai mare decât operatorul de deferenţiere `*`, declaraţia `char*str_ptr[3]` este echivalentă cu `char*(str_ptr[3])`, care precizează că `str_ptr` este un vector de trei elemente, fiecare element este pointer către caracter.

Figura 5.4. Tabloul de pointeri **str_ptr**

În ceea ce priveşte declaraţia: `char*(str_ptr[3])`, se poate observa:

1. **`str_ptr[3]`** este de tipul **`char *`** (fiecare dintre cele trei elemente ale vectorului `str_ptr[3]` este de tipul pointer către char);
2. **`*(str_ptr[3])`** este de tip **`char`** (conţinutul locaţiei adresate de un element din `str_ptr[3]` este de tip *char).*

Fiecare element (pointer) din `str_ptr` este iniţializat să pointeze către un şir de caractere constant. Fiecare dintre aceste şiruri de caractere se găsesc în memorie la adresele memorate în elementele vectorului `str_ptr`: `str_ptr[0], str_ptr[1]`, etc.

Să ne amintim de la pointeri către şiruri de caractere:

```
char  *p="heLLO";
*( p+1) = 'e'     ⇔       p[1] = 'e';
```

În mod analog:

```
str_ptr[1] = "este";
*( str_ptr[1] + 1) = 's';     ⇔     str_ptr[1][1]='s';
```

Putem conculziona:

- **str_ptr** este un *pointer către un pointer de caractere*.
- ***str_ptr** este *pointer către char*. Este evident, deoarece `str_ptr[0]` *este pointer către char*, iar ***str_ptr = *(str_ptr [0] + 0) ⇔ str_ptr[0].**
- ****str_ptr** este un de tip *char*. Este evident, deoarece `str_ptr[0][0]` *este de tip char*, iar ****str_ptr=*(*str_ptr)⇔*(str_ptr[0])=*(str_ptr[0]+0)⇔ str_ptr[0][0].**

5.5. POINTERI LA POINTERI

Să revedem exemplul cu tabloul de pointeri ***str_ptr***. Şirurile spre care pointează elementele tabloului pot fi accesate prin `str_ptr[index]`, însă deoarece `str_ptr` este un *pointer constant,* acestuia nu i se pot aplica operatorii de incrementare şi decrementare. Este ilegală :

```
for (i=0;i<3;i++)
      cout<<str_ptr++ ;
```

De aceea, putem declara o variabilă pointer ***ptr_ptr***, care să pointeze către primul element din str_ptr. Variabila ***ptr_ptr*** este ***pointer către pointer*** şi se declară astfel:

```
char **ptr_ptr;
```

În exemplul următor este prezentat modul de utilizare a pointerului la pointer ptr_ptr (figura 5.5).

Exemplu:

```
char **ptr_ptr;
char *  str_ptr[3] = { "Programarea", "este", "frumoasă!" };
char **ptr_ptr;
ptr_ptr = str_ptr;
```

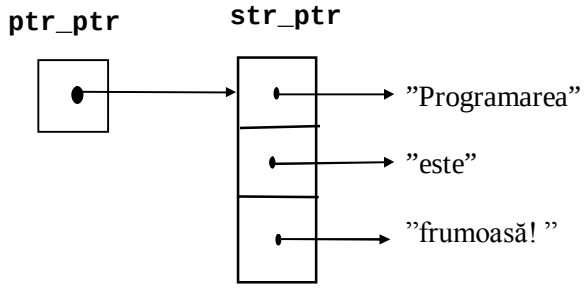


Figura 5.5. Pointerul la pointer **ptr_ptr**

După atribuire, şi `str_ptr` şi `ptr_ptr` pointează către aceeaşi locaţie de memorie (primul element al tabloului `str_str`). În timp ce fiecare element al lui str_str este un pointer, ptr_ptr este un *pointer către pointer*. Deoarece ptr_ptr este un pointer variabil, valoarea lui poate fi schimbată:

```
for (i=0;i<3;i++)
      cout<<ptr_ptr++ ;
```

Referitor la declaraţia **char **ptr_ptr**, putem concluziona:

- **ptr_ptr** este de *tipul* **char**** (*ptr_ptr este pointer la pointer către char*);
- ***ptr_ptr** este de *tipul* **char*** (*conţinutul locaţiei ptr_ptr este de tipul pointer către char*);
- ****ptr_ptr** este de *tipul* **char** (****ptr_ptr ⇔ *(*ptr_ptr)**; *conţinutul locaţiei *ptr_ptr este de tipul char*).

5.6. MODIFICATORUL *const* ÎN DECLARAREA POINTERILOR

Modificatorul ***const*** se utilizează frecvent în declararea pointerilor, având următoarele roluri:

- Declararea unui ***pointer*** spre ***o dată constantă***

```
const *tip nume_pointer=dată_constantă;
```

Exemplu:

```
const char *sirul=”azi”;
//variabila sirul este pointer spre un şir constant de caractere
```

Atribuirile de forma:

```
*sirul=”coco”;
*(sirul+2)=’A’;
```

nu sunt acceptate, deoarece pointerul `sirul` pointează către o dată constantă (şir constant).

- Declararea ***unui pointer constant*** către *o* ***dată care nu este constantă***

```
tip * const nume_pointer=dată_neconst;
```

Exemplu:

```
char * const psir=”abcd”; const char *sir=”un text”;
sir=”alt sir”;                  //incorect, sir pointează către dată constantă
psir=sir;                       //incorect, deoarece psir este pointer constant
```

- Declararea unui ***pointer constant*** către o ***dată constantă***

```
const tip * const nume_pointer=dată_constantă;
```

Exemplu:

```
const char * const psir1="mnP";
*(psir1+2)='Z';     // incorect, data spre care pointeză psir1 este constantă
psir1++;            // incorect, psir1 este pointer constant
```

ÎNTREBĂRI ŞI EXERCIȚII

Chestiuni teoretice

1. În ce constă operația de incrementare a pointerilor?
2. Tablouri de pointeri.
3. Ce sunt pointerii generici?
4. Ce operații se pot realiza asupra variabilelor pointer?
5. De ce numele unui pointer este lvalue?
6. Ce fel de variabile pot constitui operandul operatorului de deferențiere?
7. Operatorul de referențiere.
8. Unui pointer generic i se poate atribui valoarea unui pointer cu tip?
9. Care este legătura între tablouri şi pointeri?
10. De ce numele unui tablou este rvalue?

Chestiuni practice

1. Să se implementeze programele cu exemplele prezentate.
2. Să se scrie programele pentru exercițiile rezolvate care au fost prezentate.
3. Analizați următoarele secvențe de instrucțiuni. Identificați secvențele incorecte (acolo unde este cazul) şi sursele erorilor:
   - `int a,b,*c; a=7; b=90; c=a;`
   - `double y, z, *x=&z;  z=&y;`
   - `char x, **p, *q; x = 'A'; q = &x; p = &q; cout<<”x=”<<x<<’\n’;`
     `cout<<”**p=”<<**p<<’\n’; cout<<”*q=”<<*q<<’\n’;`
     `cout<<”p=”<<p<<” q=”<<q<<”*p=”<<*p<<’\n’;`
   - `char *p, x[3] = {'a', 'b', 'c'};  int i, *q, y[3] = {10, 20, 30};`
     `p = &x[0];`

```
for (i = 0; i < 3; i++)
{
cout<<"*p="<<*p<<" p="<<p<<'\n';
p++;
}
q = &y[0];
for (i = 0; i < 3; i++)
{
cout<<"*q="<<*q<<"q="<<q<<'\n';
q++;
}
```

- `const char *sirul="să programăm"; *(sirul)++;`
- `double a, *s; s=&(a+89); cout<<"s="s<<'\n';`
- `double a1, *a2, *a3; a2=&a1; a2+=7.8; a3=a2; a3++;`
- `int m[10], *p;p=m;`
  `for (int i=0; i<10; i++)`
  `cout<<*m++;`
- `void *p1; int *p2; int x; p2=&x; p2=p1;`
- `char c='A'; char *cc=&c; cout<<(*cc)++<<'\n';`

4. Rescrieți programele pentru problemele din capitolul 4 (3.a.-3.g., 4.a.-4.i., 5.a.-5.h.), utilizând aritmetica pointerilor.