8. VARIABILE POINTER

8.1 Tipul variabilei pointer

Pentru programatorii care cunosc un limbaj de asamblare, definirea şi utilizarea variabilelor pointer în limbaje evoluate de programare reprezintă operanzii necesari implementării adresării indirecte.

La adresarea indirectă se utilizează doi operanzi şi anume: operandul care referă şi operandul referit, notaţi în continuare R_0, respectiv O_r.

Operandul R_0 conţine adresa de început a zonei de memorie asociată operandului O_r. Evaluarea expresiilor:

$$\mathrm{R0} = \mathrm{adr(Or)}, \mathrm{Or} = 15 \qquad (8.1)$$

conduce la modificarea conţinutului operandului R_0 şi a operandului O_r, figura 8.1, ceea ce se exprimă prin:

$$\mathrm{cont(R0)} = \mathrm{adr(Or)} \qquad (8.2)$$

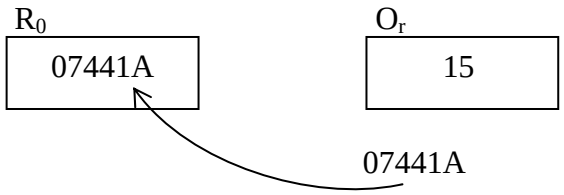

Figura 8.1 *Modificarea conţinuturilor operanzilor R_0 şi O_r*

Această expresie arată că operandul R_0 este de un nou tip, T_p, ale cărui valori aparţin intervalului $[A_i, A_f] \cap N$, unde A_i şi A_f sunt adrese de început, respectiv, de sfârşit ale unui segment de memorie.

Dacă:

$$\mathrm{f(cont(R0), Tp)} = \mathrm{TRUE} \qquad (8.3)$$

adică

$$cont\ (R_0) \cap [Ai, Af] \cap N \neq \phi \qquad (8.4)$$

se spune ca operandul R_0 este de tipul T_p.

În continuare acest nou tip de dată este numit tipul pointer.

Funcţia $lg(R_0,T_P)$ defineşte o astfel de mărime care să permită stocarea completă a informaţiei necesare localizării operanzilor, ale căror adrese se încarcă în variabila R_0.

Expresia:

$$cont(R_o) = adr(R_o) \qquad (8.5)$$

are semnificaţia încărcării în operandul R_0 a propriei adrese. După evaluarea expresiei, conţinutul lui R_0 este dat în figura 8.2.

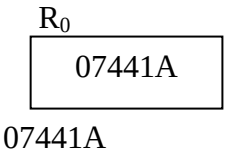

Figura 8.2 *Noul conţinut al operandului* R_0

În sine, expresia nu are o semnificaţie deosebită, mai mult oferă riscul închiderii unui lanţ de referire, transformându-l în ciclu infinit. Dacă operandul R_0 devine reper şi toate deplasările sunt evaluate avându-l ca bază, expresia de mai sus se justifică.

Dacă se definesc variabilele *a, b, c* având tipul T_i, şi variabilele p_a, p_b, p_c având tipul T_p, evaluarea expresiilor:

$$p_a = adr(a)$$
$$p_b = adr(b)$$
$$p_c = adr(c)$$

realizează iniţializarea operanzilor de tip T_p.

Dacă sunt luate în considerare aceste expresii, în loc de:

$$c = a+b; \quad (8.6)$$

se va utiliza expresia:

$$ref(p_c) = ref(p_a)+ref(p_b) \quad (8.7)$$

unde funcţia de referire *ref()* este definită:

$$ref : [A_i, A_f] \rightarrow I \quad (8.8)$$

unde mulţimea este interpretată drept conţinutul zonei de memorie asociate identificatorilor.

Se observă că prin definirea funcţiei *adr()*:

$$adr : I \rightarrow [A_i, A_f] \quad (8.9)$$

funcţiile *adr()* şi *ref()* nu sunt una inversă celeilalte. Este adevărată numai egalitatea:

$$x = ref(adr(x)) \quad (8.10)$$

Nu în toate cazurile, limbajele de programare implementează această proprietate.

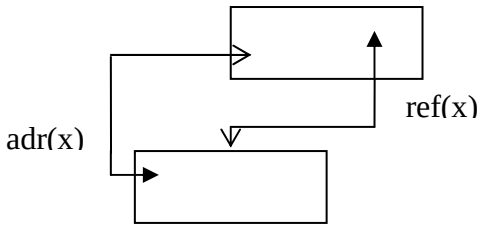

Figura 8.3 *Relaţia între funcţiile adr(x) şi ref(x)*

Proprietăţile funcţiilor sunt discutate în contextul alocării statice a memoriei, fără posibilitatea redefinirii sau suprapunerii de operanzi, obţinându-se zone de memorie comune pentru seturi de operanzi.

Deci, în contextul prefixat, pentru doi operanzi diferiţi, x_i şi x_j de acelaşi tip sau de tipuri diferite.

$$adr(x_i) \neq adr(x_j) \qquad (8.11)$$

$$ref(adr(x_i)) \neq ref(adr(x_j)) \qquad (8.12)$$

pentru că $x_i \neq x_j$, din definiţie.

Variabilele de tip T_p apar în expresii aritmetice, relaţionale sau de atribuire, nu ca elemente în sine ci ca parametri ai funcţiei *ref()*.

Revenind la limbajul de asamblare, operaţiile de calcul sunt specializate pe tipuri de operanzi. Există operator de adunare pentru operanzi de tip întreg, dar există şi operator de adunare pentru operanzi de tip virgulă mobilă simplă precizie. Pentru operanzii de tip dublă precizie, există operator distinct de cei menţionaţi.

Operatorii de comparare privesc zone de memorie de lungimi diferite, care să acopere totalitatea lungimilor asociate tipurilor fundamentale de date implementate în limbajele de asamblare.

Apariţia operanzilor de tip T_p, determină pierderea informaţiei referitoare la tipul iniţial al operandului şi există posibilitatea generării unei evaluări ambigue a expresiilor.

În acest sens, tipul T_p se defineşte ca un tip compus:

$$T_p = (pointer, T_i) \qquad (8.13)$$

unde T_i, reprezintă tipul variabilei referite şi pointer este tipul specific, fundamental al variabilei. Dacă variabila care referă *px*, este de tip T_p şi variabila referită *x*, este de tip T_i, în vorbirea curentă expresia se interpretează, pornind de la efectul fizic al operaţiilor cu pointeri, ca pointer spre T_i.

Dacă *x* are tipul *int* şi *px* este de tip T_p, se spune că *px* este pointer spre *int* şi toţi operatorii din expresiile omogene ca tip de date unde apare *px*, sunt selectaţi pentru lucrul cu operanzi de tip *int*.

Tipul variabilei referite nu influenţează lungimea variabilelor de tip T_p şi conţinutul.

Dacă o variabila *x*, este definită ca având tipul T_i şi variabila *px* este definită ca având tipul (pointer, T_i), expresia:

$$px = adr(x) \quad (8.14)$$

nu este evaluată decât numai după ce ara loc conversia de la tipul T_i la tipul T_j, operatorul de atribuire necesitând operanzi de acelaşi tip. Această conversie realizează de fapt schimbarea setului de informaţii în concordanţă cu operandul din stânga.

Prin conversie, adresa este asociată unei variabile de tip T_i cu toate implicaţiile ce decurg asupra alinierilor şi lungimii zonelor de memorie asociate noului tip, asociat zonei de memoria pusă în corespondenţă cu variabila *x*.

Există situaţii în care conversia de tip este implicită (în cazul evaluării expresiilor aritmetice), dar sunt numeroase cazurile în care aceasta trebuie specificată explicit.

Funcţia *conv(x, T_j)* modifică tipul variabilei *x* de la T_i la T_j. Dacă:

$$tip(x) = T_i \qquad (8.15)$$

$$x = conv(x,T_j) \qquad (8.16)$$

conduce la:

$$tip(x) = T_j \qquad (8.17)$$

şi în mod corespunzător:

$$px = convp(adr(x),T_j) \qquad (8.18)$$

unde *convp( )* reprezintă funcţia de conversie a tipului spre care pointează o variabila de tip T_p.

Expresia din dreapta conduce mai întâi prin evaluarea lui *adr(x)* la tipul (pointer,T_i), ca mai apoi prin folosirea funcţiei *convp()*, să se obţină conversia de tip şi rezultatul evaluării este (pointer, T_i), care este omogen în raport cu tipul T_p al variabilei *px*.

Funcţia *convp()* are rolul de a modifica al doilea termen al perechii (pointer, T_i) care defineşte tipul T_p.

Conversiile de tip, sunt facilitaţi cu importante efecte asupra performanţei programelor, cu condiţia ca ele să se afle strict sub controlul programatorului.

Atunci când conversiile de tip depăşesc simplele expresii de atribuire, prin crearea de noi vecinătăţi pentru operanzi prin redistribuirea baiţilor, sunt interpretate componente care altfel rămâneau neexplorate în program. Toate însă, e necesar să fie sub controlul programatorului. Altfel, dintr-o facilitate, conversia de tip se transformă într-o problemă generatoare de greutăţi.

8.2 Aritmetica variabilelor pointer

Întrucât pentru variabilele *px* şi *py* de tip T_p:

$$cont(px) \in [A_i, A_f] \cap N \qquad (8.19)$$

$$cont(py) \in [A_i, A_f] \cap N \qquad (8.20)$$

pe mulţimea $[A_i, A_f] \cap N$ se defineşte operaţia „o”, astfel încât pentru:

$$\forall \alpha,\beta \in [A_i, A_f] \cap N,\ \alpha\ o\ \beta = \gamma,\ \gamma \in [A_i, A_f] \cap N \quad (8.21)$$

Dacă, de exemplu, se consideră intervalul [0; 64000] ∩ N, operaţiile de adunare, înmulţire, scădere şi împărţire, nu sunt legi de compoziţie internă. Prin contraexemple se dovedeşte că există α şi β astfel încât:

$$[0;64000] \cap N \quad (8.22)$$

Aparent, nu se vorbeşte de o aritmetică a variabilelor pointer. Totuşi se construiesc expresiile cu variabile pointer, cu condiţia ca rezultatul evaluării să aparţină intervalului $[A_i, A_f] \cap N$.

Dacă se defineşte masivul unidimensional:

```
float x [10];
```

şi *px* este un pointer spre întreg, şi:

$$px = adr(x[l]) \quad (8.23)$$

evaluarea expresiei:

$$px = px + 6 = adr(x[l]) + l*lg(real) = adr(x[2]) \quad (8.24)$$

conduce la posibilitatea de a referi, elementul al doilea al vectorului x. Dacă:

$$px = adr(x[10]) \quad (8.25)$$

şi se evaluează expresia:

$$px = px\text{-}24 \quad (8.26)$$

$$px = adr(x[10])\text{-}4*lg(real) = adr(x[6]) \quad (8.27)$$

în expresiile:

$$adr(z+k) = adr(z)+k*lg(T_i) \quad (8.28)$$

unde *z* este o variabilă de tip T_i, se observă că pentru variabila *pz* de tip T_p, care a fost iniţializată prin:

$$pz: = adr(z); \quad (8.29)$$

$$adr(z+k) = pz + k*lg(T_i) \quad (8.30)$$

şi pentru că tipul T_p = *(pointer,* T_i*)*, expresia $k*lg(T_i)$ este rezultatul conversiei la tipul T_p al variabilei *pz*, a variabilei întregi *k*.

În acelaşi mod, se defineşte şi operaţia de scădere, în ambele cazuri apare cerinţa ca rezultatul evaluării expresiilor să aparţină intervalului $[A_i, A_f] \cap N$.

Se construiesc teoretic, expresii deosebit de complexe, în care operanzii să fie de tip T_p, dar restricţia de apartenenţă a rezultatului la acel interval le face inoperante.

În plus, lucrul cu adrese în zone de memorie contigue şi mai ales pentru structuri de date omogene, vizează posibilitatea de a genera termenii unor progresii aritmetice.

Dacă variabila *px* de tip T_p, este iniţializată prin:

```
px = adr(x[1]);
s = 0;
for(i=0; i<10; i++)
{
     s = s + ref(px);
     px = px+4;
};
```

variabila *px*, permite referirea rând pe rând a elementelor *x[0], x[1], ..., x[9]* ale vectorului *x*.

Mai mult, definind *px* variabila de tip T_p în secvenţa:

```
px = adr(x);
s = 0;
pi = adr(x[0]);
do
{
     s = s+ref(pi);
     pi = pi+4 ;
} while (pi <= adr(x[9]));
```

diferă foarte puţin de secvenţele care implementează structurile repetitive într-un limbaj de asamblare oarecare.

Ca şi în limbajele de asamblare, restricţiile de apartenenţă la domeniul $[A_i; A_f] \cap N$, au determinat că pentru variabilele pointer, aritmetica să se reducă la două operaţii: incrementarea şi decrementarea. Aceste operaţii înseamnă de fapt majorarea, respectiv diminuarea cu o unitate, după cum urmează:

$$adr(x+l) = adr(x) + 1*lg(T_i) = px + l*lg(T_i) \qquad (8.31)$$

$$adr(x-l) = px - l*lg(T_i) \qquad (8.32)$$

Conversia de tip, determină ca unitatea să fie egală de fapt cu lungimea asociată zonei de memorie ocupată de variabilele de tip T_i.

În cazul unor expresii complexe notate:

$$expr(a_0\ a_1\ ...\ a_{n-1}) \qquad (8.33)$$

$$adr(x+expr(a_0\ a_1\ .....a_{n-1})) = px+int(expr(a_0\ a_1\ .....a_{n-1}))*lg(T_i) \qquad (8.34)$$

Se consideră că expresia:

$$px+int(expr(a_0\ a_1\ .....a_{n-1}))*lg(T_i) = \alpha \qquad (8.35)$$

este corect definită dacă $\alpha \in [A_i, A_f] \cap N$.

În limbajul C/C++, încărcarea adreselor variabilelor definite în program, se efectuează cu operatorul „&”.

Corespondentul C/C++ al secvenţelor de mai sus este:

```
typedef int mat[10];
mat x = {1,2,3,4,5,6,7,8,9,10};
int * px;

unsigned int i, s;
{
s = 0;
for  ( i  =  0 ; i < 10 ; i + +)
{
   px = &x[i]
   s = s + (*px);
}
cout << s;
```

Absenţa controlului asupra limitelor de variaţie pentru variabilele de tip T_p, determină să devină operanzi zone nespecifice programului, alterând în acest fel inclusiv componente din software de bază, cu consecinţe asupra calităţii prelucrărilor curente şi chiar viitoare.

8.3 Niveluri de indirectare

Apare în mod firesc întrebarea dacă tipul T_p definit iniţial (pointer, T_i), unde T_i reprezintă un tip fundamental de dată, include şi perechea (pointer, T_p), ştiut fiind faptul că şi tipul T_p este considerat tip fundamental.

Acceptând că o astfel de construcţie este corectă, se creează posibilitatea realizării de pointeri spre pointeri spre T_i şi a pointerilor spre pointeri spre pointeri spre T_i.

Se definesc variabilele:

```
int  Y, x ;
int *py, *px ;   // (pointer, integer);
int **ppx;       // (pointer,(pointer, integer));
```

prin expresiile:

```
x  = 20;
px = adr(x);
ppx = adr(px));
```

se obţin legăturile reprezentate grafic în figura 8.4.

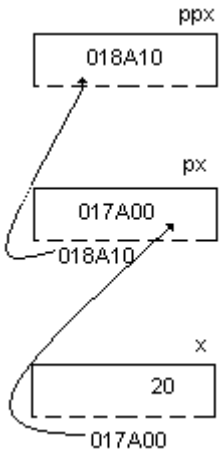

Figura 8.4 *Alocarea în memorie a variabilelor x, px şi ppx*

Expresiile:

```
py = adr(y);
y = 16;
```

conduc la reprezentarea grafică din figura 8.5.

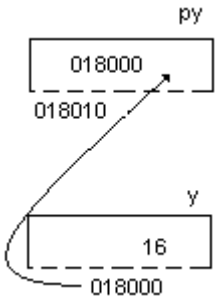

Figura 8.5 *Alocarea în memorie a variabilelor py şi y*

Expresia:

$$ref(y) = ref(y) + ref(ref(x)) \qquad (8.36)$$

este echivalentă cu:

$$y = y + x \qquad (8.37)$$

Programul C/C++ care evaluează expresia este:

```
typedef  int * pint;
int x, y;
pint px, py;
pint *ppx;
{
      x  =  20 ;
```

```
        y  = 15;
        px = &x;
        py = &y;
        ppx  =  &px;
        *py = *py + **ppx;
        cout << *py ;
};
```

Construcţia (pointer, T_p) permite compunerea unei funcţii *adr()* asociate nivelurilor indirectate.

În definiţia:

```
int x ;
int *px ; // (pointer, integer)
int **ppx ; // (pointer, (pointer, integer))
int ***pppx ; // (pointer, (pointer,(pointer, integer)))
```

şi din secvenţa:

```
x = 17;
px =adr(x);
ppx = adr(px);
pppx = adr(ppx);
```

se obţine:

$$pppx - adr(adr(adr(x))) = adr^3(x) \qquad (8.38)$$

Numărul 3 reprezintă nivelul de indirectare. Pentru referirea unei variabile având nivelul 3 de indirectare, se procedează astfel:

$$ref(ref(ref(pppx))) = ref^3(pppx) \qquad (8.39)$$

Pentru omogenizarea operanzilor în cadrul expresiilor, este necesar ca variabilele în care tipul T_p are puterea n, să fie iniţializate cu *adr()* şi să fie referite cu $ref^n()$.

Se consideră:

$T^1_p = (pointer, T_i)$
$T^2_p = (pointer, (pointer, T_i)$
$T^3_p = (pointer, (pointer\ (pointer, T_i)))$
...

Pentru efectuarea unor generalizări privind aritmetica ordinului n a tipului T_p este necesară definirea funcţiei pentru evaluarea lungimii acestui tip. Astfel, expresii ca:

$$adr^k(adr^h(x)+i) \text{ sau } adr^k(adr^h(x)-i) \qquad (8.40)$$

devin interpretabile.

Aşa cum pentru toate tipurile de date există constante ce sunt atribuite şi pentru tipul pointer, există constante corespunzător definite. Constantele pentru tipul pointer, simbolizează adrese absolute. Se vorbeşte de adresă nulă, se vorbeşte de adresa 15, sau de orice altă adresă. Dacă însă un program *P* lansat în execuţie, ocupă zona de memorie delimitată prin $[A_i^p; A_f^p] \cap N$, iniţializarea oricărei variabile pointer, este efectuată cu valori cuprinse în acest interval. Folosirea valorii nule are numai scop de jalonare la iniţializare, pentru a vedea dacă s-au făcut atribuiri ulterioare pentru a lucra corect, sau atribuirile nu au fost posibil să se efectueze şi deci nu se lucrează pentru că variabila pointer are valoare nulă.

În programele C/C++, referirea unei variabile pointer *px* cu 5 nivele de indirectare, de exemplu, se realizează prin evaluarea expresiei:

```
*****px;
```

iar pentru valoarea nulă a variabilei pointer, se foloseşte constanta simbolică NULL. Iniţializarea unei variabile pointer cu o adresă absolută de memorie se realizează cu funcţia *Ptr()*, având ca parametru o adresă în hexazecimal. De exemplu:

```
py : =  Ptr($00AA,$0020);
```

variabila pointer py este iniţializată cu o adresă absolută de memorie.

8.4 Vectori şi matrice de pointeri

Şi cu tipul T_p se construiesc structuri de date omogene, dacă toate componentele acestora au tipul T_p. Construcţia:

$$T_p\ x[n]; \qquad (8.41)$$

$$T_{pi}\ y[n][m]; \qquad (8.42)$$

reprezintă definirea unui masiv *x* având *n* componente, fiecare componentă fiind un pointer spre T_i şi, respectiv, definirea unei matrice *y* având *n* linii şi *m* coloane, elemente ce sunt pointeri spre tipul T_i.

Se justifică stocarea în masive uni- şi bidimensionale a adreselor unor operanzi, dacă aceştia au diferenţieri între ei, sau dacă tipologiile determină activări de funcţii după succesiuni stabilite.

Se memorează, de exemplu, mesajele unui program care apar în dialogul cu utilizatorul, sub forma unui text continuu. Acestor mesaje, în număr de 50, li se memorează adresa de început în componentele vectorului *text[i]*, de pointeri spre tipul *char [].*

Dacă se doreşte aflarea tuturor mesajelor prin secvenţa:

```
for (i = 0 ; i < 50 ; i + + )
      cout << ref(text[i]);
```

se obţine acelaşi lucru.

Dacă se doreşte afişarea unui anumit mesaj, este important să se cunoască poziţia în vectorul *text[]* a componentei în care este memorată adresa respectivului mesaj.

Dacă în cele cinci componente ale unui vector de pointeri numit *px*, se memorează adresele primelor componente ale vectorilor *a[], b[], c[], d[], e[],* cu număr de componente diferite şi se doreşte să se calculeze cu funcţia *suma()*, suma elementelor vectorilor, în loc să se scrie de cinci ori apelul funcţiei *suma()*, se va construi secvenţa:

```
for(i=0; i<5; i++)
   s[i] = suma(px[i], n[i]);
```

Dacă se defineşte:

```
int x1[10], x2[10], x3[10], x4[10];
Tp y[4];
T 2p  z;
```

prin atribuirile:

```
y[0]=adr(x1[0]);
y[1]=adr(x2[0]);
y[2]=adr(x3[0]);
y[3]=adr(x4[0]);
z = adr(y[0]);
```

s-a obţinut construcţia cu modelul grafic din figura 8.6.

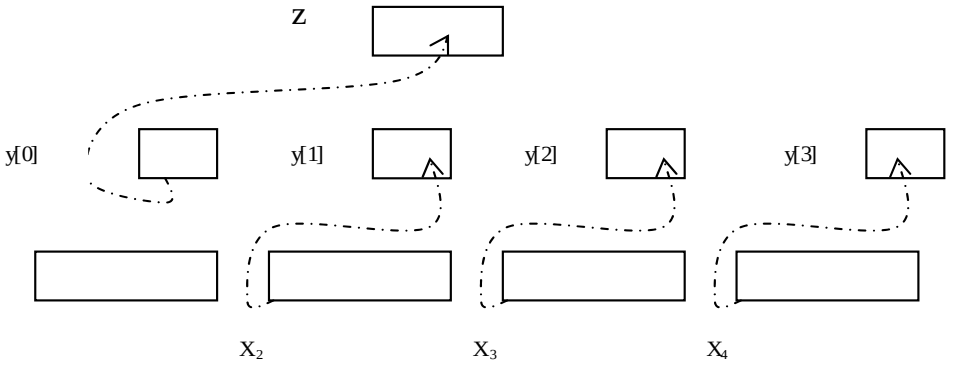

Figura 8.6 *Modelul grafic al legăturilor între variabilele x, y şi z*

Modelul grafic din figura 8.6 este identic cu modelul grafic asociat structurii de date omogene şi contigue, matricea.

Tot astfel, se defineşte o funcţie *fct()*, care se apelează prin:

$$fct(p_1, p_2, p_3, \ldots, p_n) \qquad (8.43)$$

La apel, construieşte un vector cu *n* componente în care sunt memorate adresele parametrilor reali ai funcţiei, deci vectorul acesta este un vector de pointeri.

Adresa primei componente a vectorului este memorată într-un registru. Acest registru conţine adresa listei de adrese a parametrilor.

În cazul în care la definire funcţia are parametri formali: α_1, α_2,....., α_n şi dacă:

$$tip(p_i) = tip(\alpha_i) \ \forall i \in \{1, 2, ..., n\} \qquad (8.44)$$

înseamnă că s-a obţinut concordanţa între tipul parametrilor reali şi tipul parametrilor formali.

Transmiterea parametrilor formali prin valoare vizează efectuarea copierii valorilor parametrilor reali *pi* în zonele de memorie definite în funcţie, asociate parametrilor α_i, operaţie simbolizată prin expresia de atribuire:

$$\alpha_i = p_i; \quad i = 1, 2, ...,n \qquad (8.45)$$

În cazul în care are loc o inversare a parametrilor sau omiterea unuia dintre ei, dispare concordanţa de tip şi tipul parametrului α_i este T_i, iar tipul parametrului p_i este T_j, ceea ce impune efectuarea conversiei de la tipul T_j la tipul T_i, cu toate efectele pe care conversia de tip le antrenează.

$$\alpha_i = conv(p_i, Ti) \qquad (8.46)$$

determină perturbarea radicală a rezultatelor.

Dacă parametrii p_i sunt variabile pointer în funcţie, se operează asupra zonelor de memorie externe acesteia, a zonelor ale căror adrese sunt conservate în variabilele pointer p_i.

Aceasta explică necesitatea ca funcţia de interschimb de valori să conţină pointeri pentru elemente şi nu elementele însăşi.

În continuare se ia în discuţie programul executabil ca dată. Orice program executabil este format din instrucţiuni executabile şi zone de memorie ce servesc ca operanzi.

Orice program are o primă instrucţiune executabilă şi o ultimă instrucţiune executabilă. De obicei, în cazul funcţiilor, prima instrucţiunea executabilă, etichetată cu numele funcţiei, se numeşte punct de intrare în funcţie. Instrucţiunea de apel a funcţiei, efectuează un salt necondiţionat spre punctul de intrare în funcţie.

Ultima instrucţiune executabilă dintr-o funcţie, este un salt necondiţionat către programul apelator, pe baza informaţiei care localizează unde se efectuează revenirea. Această instrucţiune se numeşte punct de ieşire din program. Din punct de vedere al structurilor de date, aceste puncte sunt de fapt elemente folclorice, care prin pitorescul lor colorează limbajul programatorilor.

Ca structură de date, textul executabil este un text omogen, atunci când toate instrucţiunile au o lungime şi o structură fixată. În cele mai multe cazuri, instrucţiunile necesită informaţii care determină extensii pe cuvinte adiacente, reducând gradul de omogenitate a structurii de date numită program.

Totuşi, structura de date numită program executabil este delimitată prin două instrucţiuni executabile cu aceeaşi semnificaţie, oricare ar fi funcţia scrisă într-un limbaj evoluat. Acest lucru se datorează standardelor de preluare a parametrilor şi de revenire în funcţia apelatoare.

Dacă se consideră funcţiile $f_1()$, $f_2()$, ..., $f_m()$ cărora le corespund *m* texte program executabil, memorate în *m* zone de memorie, prin $f_l()$, $f_2()$, ... $f_m()$, se simbolizează adresele primelor instrucţiuni executabile ale acestor funcţii, şi dacă se definesc punctele de intrare ca un nou tip de dată numit tipul de date funcţie, se construieşte un vector de pointeri:

(pointer, funcţie) pf [m] (8.47)

iniţializat astfel:

pf[i] = f( ); i=1, 2, …, m (8.48)

În loc să se scrie o secvenţa cu *m* apeluri de funcţii, se reduce totul la secvenţa, unde p1, p2,.., pn sunt vectori cu parametrii asociaţi funcţiilor f1, f2,…, fn:

```
for ( i = 0 ; i < m ; i + + )
      pf[i] (p1[i], p2[i], ..., pn[i]);
```

Deci vectorul de pointeri spre funcţii facilitează crearea unor secvenţe dinamice de apel în timpul execuţiei, prin variabilitatea indicelui *i*.

Dacă funcţiile de intrare se memorează într-o matrice de pointeri spre funcţie, atunci există o mai mare diversitate şi flexibilitate de prelucrare, ceea ce permite realizarea de sisteme de programe vecine prin complexitatea lor cu sistemele de program expert.

8.5 Variabile de tip pointer şi structurile de date de tip articol

Variabilele pointer, ca de altfel orice alt tip de variabilă, apar ca membri în structurile de date de tip articol. De asemenea, o variabilă pointer este definită ca pointer spre structură.

Construcţiile:

```
struct a
{
   int m1;
   double m2;
};
a x;
a *y;
```

definesc:

- *x* ca variabilă de tip articol;
- *y* ca variabilă de tip pointer spre articol.

Expresiile:

y = adr(x) (8.49)

ref(y).m1 (8.50)

$$ref(y).c = adr(z) \quad (8.51)$$

$$ref(y) = ref(c) \quad (8.52)$$

reprezintă modalităţile de iniţializare sau utilizare a membrilor structurii, în condiţiile în care elementul bază de referire este o variabilă pointer şi unul dintre membri este, de asemenea, tot o variabilă pointer.

Deosebirea este că *y* este un pointer spre structura *x*, iar *c* este un pointer spre întreg.

Lucrul cu fişiere presupune stocarea de informaţii privind caracteristicile fişierelor, precum şi informaţii de stare a prelucrării acestora. Informaţiile sunt neomogene şi pentru stocarea lor trebuie definite structuri corespunzătoare, care se constituie de fapt ca un vector de structură. Numărul de componente ale vectorului, indică numărul maxim de fişiere cu care se lucrează într-un program.

Acest vector de structură se pune în corespondenţă cu elementele ce se definesc în programele utilizatorilor.

Tipul de date *FILE* este de tip pointer spre o structură şi caracterul local sau global acordat variabilelor de acest tip permite definirea zonei program din care programatorul are acces prin operaţii de intrare/ieşire la fişier. Programatorul are acces la fişiere prin structurile de date de tip *FILE*.

Numeroşi parametri ai funcţiilor de lucru cu fişiere, sunt pointeri care au acelaşi tip cu membrii structurii de date *FILE*, parametri care permit iniţializări de membri, sau comparări care validează efectuarea de operaţii.

Dacă fişierul însuşi este pus în corespondenţă cu un pointer spre *FILE*, acesta apare ca parametru într-o funcţie, ceea ce oferă caracter general aplicaţiilor. Când se spune că fişierul apare ca parametru într-o funcţie, realitatea este că pointerul variabilă pointer spre structura de tip *FILE*, care conţine descrierea fişierului cu care se doreşte să se lucreze în funcţie, se transmite ca parametru real.

Observăm că toate informaţiile despre toate entităţile, date, programe şi fişiere, se structurează adecvat şi devin resurse la dispoziţia programatorului.

Limbajele evoluate, precum C şi C++, sunt puternice prin multitudinea funcţiilor de bibliotecă pe care programatorii le apelează. Numeroase funcţii necesită definirea unor variabile în programe, de un tip derivat, definit ca global în fişiere, ce trebuie incluse în program.

Programatorul trebuie să cunoască aceste structuri, pentru a folosi informaţiile pe care la returnează funcţiile apelate.

Funcţiile, primesc ca parametrii reali variabile elementare, sau masive, sau structuri, sau pointeri. sau alte tipuri derivate şi/sau definite global.

Funcţiile returnează valori ce se stochează în variabile elementare, în structuri de tip articol, în date de tip pointer. Numărul valorilor returnate este unu. Pentru a obţine ca funcţia să returneze un masiv, uni- sau bidimensional, e suficient ca acesta să fie inclus într-o structură de tip articol.

Programatorul care face distincţie între toate tipurile de date prezentate până acum şi care e deprins să vehiculeze uşor definirile şi referirile acestora, prin funcţii de bibliotecă sau funcţii proprii are acces la absolut toate resursele unui sistem de calcul.

Se observă că:

- referirea unei variabile elementare se face prin nume;
- referirea unui masiv se face prin nume, iar al unui element al său prin nume urmat de o expresie indicială cuprinsă între *„[ ]”*;
- referirea unei structuri de tip articol se face prin nume, iar a unui membru indicând numele structurii separat de operatorul *„.”*, de numele membrului respectiv;
- referirea unei date de tip pointer se face prin nume, iar a variabilei a cărui adresă o conţine, printr-un operator de referire;
- referirea unui fişier se efectuează prin numele variabilei pointer spre tipul de date *FILE*; iniţializarea şi utilizarea acestei structuri este la dispoziţia funcţiilor destinate lucrului cu fişiere.

Comparând referirile, se observă că pentru a defini aceste tipuri de date sunt necesare informaţii care să indice tipul şi ordinea pe care componentele o au în cadrul structurilor ca entităţi efective, desfăşurate liniar şi contiguu în memorie.

8.6 Definirea şi utilizarea variabilelor pointer în limbajul C++

În limbajul C++, pentru tipul pointer, prin declaraţia *tip * nume*, se prelucrează tipul.

De exemplu:

```
int *px, *py ;
int x, y ;
```

Pentru iniţializarea variabilelor pointer, se foloseşte operatorul *„&”*. De exemplu:

```
px  = & x;
py  = & y;
```

Funcţiei de referire îi corespunde operatorul *„*”*. Instrucţiunea:

```
*px = *px + *py;
```

este echivalentă cu:

```
x = x+y;
```

şi are corespondent în consideraţiile anterioare:

$$ref\ (px) = ref\ (px) + ref\ (py) \qquad (8.53)$$

În cazul definirii pointerilor spre tipuri de date derivate, se folosesc construcţii precum:

```
typedef struct b
{
   int c;
```

```
   int d;
   int e;
   int * g;
};
typedef  b *a;

a x;
b y;
```

Expresiile:

```
x = & y;
x->c = 3
x->d = x-> c*5;
x->e = 1
x->g = Addr(x,e);
x->e =  x-> e +x-> g;
```

ilustrează modalităţi de referire a membrilor unei structuri.

În cazul definirii:

```
typedef  int c;
typedef  c * b;
typedef  b * a;

a x;
b y;
c z;
……
```

expresiile:

```
z  = 7;
y  = &z;
x  = &y;
cout << **x;
```

ilustrează modalităţi de lucru cu pointeri spre întreg, variabila *y*, şi cu pointeri spre pointeri spre întreg, variabila *x*.

Funcţiei $ref^k(px)$ îi corespunde construcţia ***… de k ori …***px*, iar funcţiei $adr^k(px)$, îi corespunde secvenţa:

```
p1 = &(x);
p2 = &(p1);
… … … … … …
pk = &(pk-1);
```

Instrucţiunile:

```
int * px;
int  x;
```

```
… … … … … …
px = &(x);
x = 7;
cout << *px
```

au acelaşi efect ca:

```
cout << x;
```

Aplicaţiile complexe necesită definirea de vectori de pointeri spre matrice, pointeri spre vectori de pointeri spre pointeri spre matrice, pointeri spre vectori de pointeri.

Programele de mai jos, realizează sumele elementelor a trei matrice, folosind diferite modalităţi de referire a elementelor, specificate la fiecare program prin comentariu.

```
//vectori de pointeri spre matrice
#include <iostream>

using namespace std;

typedef int mat[2][3];
typedef mat *pmat;
typedef pmat vecp[3];
vecp vp;
int i,j,k;
int s[3] = {0,0,0};
mat a = {1,1,1,2,2,2};
mat b = {3,3,3,4,4,4};
mat c = {5,5,5,6,6,6};
main()
{
      vp[0] = &a;
      vp[1] = &b;
      vp[2] = &c;
      for (k=0; k<3; k++)
      {
        for (i=0; i<2; i++)
              for (j=0; j<3; j++)
                    s[k]+=(*vp[k])[i][j];
             cout<<"\n Suma matricei "<<k<<" este "<<s[k];
      }
}
```

Codul sursă al programului care utilizează vectori de pointeri spre pointeri la o matrice este:

```
//vectori de pointeri spre pointeri la o matrice
#include <iostream>

using namespace std;

typedef int mat[2][3];
typedef mat * pmat;
typedef pmat vecp[3];
typedef pmat * vecpp[3];
vecp vp;
vecpp vpp;
```

```
int i,j,k;
int s[3] = {0,0,0};
mat a = {1,1,1,2,2,2};
mat b = {3,3,3,4,4,4};
mat c = {5,5,5,6,6,6};
main()
{
      vp[0] = &a;
      vp[1] = &b;
      vp[2] = &c;
        for (i=0; i<3 ;i++)
              vpp[i] = &vp[i];

      for (k=0 ; k<3 ; k++)
      {
        for (i=0; i<2; i++)
              for (j=0; j<3; j++)
                    s[k] + = (**vpp[k])[i][j];
             cout<<"\n Suma matricei "<<k<<" este "<<s[k];
      }
}
```

În programul:

```
//pointeri spre vectori de pointeri

#include <iostream>

using namespace std;

typedef vecp *pvec;
int * ppp;

vec a = {1,2,3,4,5};

vecp p;
pvec pp;
int i;

main()
{
      for (i=0; i<5; i++)
            p[i] = &a[i];
      ppp = a;
      pp = &p;
      for (i=0; i<5; i++)
      {
            cout<<"\n"<<*(ppp+i)<<" ** "<<(*p[i]);
            cout<<"\n"<<(*(*pp)[i]);
      }
}
```

se utilizează pointeri spre vectori de pointeri, iar în programul următor sunt folosiţi pointeri spre vectori de pointeri spre pointeri de matrice:

```
//pointeri spre vectori de pointeri spre pointeri la matrice

#include <iostream>

using namespace std;
```

```
typedef pmat vecp[3];
typedef pmat* vecpp[3];
typedef vecpp * pvecpp;

vecp vp;
vecpp vpp;
pvecpp pvpp;
int i,j,k;

int s[3] = {0,0,0};
mat a = {1,1,1,2,2,2};
mat b = {3,3,3,4,4,4};
mat c = {5,5,5,6,6,6};
main()
{
      vp[0] = &a;
      vp[1] = &b;
      vp[2] = &c;
      for (i=0; i<3; i++)
        vpp[i] = &vp[i];

      pvpp = &vpp;

      for (k=0; k<3; k++)
      {
          for (i=0; i<2; i++)
             for (j=0; j<3; j++)
                s[k] + = (**(*pvpp)[k])[i][j];
          cout<<"\n Suma matricei "<<k<<" este "<<s[k];
      }
}
```

Un exemplu de program în care sunt puse în evidenţă modalităţi de referire a unor structuri complexe, este considerat următorul:

```
#include <iostream>

using namespace std;

typedef int vec[5];
typedef int * pvec[5];
typedef pvec * ppvec;
typedef struct strz
{
      vec aa;
      pvec paa;
      ppvec pppa;
};
typedef strz * pstr;
typedef pstr ps[2];
typedef ps* pps;

int i,j;
strz st[2];
ps pst;
pps pss;

main()
{
```

```
    st[0].aa[0] = 100;
    st[0].aa[1] = 200;
    st[0].paa[0] = &st[0].aa[0];
    st[0].paa[1] = &st[0].aa[1];
    st[1].aa[0] = 300;
    st[1].aa[1] = 400;
    st[1].paa[0] = &st[1].aa[0];
    st[1].paa[1] = &st[1].aa[1];
    pst[0] = &st[0];
    pst[1] = &st[1];
    pss = &pst;
    st[0].pppa = &st[0].paa;
    st[1].pppa = &st[1].paa;

    for (i=0; i<2; i++)
      for (j=0; j<2; j++)
      {
        cout<<"\n*"<<st[i].aa[j]<<"**"<<*(pst[i]->paa[j]);
        cout<<"++"<<*((*pss)[i]->paa[j])<<"--"<<*((*pss)[i]->pppa[j]);
      }
}
```

Un aspect important în utilizarea variabilelor pointer îl reprezintă alocarea dinamică a memoriei.