

# 1

## TIPURI DINAMICE DE DATE LUCRUL CU ADRESE

Datele de tip static au caracteristici care limitează rezolvarea unor clase de probleme. În primul rând, spațiul de memorie aferent unor astfel de date se definește și se rezervă la dimensiune maximă, prestabilită, ca spațiu propriu care nu poate fi disponibilizat și nici împărțit cu alte date, chiar dacă, în momentul diverselor execuții ale programului, nu este în întregime utilizat (rezervare statică sau la momentul compilării). În al doilea rând, componentele structurilor statice ocupă locuri prestabilite în spațiul rezervat, determinate de relația de ordonare specifică fiecărei structuri. În al treilea rând, limbajul definește operațiile admise cu valorile componentelor, potrivit tipului de bază al structurii, astfel încât numărul maxim și ordinea componentelor structurii nu pot fi modificate. În aceste condiții, structurile statice sunt dificil de utilizat în rezolvarea problemelor care prelucrează mulțimi de date pentru care numărul și ordinea componentelor se modifică frecvent în timpul execuției programului. Pentru astfel de situații, limbajul PASCAL oferă posibilitatea utilizării datelor de tip dinamic, cărora li se pot alocă și elibera zone de memorie pe parcursul execuției programului.

### 1.1 Lucrul cu adrese în Pascal

Adresarea memoriei se realizează prin registre ale unității centrale, care au capacitatea de un cuvânt. La adresarea în modul *real*, pentru formarea unei adrese fizice din spațiul de 1Mo este necesară folosirea a două registre: de segment (*segment*), care conține adresa de început a segmentului, numită și adresa de bază; de deplasare (*offset*), care precizează distanța la care se află octetul adresat față de începutul segmentului. Astfel, orice adresă din memorie poate fi specificată în formatul **segment:offset** sau, în alți termeni, **bază:deplasare**. Întrucât deplasarea de 16 biți nu poate accesa o locație de memorie din afara domeniului  $0..2^{16}-1$ , rezultă că dimensiunea maximă a unui segment este de 64 Ko, restricție valabilă pentru orice produs utilizat sub MS-DOS. Memoria este împărțită în *paragrafe* de câte 16 octeți, iar fiecare segment începe la granița de paragraf, adică de la o adresă divizibilă cu 16. Într-un spațiu de 1Mo sunt  $2^{16}$  paragrafe, ceea ce înseamnă că adresa de început a unui segment, corespunzând unui număr de

paragraf, poate fi reprezentată ca o valoare pe 16 biți. În calculul adresei fizice pe 20 biți, se aplică următoarea relație:  $segment * 16 + offset$ , unde *segment* și *offset* desemnează conținutul registrelor de segment (un număr de paragraf), respectiv de deplasare. Adresele din segmentul curent se numesc *apropriate (near)*, iar cele din afara acestuia sunt *îndepărtate (far)*. Accesul la o adresă apropiată presupune doar schimbarea conținutului registrului de deplasare, în timp ce pentru o adresă îndepărtată trebuie schimbată atât valoarea registrului de segment, cât și a celui de deplasare.

În unitatea System sunt definite funcțiile **Seg(Var x):WORD, Ofs(Var x):WORD** care furnizează adresa de segment și deplasarea variabilei, procedurii sau funcției **x**. În Pascal există tipul de date *pointer*, memorat pe două cuvinte, în care cuvântul superior (*high*) conține partea de segment, iar cuvântul inferior (*low*) pe cea de deplasare asociate unei adrese. Pentru a se putea exemplifica modul de lucru cu adrese, se precizează faptul că:

- tipul *pointer* se definește prin construcția de forma  $\wedge tip$ ;
- adresarea indirectă a unei variabile se definește prin construcția *identificator* $\wedge$ ;
- referirea adresei unei variabile se definește prin construcția *@identificator*.

Programatorul trebuie să facă distincție între adresa și conținutul unei variabile, precum și între adresarea directă și cea indirectă. În exemplul de mai jos, liniile sursă numerotate de la 1 la 8 se generează următoarele tipuri de adresare: **1, 2, 4**: adresare directă pentru ambii operanzi și lucru cu conținut; **3**: adresare directă pentru **pa** și **a**, lucru cu conținut pentru **pa** și cu adresa **a**; **5**: adresare directă pentru **c** și indirectă pentru **pa**, lucru cu conținut; **6, 7, 8**: adresare indirectă și lucru cu conținut. De remarcat că **pa** este de tip *pointer*, **pa** $\wedge$  este de tip *REAL* iar **@a** este adresa **a** (are configurație de *pointer*).

În sintaxa *@identificator*, *identificator* se referă la o variabilă, procedură sau funcție. Efectul referirii *@identificator* este similar celui obținut prin funcția **Addr** definită în unitatea System astfel: **Addr(Var x):pointer**, unde **x** este variabilă, funcție sau procedură. Folosirea referirii *identificator* $\wedge$  presupune existența unei adrese valide în variabila *identificator*.

## Exemplu:

### 1.1.

```

VAR  a,b,c:REAL;      {Se rezervă câte 6 octeți pentru fiecare variabilă}
      pa,pb: ^REAL;    {Se rezervă câte 4 octeți pentru fiecare variabilă}

BEGIN
  a:=20;               {Se atribuie valoarea 20 variabilei de adresa a}      1
  b:=a;                {Se atribuie variabilei de adresa b, conținutul variabilei
                        de adresa a}                                         2
  pa:=@a;              {Se atribuie variabilei de adresa pa, adresa a}      3

  pb:=pa;              {Se atribuie variabilei de adresa pb, conținutul variabilei de
                        adresa pa}                                          4
  c:=pa $\wedge$ ;          {Se atribuie variabilei c, conținutul variabilei a cărei adresă
                        este memorată în pa; aceasta este adresare indirectă prin pa.
                        Lui c i se atribuie conținutul lui a (20)}          5

```

```

WriteLn( 'Valoarea ',pb^:10:2,');
    {Se scrie continutul variabilei a carei adresa este în pb }           6
WriteLn( 'Adresa fizica a lui A : ',seg(pb^),'::',ofs(pb^));
    {Se scrie adresa a, sub forma segment:deplasare}                     7
WriteLn( 'Adresa fizica a lui PB:',seg(pb),'::',ofs(pb));
    ⑧ {Se scrie adresa pb, sub forma segment:deplasare}                 8

```

## 1.2 Structura memoriei la execuția unui program

După încărcarea programului executabil, memoria aferentă lui se structurează în următoarele regiuni (segmente): segmentul prefix program, regiunea de cod, segmentul de date, stiva și zona heap (figura 1.1). Pentru adresarea acestora, unitatea centrală de prelucrare folosește registre specializate (tabelul 1.1), la unele dintre ele existând acces direct (vezi tipul Registers din unitatea DOS, anexa 1) sau indirect din programe Pascal.

**Tabelul 1.1** Registre de segment /deplasare și funcții standard asociate

Tipul segmentului	Registrul de segment	Registrul de deplasare	Seg:Ofs
Segment de cod	CS (CSeg)	IP	CS:IP
Segment de date	DS (DSeg)	SI	DS:SI
Segment de stivă	SS (SSeg)	SP (SPtr)	SS:SP (SSeg:SPtr)

**Observație:** *Cseg, DSeg, SSeg, SPtr* sunt funcții de tip WORD, nu au parametri și sunt definite în unitatea System.

- *Segmentul prefix al programului (PSP)* este o zonă de 256 de octeți constituită de MS-DOS la încărcarea în memorie a fișierului de tip .EXE. El conține informații necesare sistemului de operare pentru a transfera controlul către program, respectiv pentru a returna controlul către sistemul de operare la încheierea execuției acestuia. Adresa de segment este memorată în variabila publică **PrefixSeg**, de tip WORD, definită în unitatea System.

- *Regiunea de cod* este constituită din mai multe segmente de cod: unul corespunzând programului principal, respectiv câte unul pentru fiecare unitate referită în program. Primul segment de cod este cel asociat programului principal, urmat de cele ale unităților, în ordinea inversă specificărilor din clauza USES. Ultimul segment de cod, introdus implicit în orice program executabil, corespunde unității System, care conține biblioteca de subprograme standard referite la momentul execuției (*Run-time library*). În absența clauzei USES, zona de cod va conține două segmente: cel al programului principal și cel al unității System. Codul poate fi împărțit într-un număr oarecare de segmente, singura limitare fiind dată de memoria disponibilă. Registrul CS

conține adresa de start a instrucțiunilor programului, iar registrul **IP** (registru pointer al instrucțiunilor) precizează adresa următoarei instrucțiuni de executat. Programele Pascal nu au acces la registrul **IP**, dar valoarea curentă a registrului **CS** poate fi obținută cu funcția **CSeg**. Adresa de început a zonei de cod este CSeg:0000. Conținutul registrului **CS** se poate modifica pe parcursul execuției, depinzând de faptul că instrucțiunile executate sunt din programul principal sau din una dintre unități.

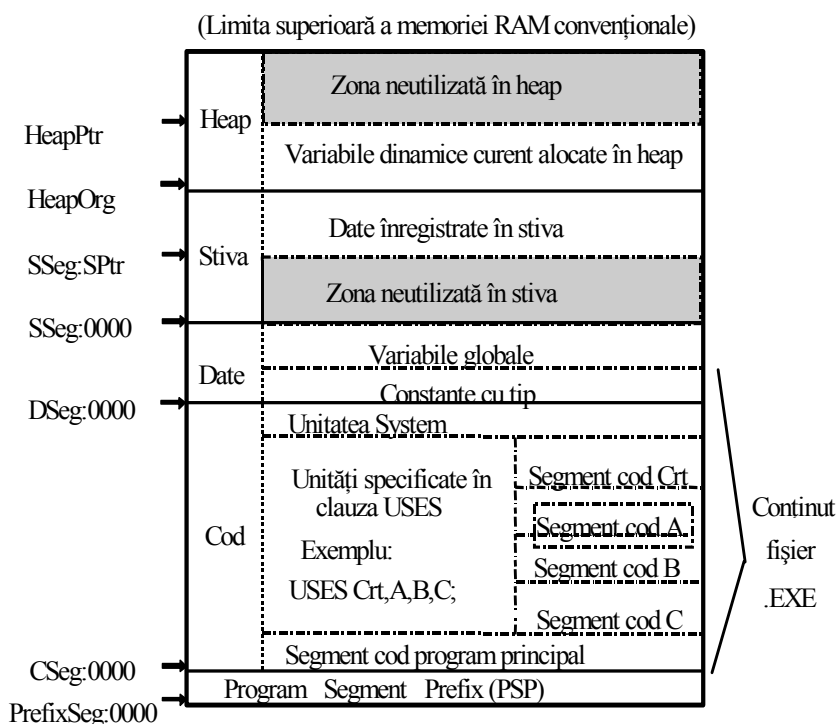


Fig. 1.1 Harta memoriei la execuția unui program Pascal

- *Segmentul de date* este unic și conține constantele cu tip urmate de variabilele globale. Atunci când necesarul de memorie pentru datele interne depășește 64Ko, trebuie să se recurgă la folosirea unor tehnici adecvate (memorarea datelor în *heap* sau pe medii externe, folosirea compactării etc.). Registrul **DS** conține adresa de început a segmentului de date și nu se modifică pe parcursul execuției. **SI** reprezintă *registrul index al sursei*, folosit pentru a puncta (a indexa) o dată în cadrul segmentului de date. Pentru instrucțiunile asupra șirurilor de caractere, registrul **SI** punctează pe operandul sursă, în timp ce un alt registru, *index al destinației* (**DI**), punctează operandul destinație. Funcția **DSeg** returnează valoarea curentă a registrului **DS**. Registrele **SI** și **DI** pot fi accesate indirect, printr-un apel la o întrerupere. Adresa de început a segmentului de date este DSeg:0000.

- *Segmentul de stivă*, ca și cel de date, poate avea maximum 64Ko, reducându-se la unul singur. Stiva este folosită în lucrul cu subprograme pentru memorarea parametrilor formali, variabilelor locale și adreselor de revenire. Registrul **SS** conține adresa de început a stivei și nu se modifică pe parcursul execuției. *Registrul pointer al stivei* (**SP**) precizează deplasarea curentă în cadrul stivei. Funcția **SSeg** returnează valoarea registrului **SS**, iar **SPtr** returnează valoarea curentă a registrului **SP**. În cadrul stivei, alocarea spațiului se face începând de la adrese mai mari spre adrese mai mici. Adresa curentă este definită de **SS:SP** sau, conform celor precizate anterior, de **SSeg:SPtr**.

- *Zona variabilelor dinamice* poate corespunde întregii memorii convenționale a calculatorului, rămasă disponibilă după încărcarea programului. În *heap* se memorează variabilele dinamice, buffer-ele pentru structuri de reacoperire și pentru lucrul în modul grafic. Adresa de început a zonei heap este dată de variabila publică **HeapOrg**, iar adresa curentă este dată de variabila **HeapPtr**, ambele de tip *pointer*, definite în unitatea System. Alocarea variabilelor începe de la adrese mai mici către adrese mai mari, spațiul maxim ce poate fi alocat unei variabile neputând depăși 64Ko (strict riguros, 65520 octeți), ca urmare a limitărilor impuse mecanismului de adresare a memoriei. După modul lor de funcționare, stiva și *heap*-ul pot fi asimilate cu două stive așezate spate în spate.

Programatorul poate controla repartizarea memoriei disponibile între stivă și heap în faza de execuție cu directiva de compilare `{M}`, care are următoarea formă sintactică:

`{M StackSize,HeapMin,HeapMax}`

*StackSize* trebuie să fie un întreg din domeniul 1024 (1Ko) la 65520 (64 Ko), prin care se specifică mărimea segmentului de stivă. *HeapMin* și *HeapMax* specifică dimensiunea minimă, respectiv maximă a heap-ului, teoretic cu valori între 0 și 640 Ko. Riguros, *HeapMin* poate avea valori de la 0 la 655360, iar *HeapMax* trebuie să fie în domeniul de la *HeapMin* la 655360. Valorile implicite pentru acești parametri de alocare sunt `{M 16384,0,655360}`.

Rezultă că dimensiunea implicită a stivei este de 16 Ko, iar zona de heap se extinde, teoretic, în tot spațiul rămas liber în memoria convențională. Practic, din dimensiunea de 640 Ko trebuie scăzut, pe lângă spațiul ocupat de programul însuși, cel corespunzător componentelor sistemului de operare rezidente în memorie pe parcursul execuției.

### 1.3 Tipuri dinamice de date

În Pascal se operează cu două tipuri de date dinamice - referință și pointer - primele fiind "cu tip" iar celelalte "fără tip".

- *Tipul referință* are sintaxa: ***tip\_referință*=^tip\_de\_bază;**. Simbolul ^ are semnificația de "indirect". Datorită asocierii cu un tip de bază, variabilele *tip\_referință* se mai numesc și *variabile cu referință legată*. La compilare, pentru astfel de variabile, se vor rezerva în segmentul de date două cuvinte și la referirea lor se vor genera

instrucțiuni cod mașină conform tipului de bază, dar cu adresare indirectă. Înainte de referire, în variabilele de *tip\_referință* trebuie să se încarce adresele variabilelor de tipul *tip\_de\_bază*. Declararea unui tip referință permite referirea anterior declarării tipului de bază. Astfel, următoarea secvență de declarări este corectă: TYPE pointer\_a=^vector; vector=ARRAY[1..20] OF REAL;

Construcția sintactică a referirii unei variabile dinamice depinde de caracteristicile tipului său de bază: este de forma **identificator**<sup>^</sup> în cazul tipurilor nestructurate sau celor structurate care permit referirea globală (STRING, RECORD și SET); conține prefixul **identificator**<sup>^</sup>, urmat de elementele specifice modului de referire a componentelor, în cazul tipurilor structurate care permit referirea pe componente (ARRAY, STRING și RECORD). Aceste posibilități de referire sunt ilustrate în programul *Alocare\_dinamică\_1*.

- *Tipul pointer* este desemnat prin cuvântul rezervat **pointer**. Variabilele de tip *pointer* pot fi denumite *variabile cu referință liberă*, deoarece pot fi folosite la memorarea adreselor pentru variabile de orice tip. Tehnica de lucru cu astfel de variabile este asemănătoare celei prezentate la tipul *referință*. Utilizarea efectivă presupune și în acest caz o asociere explicită cu un anumit tip de bază, dar soluția folosită este diferită. La tipul *referință*, asocierea se face prin declarare, iar în cazul tipului *pointer* asocierea se realizează la utilizare, prin diverse tehnici. O posibilitate este asigurată de referința **typecasting** (transfer de tip), care are forma generală: *tip(variabilă)*, unde *tip* este un tip standard sau declarat anterior de utilizator iar *variabilă* poate fi cu/fără tip sau o referință prin pointer, de forma *variabilă\_pointer*<sup>^</sup>.

Din punct de vedere fizic, variabilele de tip *referință\_legată* și *pointer* memorează adrese sub forma *segment:offset*. De aceea, în limbajul curent de specialitate, ambele tipuri se definesc prin termenul *pointer*. Următorul exemplu evaluează expresia **e:=a+b**, folosind adresarea indirectă pentru toate variabilele (**a** și **e** prin *referință\_cu\_tip* iar **b** prin *pointer*):

```
VAR
a,b,e:REAL;
pa,pe:^REAL;
pb:POINTER;
BEGIN
pa:=addr(a); pb:=@b; pe:=@e;
Write('A='); ReadLn(pa^);
Write('B='); ReadLn(REAL(pb^));
pe^:=pa^+REAL(pb^);
WriteLn('E= ',pe^:8:2);
END.
```

Variabilele *pointer* (*referință* sau *pointer*) pot fi puse în relație cu operatorii = și < >. Două variabile vor fi egale dacă au componentele *segment*, respectiv *offset* egale. De remarcat faptul că două variabile de tip *pointer* care indică aceeași adresă pot fi neegale, deoarece le diferă componentele. Variabilele *pointer* pot fi folosite în atribuirii. Atât în relații cât și în atribuirii sunt definite următoarele clase de compatibilitate: tipul *referință* este compatibil cu orice alt tip dinamic; două variabile de tip *referință* sunt compatibile dacă sunt de același tip.

**Observație:** **pa** și **pb** nu sunt de același tip dacă sunt declarate astfel: pa:^real; pb:^real.

Ele sunt de același tip dacă sunt declarate astfel: pa,pb:^real.

Este definită o constantă simbolică (**nil**) cu semnificație de valoare nulă a tipului dinamic (valoarea **nil** nu punctează o zonă de memorie).

## 1.4 Utilizarea zonei heap

Noțiunea de *dinamic* este strâns legată de utilizarea zonei de memorie *heap* (deși tipul dinamic poate fi asociat variabilelor memorate în orice componentă a memoriei principale). În unitatea System sunt definite următoarele variabile de tip pointer, care pot fi folosite în gestionarea zonei *heap*: HeapOrg, HeapPtr, HeapEnd, HeapError, FreeList.

**HeapOrg** punctează pe adresa de început a zonei *heap*, iar **HeapEnd** dă adresa de sfârșit a heap-ului. **HeapPtr** conține următoarea adresa disponibilă din *heap*. Ea este variabila prin care se gestionează fiecare nouă alocare, punctând pe prima adresă disponibilă din *heap*. Toate procedurile de alocare (**New**, **GetMem**, **Mark**) lucrează cu această variabilă. **HeapError** corespunde adresei rutinei de tratare a erorilor de alocare pentru variabile dinamice. **FreeList** servește la gestiunea blocurilor devenite libere în interiorul *heap*-ului, punctând pe primul bloc liber în *heap*, care punctează pe al doilea ș.a.m.d. Ultimul bloc liber punctează pe vârful *heap*-ului, adică pe locația dată de **HeapPtr**, asigurându-se astfel posibilitatea realocării acestor spații. Dacă în interiorul *heap*-ului nu există blocuri libere, atunci **FreeList** va fi egală cu **HeapPtr**.

De asemenea, în unitatea System sunt definite o serie de proceduri și funcții care pot fi utilizate în lucrul cu variabilele dinamice. Procedurile **GetMem(p,n)**, **FreeMem(p,n)**, respectiv **New(p)** și **Dispose(p)** se folosesc pentru a aloca/elibera un bloc a cărui adresă este dată de variabila pointer sau referință, **p**.

Deoarece zona *heap* este limitată, iar alocarea și eliberarea dinamică determină alternanța unor zone libere cu cele ocupate, este necesară cunoașterea spațiului disponibil și a spațiului contiguu maxim disponibil. În acest scop pot fi folosite funcțiile (fără argumente, cu rezultat de tip LONGINT) **MemAvail** (pentru spațiul total disponibil) și **MaxAvail** (pentru spațiul contiguu maxim disponibil). Inițial, rezultatul furnizat de **MemAvail** corespunde dimensiunii totale a *heap*-ului, care poate fi obținută și prin aplicarea formulei  $(\text{Seg}(\text{HeapEnd}^{\wedge}) - \text{Seg}(\text{HeapOrg}^{\wedge})) * 16$ , întrucât adresele de început și de sfârșit ale *heap*-ului sunt exprimate ca numere de paragraf. De asemenea, această dimensiune coincide inițial cu cea furnizată de funcția **MaxAvail**, care precizează cel mai lung bloc de locații de memorie contigue disponibile în *heap*. Se poate determina dacă spațiul disponibil este contiguu, pe baza expresiei relaționale **MemAvail = MaxAvail**. Dimensiunea în octeți ocupată de o variabilă poate fi obținută cu funcția **SizeOf(VAR x):WORD**, unde **x** este identificator de variabilă. Se poate stabili dacă spațiul disponibil este acoperitor pentru o variabilă de un anumit tip, scriind o relație de forma **MaxAvail >= SizeOf (tip)**.

- Alocarea și eliberarea zonelor pentru variabile referință\_legată se face cu

procedurile **New**, respectiv **Dispose**, definite în unitatea System astfel: **New(VAR p:pointer)**, **Dispose(VAR p:pointer)**. Procedura **New** rezervă în *heap* o zonă de memorie de lungime egală cu cea indicată de tipul de bază și încarcă în variabila **p** adresa acestei zone.

### Exemplu:

1.2.

VAR

px: ^INTEGER; {La compilare se rezerva 4 octeti in segmentul de date}

.....  
BEGIN

.....  
New(px); {La executie se rezerva 2 octeti si se incarca adresa zonei rezervate in  
**px**}

px:=21; {Se memoreaza valoarea 21 in zona de tip INTEGER din heap}

Dispose(px); {Se elibereaza spatiul rezervat in heap}

De remarcat că există două niveluri de rezervare: statică (corespunzătoare lui **px**) și dinamică (datorată procedurii **New(px)**, care rezervă variabilă în heap, în conformitate cu tipul de bază). Din punct de vedere fizic, operația realizează memorarea în variabila **p** a valorii **HeapPtr** și avansarea acesteia din urmă cu lungimea specifică tipului de bază. Dacă nu există o zonă contiguă disponibilă de lungime mai mare sau egală cu cea necesară, se generează eroare de execuție. Procedura **Dispose** eliberează zona alocată variabilei. Următorea alocare se poate face pe spațiul eliberat, dacă este acoperitor ca lungime. Fizic, se reface în **HeapPtr** valoarea de dinaintea alocării prin **New**.

În programul *Alocare\_dinamică\_1* se ilustrează prin exemple simple alocarea și eliberarea spațiului din *heap*, precum și referirile globale și pe componente pentru variabile dinamice ale căror tipuri de bază sunt nestructurate sau structurate. La execuția programului, pe lângă afișarea mesajelor care ilustrează corectitudinea folosirii tehnicii de lucru cu variabile dinamice, se remarcă revenirea în final, după eliberarea tuturor zonelor alocate, la dimensiunea inițială a *heap*-ului.

Prin posibilitatea de generare a unor succesiuni de valori de același tip, de la simpla alocare dinamică de spațiu pentru o singură valoare a unei variabile se poate trece la realizarea unor *structuri dinamice de date*. Acest lucru este posibil pe baza înlănțuirii succesiunilor de valori, ca urmare a includerii la fiecare element al structurii a două părți: o *parte de informații*, corespunzând valorii propriu-zise a elementului; o *parte de legătură*, care va conține adresa următorului element.

Fără a se urmări, în acest context, abordarea problematicei implementării structurilor de date complexe (liste, stive, cozi, arbori binari etc.), în programul *Alocare\_dinamică\_2* se prezintă realizarea unei liste înlănțuite pentru memorarea în *heap* a unui vector. Aplicația are în vedere memorarea în *heap* a unui vector de mari dimensiuni, cu elemente de tip întreg. Pentru fiecare element se construiește o structură de tip RECORD, în forma asociată tipului **element**. Pentru simplificare, se cere utilizatorului să precizeze numărul total de valori și valoarea de start, elementele fiind generate automat, ca numere întregi consecutive. După construire se reia traversarea



listei, cu afișarea pe monitor a elementelor vectorului.

Prin procedura **New(leg)** se va alocă o zonă de memorie de dimensiunea unei date de tipul **element**, iar în variabila de tip *leg* se va memora adresa de început a acestei zone. Zona de memorie a cărei adresă este conținută în variabila de tip *leg* va fi referită prin *indirectare*, folosind o construcție sintactică formată din - sau începând cu - variabila de tip *leg*<sup>^</sup>.

```
PROGRAM Alocare_dinamica_1;
USES Crt;
TYPE
  pv=^v;
  v=ARRAY[1..100] OF INTEGER;
  ps=^s;
  s=STRING[120];
  pm=^m;
  m=SET OF CHAR;
  pc=^CHAR;
  pa=^a;
  a=RECORD
    nume:STRING[15];
    nota:ARRAY[1..5] OF 1..10;
  end;
VAR
  legv:pv; legs:ps; legm:pm; legc:pc; lega:pa;
BEGIN
  ClrScr;
  WriteLn('Adresa de inceput heap: ',
    Seg(HeapOrg^),':',Ofs(HeapOrg^));
  Writeln('Adresa de sfirsit heap: ',
    Seg(HeapEnd^),':',Ofs(HeapEnd^));
  WriteLn('Valoare pointer heap: ',
    Seg(HeapPtr^),':',Ofs(HeapPtr^));
  WriteLn('Dimensiune totala heap: ',MemAvail);
  {Rezultat echivalent: (Seg(HeapEnd^)-Seg(HeapOrg^))*16 }
  WriteLn('Bloc maxim in heap: ',MaxAvail);
  New(legv);
  WriteLn('Memorie alocata in heap: ',SizeOf(legv^));
  WriteLn('Memorie libera in heap: ',MemAvail);
  legv^[1]:=1;
  WriteLn('v[1]=' ,legv^[1]);
  Dispose(legv);
  WriteLn('Memorie libera in heap: ',MemAvail);
  New(legs); legs^:='PASCAL';
  WriteLn('Al treilea caracter din sir este ',legs^[3]);
  New(legm); legm^:=['a'..'c'];
  IF 'c' IN legm^ THEN Writeln('Litera "c" apartine multimii');
  New(legc); legc^:=#65; Writeln('Caracter atribuit: ',legc^);
  New(lega); lega^.nume:='POPESCU ION'; lega^.nota[2]:=10;
  WriteLn('Studentul ',lega^.nume,' are nota ',lega^.nota[2], '
    la disciplina a doua');
  Dispose(lega); Dispose(legs); Dispose(legc); Dispose(legm);
  WriteLn('Memorie libera in heap: ',MemAvail);
END.
```

Din analiza programului *Alocare dinamica\_2* se pot desprinde câteva din cerințele impuse programatorului în construirea și prelucrarea structurilor dinamice. Astfel, pentru a face posibilă traversarea ulterioară, se memorează în variabila de tip referință **inceput** adresa primului element. La construirea unui nou element, inclusiv a primului, câmpul de legătură **urm** (de tip referință) este inițializat cu valoarea **nil**; la

traversarea ulterioară, regăsirea unei valori **nil** corespunde identificării ultimului element din listă. Pentru toate elementele, începând cu al doilea, construirea presupune revenirea la elementul anterior, a cărui informație de legătură trebuie să punteze pe elementul nou construit. Această balansare este asigurată prin “jocul” variabilelor de tip referință **curent**, respectiv **următor**.

```
PROGRAM Alocare_dinamica_2;
leg=^element;
element=RECORD
    v:INTEGER;
    urm:leg;
END;
VAR
    inceput,curent,urmator: leg;
    n,vi,i:INTEGER;
BEGIN
    Write('Nr. de elemente si valoare de inceput: ');
    ReadLn(n,vi);
    New(inceput);
    inceput^.v:=vi;
    inceput^.urm:=nil;
    WriteLn('S-a creat elementul 1 = ',inceput^.v);
    curent:=inceput;
    FOR i:=2 TO n DO
        BEGIN
            New(urmator);
            urmator^.v:=curent^.v+1;
            urmator^.urm:=nil;
            WriteLn('S-a creat elementul ',i,' = ',urmator^.v);
            curent^.urm:=urmator;
            curent:=urmator
        END;
        i:=i+1;
        curent:=inceput;
    WHILE curent^.urm <> nil DO
        BEGIN
            WriteLn('v[' ,i ,']: ',curent^.v);
            curent:=curent^.urm;
            Inc(i)
        END;
        Writeln('v[' ,i ,']: ',curent^.v)
    END.
```

• *Alocarea și eliberarea zonelor pentru variabile de tip **pointer*** se realizează cu procedurile **GetMem**, respectiv **FreeMem**, definite în unitatea System astfel: **GetMem(VAR p:pointer; l:WORD)**, **FreeMem(VAR p:pointer; l:WORD)**. Efectul acestora este asemănător procedurilor New și Dispose, cu precizarea că este necesară specificarea lungimii, care nu poate fi dedusă implicit, neexistând un tip de bază. Un exemplu simplu de folosire a variabilelor pointer din heap este ilustrat în programul *Alocare\_dinamica\_3*.

```
PROGRAM Alocare_dinamica_3;
```

```
VAR
  p: POINTER;
BEGIN
  GetMem(p,6);
  REAL(p^):=5.25;
  WriteLn(REAL(p^):4:2);
  FreeMem(p,6)
END.
```

Reluând exemplul propus prin programul *Alocare\_dinamică\_2*, trecerea la folosirea variabilelor *pointer* presupune unele adaptări, ilustrate în programul *Alocare\_dinamică\_4*. Din program se constată că aplicarea tehnicii de *typecasting* permite atât referirea globală, cât și pe componente, ilustrată în acest caz pentru câmpurile unui articol. Spațiul alocat la un apel al procedurii **GetMem** corespunde dimensiunii articolului (6 octeți).

```
PROGRAM Alocare_dinamica_4;
TYPE
  element=RECORD
    v:INTEGER; urm:POINTER;
  END;
VAR
  inceput,curent,urmator: POINTER;
  n,vi,i : INTEGER;
BEGIN
  Write('Nr. de elemente si valoare de inceput: ');
  ReadLn(n,vi);
  GetMem(inceput,6);
  element(inceput^).v:=vi;
  element(inceput^).urm:=nil;
  WriteLn('S-a creat elementul 1 = ',element(inceput^).v);
  curent:=inceput;
  FOR i:=2 TO n DO
    BEGIN
      GetMem(urmator,6);
      element(urmator^).v:=element(curent^).v+1;
      element(urmator^).urm:=nil;
      WriteLn('S-a creat elementul ',i,' = ',
        element(urmator^).v);
      element(curent^).urm:=urmator;
      curent:=urmator
    END;
    i:=i+1;
    curent:=inceput;
  WHILE element(curent^).urm <> nil DO
    BEGIN
      WriteLn('v[' ,i ,']: ',element(curent^).v);
      curent:=element(curent^).urm;
      Inc(i)
    END;
    WriteLn('v[' ,i ,']: ',element(curent^).v)
  END.
```

• Ca alternative ale procedurilor **New**, **GetMem**, respectiv **Dispose**, **FreeMem**, pot fi folosite procedurile **Mark** și **Release**, definite în unitatea **System** astfel: **Mark(VAR p:pointer)**, **Release(VAR p:pointer)**. Procedura **Mark** memorează în variabila **p** valoarea din **HeapPtr**, iar procedura **Release** depune în

variabila **HeapPtr** conținutul variabilei **p**. Folosirea în pereche a procedurilor **Mark** și **Release** oferă posibilitatea ca, după diverse alocări, să se restabilească valoarea variabilei **HeapPtr** cu valoarea memorată prin **Mark**. Apelul *Release(HeapOrg)* aduce **HeapPtr** pe începutul zonei heap (eliberează zona).

În exemplul următor, se memorează în **y** adresa de la un moment dat a lui **HeapPtr** (fie ea **a**), se modifică **HeapPtr** prin rezervarea a 12 octeți (pentru **x2**, **x3**), se reface conținutul lui **HeapPtr** cu adresa **a**, ceea ce înseamnă că **x3**<sup>^</sup> se rezervă la această adresă (în locul variabilei **x2**<sup>^</sup>).

```
VAR
    x1,x2,x3,x4:^REAL;
    y:POINTER;
BEGIN
    New(x1); x1^:=12;
    Mark(y);           {Memorarea valorii HeapPtr in Y}
    New(x2); x2^:=10;
    New(x3); x3^:=34;
    Release(y);        {Reincarcarea valorii din Y in Heapptr}
    New(x4); x4:=46;   {Se va memora peste valoare 10 din x2}
    .....
```