

Les chaînes sont donc :

- statiques par leur taille maximale (le système réserve la place correspondant à la structure définie par le programmeur)
- dynamiques par leur utilisation. Le système gère en permanence la taille actuelle.

4.5.8 RESUME

Les tableaux:

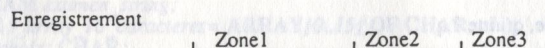
- utilisent la somme des places nécessaires pour chaque case
- sont stockés en mémoire en faisant varier d'abord l'indice déclaré en dernier (Ligne1, Colonne1), (Ligne1, Colonne2) ...)
- TURBO PASCAL ne sait pas compacter plusieurs cases dans un octet
- les tableaux de caractères sont compatibles avec les chaînes constantes de même longueur. Ils peuvent être initialisés par une seule instruction d'affectation, ou affichés par WRITE
- les chaînes de caractères (STRING) correspondent:
 - à une structure occupant un nombre fixe d'octets (l'octet de longueur plus le nombre de caractères précisé dans la déclaration)
 - dont la longueur est gérée dynamiquement (en même temps que l'exécution) par le système

4.6 LES ENREGISTREMENTS

4.6.1 DEFINITION

Les enregistrements sont constitués par une suite de zones. L'accès à la zone se fait en indiquant le nom de l'enregistrement suivi d'un point et du nom de la zone.

Schématiquement, nous dessinerons les enregistrements de la façon suivante:



4.6.2 IMPLEMENTATION DES ENREGISTREMENTS

Les enregistrements sont constitués par la juxtaposition de leurs zones.

L'enregistrement non compacté est donc uniquement un mécanisme:

- permettant de regrouper plusieurs variables sous un même nom au niveau du programme source
- permettant un certain nombre de traitements globaux (affectation, comparaison d'égalité ou de différence)

4.6.3 IMPLANTATION MEMOIRE DES ENREGISTREMENTS

Prenons le cas de

```

VAR personne: RECORD
  age: 0..120;
  prenom: STRING[4];
END;
  
```

Si nous affectons les valeurs 25 et JO, nous aurons en mémoire:

```

adresse      25
adresse+ 1   2
adresse+ 2   J
adresse+ 3   O
adresse+ 4   ?
adresse+ 5   ?
  
```

Voici un programme qui permet d'examiner un enregistrement en mémoire:

```

0  (* 001 affec *)
1  (* 08 avr 87 *)
2
3  (* $r+*)
4  PROGRAM examen_enregistrement;
5  TYPE t_array_16_caracteres= ARRAY[0..15] OF CHAR;
6  VAR choix: CHAR;
7      seize_A: t_array_16_caracteres;
8      enregistrement: RECORD
9          caractere: CHAR;
10         entier: INTEGER;
11     END;
12     seize_B: t_array_16_caracteres;
13
14  PROCEDURE affiche_memoire(p_segment, p_decalage_debut,
15
16  ...
17  END; (* affiche_memoire *)
18
19  PROCEDURE affiche_enregistrement;
20  BEGIN
21      WRITE(' zone caractere <a> ? ');
22      READLN(enregistrement.caractere);
23      WRITE(' zone entier <$77> ? ');
24      READLN(enregistrement.entier);
25
26      affiche_memoire(SEG(enregistrement), OFS(enregistrement)- 16,
27                     OFS(enregistrement)+ SIZEOF(enregistrement)+ 16- 1);
28  END; (* affiche_enregistrement *)
29
  
```



```

86  PROCEDURE initialise;
87  BEGIN
88    seize A:= 'AAAAAAAAAAAAAAAA';
89    seize B:= 'BBBBBBBBBBBBBBBB';
90  END; (* initialise *)
...

```

Fig. 4.34 Le programme *examen_enregistrement*.

Voici le résultat de l'exécution:

```

e(nregistrement, q(uitte ? e
  zone caractere <a> ? a
  zone entier <$77> ? $77

$153F:$0260          $41 $41 $41 $41 $41 : AAAAA
$153F:$0268 $41 $41 $41 $41 $41 $41 $41 : AAAAAAA
$153F:$0270 $41 $41 $41 : AAA
$153F:$0270          $61 $77 $00 : aw.
$153F:$0270          $42 $42 : BB
$153F:$0278 $42 $42 $42 $42 $42 $42 $42 : BBBBBBB
$153F:$0280 $42 $42 $42 $42 $42 : BBBBBB

e(nregistrement, q(uitte ? q

```

Fig. 4.35 Un enregistrement en mémoire

Notez que:

- l'ordre dans lequel les zones seront placées en mémoire ne dépend pas de l'utilisation d'une suite de zones ou de déclarations séparées. Les deux déclarations suivantes sont équivalentes au point de vue stockage mémoire:

```

VAR r_suite: RECORD
  a, b, c: CHAR;
END;

```

```

r_separe: RECORD
  a: CHAR;
  b: CHAR;
  c: CHAR;
END;

```

- TURBO ne sait pas compacter dans un octet plusieurs zones.

4.6.4 LES ENREGISTREMENTS VARIANTS

4.6.4.1 DEFINITION

Un enregistrement variant est un type permettant de manipuler une même zone de données comme une variable ayant des types différents.

Une variable est au départ un concept abstrait nous permettant de manipuler à l'aide d'un nom une zone mémoire ayant une certaine taille et située à un emplacement donné. Si, par exemple, nous déclarons:

```
VAR Total: INTEGER;
```

le compilateur met dans le code les informations correspondant à la zone qui sera allouée à cet entier. Supposons que ce soient les 2 octets à l'adresse 33004 et 33005 qui soient alloués à cette variable. Si plus loin nous exécutons:

```
Total:= 5;
```

le code du programme se charge de placer 5 à l'adresse 33004 et 0 à l'adresse 33005.

Une variable correspond donc à un concept abstrait dans la mesure où le programmeur n'a à se soucier ni de la taille ni de l'emplacement de ses variables:

- le compilateur se charge de gérer la taille nécessaire pour vous
- le code du programme et l'Operating System se chargent de placer les variables en mémoire, sans que vous ayez à vous en préoccuper

Le type que nous devons préciser au moment de la déclaration sert bien sûr au compilateur pour dimensionner la zone qu'il faudra utiliser pour la variable, mais permettra aussi d'effectuer des contrôles à la compilation et à l'exécution.

Or il existe des situations dans lesquelles nous souhaiterions pouvoir manipuler dans une même expression des types habituellement incompatibles:

- lorsqu'un enregistrement contient une zone dont le type n'est pas connu a priori par le programmeur
- lorsque nous voulons utiliser une procédure pour plusieurs types de données
- lorsque nous souhaitons utiliser les opérations valables pour un type donné sur une structure d'un autre type

Les enregistrements variants ont été créés pour répondre au premier de ces besoins. Ils peuvent être utilisés pour les deux autres, avec certains risques. Ces utilisations sont cependant la clé d'extensions des possibilités de PASCAL, nécessaires pour effectuer de la programmation système.

4.6.4.2 EXEMPLE D'ENREGISTREMENT VARIANT

Nous allons examiner le cas "naturel" ayant conduit à la création de l'enregistrement variant.

Nous souhaitons gérer les personnes d'une entreprise. Notre entreprise comporte par exemple des ouvriers, des contremaîtres et des cadres. Pour chaque personne, nous décrirons:

- son nom
- son âge
- AU CAS OU la fonction est
 - un ouvrier: sa qualification (OS, OHQ etc.)
 - un agent de maîtrise: son coefficient (PARODI ou autre)
 - un cadre: son niveau (niveau 1, niveau 2...) et le nombre de personnes sous ses ordres

Pour gérer cette population comportant plusieurs sous-populations, nous avons 3 solutions possibles:

- utiliser une fiche contenant les informations concernant tous les (3) cas possibles:

VAR *personne*: FILE OF RECORD

```
nom: STRING[3];
age: BYTE;
fonction: (ouvrier, maitrise, cadre);
qualification: (OS, OQ1, OQ2);
coefficient: INTEGER;
niveau: BYTE;
effectifs: INTEGER;
END;
```

ce que nous pouvons dessiner schématiquement ainsi, pour JOE, 33 ans, ouvrier OQ2:

personne:	JOE	33	ouvrier	OQ2	xxx	xxx	xxx
	nom	age	fonction	qualification	coefficient	niveau	effectifs

ou pour SAM, 38 ans, agent de maîtrise coefficient 330:

personne:	SAM	38	maitrise	xxx	330	xxx	xxx
	nom	age	fonction	qualification	coefficient	niveau	effectifs

ou pour AL, 42 ans, cadre niveau 15, dirigeant 63 personnes:

personne:	AL	42	cadre	xxx	xxx	15	63
	nom	age	fonction	qualification	coefficient	niveau	effectifs

Cette solution:

- consomme systématiquement trop de place
- risque de donner lieu à des incohérences

- Nous pouvons aussi créer autant de fichiers que de populations distinctes:

VAR *ouvriers*: FILE OF RECORD

```
nom: STRING[3];
age: BYTE;
qualification: (OS, OQ1, OQ2);
END;
```

maitrise: FILE OF RECORD

```
nom: STRING[3];
age: BYTE;
coefficient: INTEGER;
END;
```

cadre: FILE OF RECORD

```
nom: STRING[3];
age: BYTE;
niveau: BYTE;
effectifs: INTEGER;
END;
```

ce que nous pouvons dessiner schématiquement ainsi, pour JOE, 33 ans, ouvrier OQ2:

ouvrier:	JOE	33	OQ2
	nom	age	qualification

ou pour SAM, 38 ans, agent de maîtrise coefficient 330:

maitrise:	SAM	38	330
	nom	age	coefficient

ou pour AL, 42 ans, cadre niveau 15, dirigeant 63 personnes:

cadre:	AL	42	15	63
	nom	age	niveau	effectifs

Cette solution:

- n'utilise apparemment que la place nécessaire pour chaque cas (apparemment, car suivant l'Operating System utilisé, il y aura en moyenne plus de blocs inutilisés sur la disquette avec 3 fichiers séparés qu'avec un fichier unique)
- alourdira le programme (gestion simultanée de trois fichiers)

- Finalement, nous pouvons utiliser un enregistrement variant:


```

TYPE t_fonction = (ouvrier, maitrise, cadre);
VAR enregistrement: RECORD
    nom: STRING[3];
    age: BYTE;
    CASE fonction: t_fonction OF
        ouvrier : (qualification: (OS, OQ1, OQ2));
        maitrise : (coefficient: INTEGER);
        cadre : (niveau: BYTE;
                  effectifs: INTEGER);
    END;

```

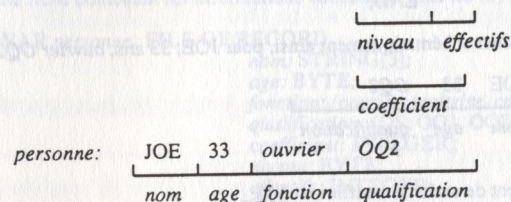
END;

Outre les informations communes, chaque enregistrement contient alors:

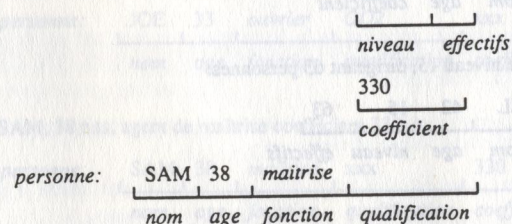
- soit la qualification
- soit le coefficient
- soit le niveau et les effectifs

et est muni d'un sélecteur, qui peut être utilisé pour documenter le choix pris.

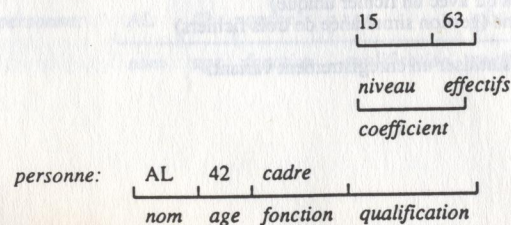
Cela peut être schématisé, pour JOE, 33 ans, ouvrier OQ2:



ou pour SAM, 38 ans, agent de maîtrise coefficient 330:



ou pour AL, 42 ans, cadre niveau 15, dirigeant 63 personnes:



L'enregistrement variant permet donc de réaliser l'union de plusieurs éventualités.

4.6.4.3 IMPLEMENTATION DES ENREGISTREMENTS VARIANTS

Un enregistrement variant est implémenté en utilisant:

- les bits nécessaires pour les zones communes (s'il en existe)
- les bits nécessaires pour le sélecteur
- les bits nécessaires pour pouvoir contenir la plus grande des variantes

Voici le contenu de la mémoire pour un variant similaire au précédent:

```

0 (* 001 affrecv *)
1 (* 08 avr 87 *)
2
3 (*$r+*)
4 PROGRAM examen_enregistrement_variant;
5 TYPE t_array_16_caracteres = ARRAY[0..15] OF CHAR;
6     t_fonction = (ouvrier, maitrise, cadre);
7 VAR choix: CHAR;
8     seize_A: t_array_16_caracteres;
9     enregistrement: RECORD
10         nom: STRING[3];
11         age: byte;
12         CASE fonction: t_fonction OF
13             ouvrier : (qualification: (OS, OQ1, OQ2));
14             maitrise : (coefficient: INTEGER);
15             cadre : (niveau: byte;
16                     effectifs: INTEGER);
17         END;
18     seize_B: t_array_16_caracteres;
19
20 PROCEDURE affiche_memoire(p_segment, p_decalage_debut,
21 ...
22 END; (* affiche_memoire *)
23
24 PROCEDURE affiche_enregistrement;
25 VAR choix1: CHAR;
26 BEGIN
27     WITH enregistrement DO
28     BEGIN
29         WRITE(' nom <JOE> ? '); READLN(nom);
30         WRITE(' age <33> ? '); READLN(age);
31         WRITE(' o(uvri)er, m(a)itrise, c(adre) ? ');
32         READ(KBD, choix1); WRITELN(choix1);
33         CASE choix1 OF
34             'o' : BEGIN
35                 fonction := ouvrier;
36                 WRITE(' O(S, OQ1(, OQ2( ? ');
37                 READ(KBD, choix1); WRITELN(choix1);

```



```

96 CASE choix1 OF
97   'O' : qualification:= OS;
98   '1' : qualification:= OQ1;
99   '2' : qualification:= OQ2;
100 END;
101 END;
102 'm' : BEGIN
103   fonction:= maitrise;
104   WRITE(' coefficient <155> ? ');
105   READLN(coefficient);
106 END;
107 'c' : BEGIN
108   fonction:= cadre;
109   WRITE(' niveau <12> ? '); READLN(niveau);
110   WRITE(' effectifs <53> ? '); READLN(effectifs);
111 END;
112 END; (* CASE choix1 *)
113 END; (* WITH enregistrement *)
114
115 affiche_memoire(SEG(enregistrement), OFS(enregistrement)- 16,
116   OFS(enregistrement)+ SIZEOF(enregistrement)+ 16- 1);
117 END; (* affiche_enregistrement *)
118
119 PROCEDURE initialise;
120 BEGIN
121   size_A:= 'AAAAAAAAAAAAAAAA';
122   size_B:= 'BBBBBBBBBBBBBBBB';
123 END; (* initialise *)

```

Fig. 4.36 Le programme *examen_enregistrement_variant*.

Et voici un exemple d'exécution:

```

e(nregistrement, q(uitte ? e
nom <JOE> ? JOE
age <33> ? 33
o(uvrier, m(aitrise, c(adre ? o
O(S, OQ1, OQ2 ? 2

$15B1:$0260          $41 $41 $41 $41 $41 : AAAAA
$15B1:$0268 $41 $41 $41 $41 $41 $41 $41 : AAAAAAAA
$15B1:$0270 $41 $41 $41 : AAA
$15B1:$0270          $03 $4A $4F $45 $21 : .JOE:
$15B1:$0278 $00 $02 $00 $00 : ....
$15B1:$278           $42 $42 $42 $42 : BBBB
$15B1:$0280 $42 $42 $42 $42 $42 $42 $42 : BBBBBBBB
$15B1:$0288 $42 $42 $42 $42 : BBBB

```

```

e(nregistrement, q(uitte ? e
nom <JOE> ? AL
age <33> ? 42
o(uvrier, m(aitrise, c(adre ? c
niveau <12> ? 15
effectifs <53> ? 63

$15B1:$0260          $41 $41 $41 $41 $41 : AAAAA
$15B1:$0268 $41 $41 $41 $41 $41 $41 $41 : AAAAAAAA
$15B1:$0270 $41 $41 $41 : AAA
$15B1:$0270          $02 $41 $4C $45 $2A : .ALE*
$15B1:$0278 $02 $0F $3F $00 : ...?
$15B1:$0278          $42 $42 $42 $42 : BBBB
$15B1:$0280 $42 $42 $42 $42 $42 $42 $42 : BBBBBBBB
$15B1:$0288 $42 $42 $42 $42 : BBBB

e(nregistrement, q(uitte ? q

```

Fig. 4.37 Un enregistrement variant en mémoire

Remarquons que l'utilisation du sélecteur n'est pas obligatoire:

- il n'intervient absolument pas dans l'accès. Nous pourrions d'ailleurs écrire, sans provoquer aucune erreur:

```

personne^fonction:= cadre;
personne^qualification:= OS;

```

- il peut servir pour stocker une information (ici la catégorie professionnelle) et pour gérer les différentes variantes. Nous pourrions par exemple utiliser le sélecteur pour afficher le contenu de l'enregistrement:

```

WITH personne DO
BEGIN
  WRITELN(nom:22, age: 3);
  CASE fonction OF
    ouvrier : CASE qualification OF
      OS : WRITELN('OS');
      OQ1 : WRITELN('OQ1');
      OQ2 : WRITELN('OQ2');
    END;
    maitrise : WRITELN('COEFFICIENT:', coefficient);
    cadre : WRITELN('NIVEAU ', niveau, ' EFFECTIFS ', effectifs);
  END;
END;

```



```

16  PROCEDURE affiche_octet_hexadecimal(p_octet: byte);
17  CONST ks_hex: ARRAY[0..15] OF CHAR= '0123456789ABCDEF';
18  BEGIN
19    WRITE(ks_hex[(p_octet AND $F0) SHR 4], ks_hex[p_octet AND $0F]);
20  END; (* affiche_octet_hexadecimal *)
21
22  PROCEDURE affiche_memoire(p_segment, p_decalage_debut,
...
76  END; (* affiche_memoire *)
77
78  PROCEDURE affiche_enregistrement;
79  BEGIN (* affiche_enregistrement *)
80    WRITE(' entier <$1234> ? '); READLN(enregistrement.entier);
81
82    affiche_memoire(SEG(enregistrement), OFS(enregistrement)- 16,
      OFS(enregistrement)+ SIZEOF(enregistrement)+ 16- 1);
83
84    WRITELN;
85    WITH enregistrement DO
86      BEGIN
87        affiche_octet_hexadecimal(tableau_octets[0]); WRITE(' ');
88        affiche_octet_hexadecimal(tableau_octets[1]); WRITELN;
89
90        WRITE(tableau_caracteres[0], tableau_caracteres[1]);
91      END;
92    WRITELN;
93  END; (* affiche_enregistrement *)
94
95  PROCEDURE initialise;
96  BEGIN
97    seize_A:= 'AAAAAAAAAAAAAAAA';
98    seize_B:= 'BBBBBBBBBBBBBBBB';
99  END; (* initialise *)
...

```

Fig. 4.38 Le programme *examen_variant_libre*.

Voici le contenu de la mémoire qui vous persuadera de l'absence du sélecteur:

```

e(nregistrement, q(uite ? e
entier <$1234> ? $2345

$155E: $0260      $41 $41 $41 $41 $41 :   AAAAA
$155E: $0268 $41 $41 $41 $41 $41 $41 $41 : AAAAAAAA
$155E: $0270 $41 $41 $41 :   AAA
$155E: $0270      $45 $23      :   E#
$155E: $0270      $42 $42 $42 :   BBB
$155E: $0278 $42 $42 $42 $42 $42 $42 $42 : BBBBBBBB
$155E: $0280 $42 $42 $42 $42 : BBBBB

```

```

45 23
E#
e(nregistrement, q(uite ? q

```

Fig. 4.39 Un enregistrement variant libre en mémoire

Puisque seul le type du sélecteur est utilisé, nous pouvons employer n'importe quel type scalaire (INTEGER, CHAR, BOOLEAN, énuméré, ou leurs intervalles). Traditionnellement pour de simples alternatives la littérature utilise un booléen. Par exemple:

```

VAR trix: RECORD
  CASE BOOLEAN OF
    TRUE : (nombre: INTEGER);
    FALSE : (caractère: CHAR);
  END;

```

Je vois mal ce que Nombre a de "vrai" par rapport à Caractère qui serait "faux". Nous préférons donc utiliser des entiers:

```

VAR trix: RECORD
  CASE BYTE OF
    1 : (nombre: INTEGER);
    2 : (caractère: CHAR);
  END;

```

ou encore:

```

TYPE t_donnee = (entier, lettre);
VAR trix: RECORD
  CASE t_donnee OF
    entier : (nombre: INTEGER);
    lettre : (caractère: CHAR);
  END;

```

Notez que:

- nous examinerons plus en détail l'utilisation des variants pour les conversions ou les paramètres ayant plusieurs types
- il existe d'autres moyens pour convertir des données ou pour éviter le contrôle des types des paramètres, que nous présenterons sous ces rubriques au chapitre 4

4.6.4.5 RISQUE DES ENREGISTREMENTS VARIANTS

Mentionnons dès à présent que l'utilisation de variants va à l'encontre de l'esprit de vérification à l'origine du concept de types. Outre les possibilités de manipuler les zones variantes sans aucune cohérence avec la valeur du sélecteur, signalons les erreurs de manipulations globales d'enregistrements variants:


```

VAR un, deux: RECORD
  CASE selecteur: INTEGER OF
    1: (age: INTEGER);
    2: (nom: STRING);
  END;

```

```

...
un.nom:= 'ZOE';
deux.nom:= 'ALI BABA';
un.age:= 16;
deux.age:= 16;

```

Considéré comme un enregistrement contenant des âges, le résultat de:

```
un= deux
```

devrait être **TRUE**. Malheureusement, il y a les octets résiduels du nom qui ne sont pas tous identiques, et le résultat sera, envers et contre tout, **FALSE**. A bon entendeur...

4.6.5 RESUME

- Le type enregistrement:
- est composé de zones qui peuvent avoir des types distincts
 - est implémenté en juxtaposant les bits nécessaires à chaque zone

Les enregistrements variants permettent d'utiliser les mêmes octets d'un enregistrement pour représenter différents types de données. La place utilisée est celle correspondant à la plus grande variante.

4.7 LES ENSEMBLES

4.7.1 DEFINITION

Les ensembles sont des types permettant essentiellement de vérifier l'appartenance d'une valeur à un ensemble de valeurs.

4.7.2 IMPLEMENTATION DES ENSEMBLES NUMERIQUES

Au point de vue implémentation, les ensembles sont constitués par des suites de bits, chaque bit correspondant à un élément possible de l'ensemble.

Prenons le cas de valeurs numériques d'abord:

```
VAR ensemble: SET OF 0..5;
```

Nous devons pouvoir placer dans notre ensemble les valeurs 0, 1, 2, 3, 4, 5. Par conséquent, le système a besoin de 6 bits.

L'instruction `ensemble:= []`; place 0 dans les 6 bits.

Numéro du bit:	54 3210
Valeur du bit:	00 0000
Qui représente l'élément:	54 3210

L'instruction `ensemble:= [3]` place un 1 dans le 4ième bit:

Numéro du bit:	54 3210
Valeur du bit:	00 1000
Qui représente l'élément:	54 3210

et l'instruction `ensemble:= [0, 3..5]` transforme l'ensemble en:

Numéro du bit:	54 3210
Valeur du bit:	11 1001
Qui représente l'élément:	54 3210

En **TURBO PASCAL**, le plus grand ensemble que nous puissions utiliser est l'ensemble `[0..255]`. Il utilise 256 bits, soit 32 octets.

Toutefois, lorsque la plus petite valeur de l'ensemble n'est pas zéro, il semble inutile d'utiliser des bits pour représenter les éléments compris entre 0 et la plus petite valeur de l'ensemble. Par exemple, pour représenter:

```
VAR ensemble2: SET OF 12..17;
```

nous pourrions utiliser 18 bits, mais les bits 0 à 12 ne seront jamais employés. En toute rigueur, 6 bits devraient suffire (un bit pour 12, un bit pour 13...), plus une information indiquant que le premier élément est 12.

TURBO PASCAL a choisi une méthode hybride: seuls les groupes de 8 bits contenant des valeurs possibles de l'ensemble sont utilisés.

Dans notre exemple,

Octet numéro:	0	1	2	3
Numéro du bit:	00000000	00111111	11112222	22222233
	01234567	89012345	67890123	45678901

élément possible: xxxx xx

seuls les bits de l'octet numéro 1 et numéro 2 sont utilisées. L'octet 0 et les octets au delà de l'octet 3 ne sont pas utilisés.

Cette technique:

- évite de consommer 32 octets pour chaque ensemble
- évite des décalages coûteux en temps de calcul lorsque le premier élément d'un ensemble n'est pas situé à la frontière d'un octet

Bien qu'il y ait un gâchis de place, celui-ci est limité à 14 bits (7 bits dans l'octet de départ, 7 dans l'octet final). Ceci serait le cas, par exemple, pour [15..16].

Notez que:

- lorsque nous plaçons dans l'ensemble un élément en dehors de la plage déclarée, mais qui peut être codé dans les bits utilisés, TURBO ne provoque pas d'erreur et place bel et bien l'élément. Ainsi:

```
VAR ensemble2: SET OF 12..17;
```

```
ensemble2:= [8];
IF 8 IN ensemble2
THEN WRITE('EH OUI !');
```

- lorsque nous plaçons dans l'ensemble une valeur qui ne peut pas être codée par les bits utilisés, TURBO PASCAL néglige cette instruction, sans provoquer aucune erreur:

```
VAR ensemble2: SET OF 12..17;
```

```
ensemble2:= [1];
IF ensemble2= []
THEN WRITE('MAIS SI !');
```

TURBO PASCAL tronque donc les valeurs avant de tenter de les placer dans les octets utilisés pour l'intervalle. Cette technique évite d'écraser des données situées avant ou après ceux de l'intervalle, mais oblige le programmeur à surveiller par programme les valeurs utilisées.

4.7.3 IMPLANTATION EN MEMOIRE DES ENSEMBLES

Les octets nécessaires à un ensemble sont placés, en mémoire, l'octet de poids le plus fort en dernier.

4.7.4 EXAMEN MEMOIRE DES ENSEMBLES

Voici le programme nous permettant d'examiner un ensemble en mémoire:

```
0 (* 001 affens *)
1 (* 08 avr 87 *)
2
3 (*$r+*)
4 PROGRAM examen_ensemble;
5   TYPE t_array 16 caracteres= ARRAY[0..15] OF CHAR;
6   VAR choix: CHAR;
7       avant: t_array 16 caracteres;
8       ensemble: SET OF BYTE;
9       apres: t_array 16 caracteres;
10
11  PROCEDURE affiche_memoire(p_segment, p_decalage_debut,
12
13    ...
14    END; (* affiche_memoire *)
15
16  PROCEDURE affiche_ensemble;
17    VAR l_choix1: CHAR;
18        l_valeur: INTEGER;
19        l_octet: INTEGER;
20
21  PROCEDURE affiche_bits(p_octet: byte);
22    VAR l_numero_bit: 0..7;
23        l_bit: 0..1;
24
25    BEGIN
26      FOR l_numero_bit:= 0 TO 7 DO
27        BEGIN
28          l_bit:= p_octet AND $01;
29          WRITE(l_bit);
30          p_octet:= p_octet SHR 1;
31        END;
32        WRITE(' ');
33      END; (* affiche_bits *)
34
35  BEGIN (* affiche_ensemble *)
36    ensemble:= [];
37    REPEAT
38      WRITE(' element <$77> ? ');
39      READLN(l_valeur);
40      ensemble:= ensemble+ [l_valeur];
41
42      WRITE(' continuer, q(uitter ? ');
43      READ(KBD, l_choix1); WRITELN(l_choix1);
44      UNTIL l_choix1= 'q';
45
46      affiche_memoire(SEG(ensemble), OFS(ensemble)- 16,
47        OFS(ensemble)+ SIZEOF(ensemble)+ 16- 1);
48
49    FOR l_octet:= 0 TO SIZEOF(ensemble)- 1 DO
50      BEGIN
51        affiche_bits(mem[SEG(ensemble):OFS(ensemble)+ l_octet]);
52        IF (l_octet+ 1) MOD 8= 0
53          THEN WRITELN;
54      END;
55      WRITELN;
56    END; (* affiche_ensemble *)
57
58  112
```



```

113  PROCEDURE initialise;
114  BEGIN
115    avant:= 'AAAAAAAAAAAAAAA';
116    apres:= 'BBBBBBBBBBBBBBB';
117  END; (* initialise *)

```

Fig. 4.40 Le programme *examen_ensemble*.

Notez que:

- nous avons pris le soin d'afficher les bits de l'ensemble dans l'ordre croissant du numéro de bit (en premier le bit 0, puis le bit 1...). Cette présentation n'est pas la présentation traditionnelle des bits d'un octet (normalement les bits d'un octet sont présentés dans l'ordre 7, 6, 5...), mais permet de mieux repérer les éléments de l'ensemble

Voici alors un exemple d'exécution:

```

e(nsemble, q(uitter ? e
element <$77> ? 1
continuer, q(uitter ? c
element <$77> ? 8
continuer, q(uitter ? c
element <$77> ? 9
continuer, q(uitter ? q

$157B:$0260          $41 $41 $41 $41 $41 : AAAAA
$157B:$0268 $41 $41 $41 $41 $41 $41 $41 : AAAAAAAA
$157B:$0270 $41 $41 $41 : AAA
$157B:$0270          $02 $03 $00 $00 $00 : .....
$157B:$0278 $00 $00 $00 $00 $00 $00 $00 : .....
$157B:$0280 $00 $00 $00 $00 $00 $00 $00 : .....
$157B:$0288 $00 $00 $00 $00 $00 $00 $00 : .....
$157B:$0290 $00 $00 $00 : ....
$157B:$0290          $42 $42 $42 $42 $42 : BBBBBB
$157B:$0298 $42 $42 $42 $42 $42 $42 : BBBBBBBB
$157B:$02A0 $42 $42 $42 : BBB

01000000 11000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000

e(nsemble, q(uitter ? q

```

Fig. 4.41 Un ensemble en mémoire

Voici un programme qui met en évidence la représentation d'un ensemble n'utilisant pas tous les 32 octets:

```

0  (* 001 affens2 *)
1  (* 08 avr 87 *)
2
3  (* $r+*)
4  PROGRAM examen_ensemble_2;
5  TYPE t_array_16_caracteres= ARRAY[0..15] OF CHAR;
6  VAR choix: CHAR;
7      avant: t_array_16_caracteres;
8      ensemble: SET OF 12..40;
9      apres: t_array_16_caracteres;
10
11  PROCEDURE affiche_memoire(p_segment, p_decalage_debut,
12
13  ...
14  END; (* affiche_memoire *)
15
16  PROCEDURE affiche_ensemble;
17  VAR l_choix1: CHAR;
18      l_valeur: INTEGER;
19      l_octet: INTEGER;
20
21  ...
22  PROCEDURE affiche_bits(p_octet: byte);
23  VAR l_numero_bit: 0..7;
24      l_bit: 0..1;
25
26  BEGIN
27    FOR l_numero_bit:= 0 TO 7 DO
28    BEGIN
29      l_bit:= p_octet AND $01;
30      WRITE(l_bit);
31      p_octet:= p_octet SHR 1;
32    END;
33    WRITE(' ');
34  END; (* affiche_bits *)
35
36  BEGIN (* affiche_ensemble *)
37    ensemble:= [];
38    REPEAT
39      WRITE(' element [12..40] ? ');
40      READLN(l_valeur);
41      ensemble:= ensemble+ [l_valeur];
42    UNTIL l_choix1='q';
43
44    affiche_memoire(SEG(ensemble), OFS(ensemble)- 16,
45      OFS(ensemble)+ SIZEOF(ensemble)+ 16- 1);
46
47    FOR l_octet:= 0 TO SIZEOF(ensemble)- 1 DO
48    BEGIN
49      affiche_bits(mem[SEG(ensemble):OFS(ensemble)+ l_octet]);
50      IF (l_octet+ 1) MOD 8= 0
51      THEN WRITELN;
52    END;
53    WRITELN;
54  END; (* affiche_ensemble *)

```



```

112
113 PROCEDURE initialise;
114 BEGIN
115     avant:= 'AAAAAAAAAAAAAAAA';
116     apres:= 'BBBBBBBBBBBBBBBB';
117 END; (* initialise *)
...

```

Fig. 4.42 Le programme *examen_ensemble_2*.

et voici un exemple d'utilisation:

```

e(nsemble, q(uitte ? e
  element [12..40] ? 12
  continuer, q(uitter ? c
  element [12..40] ? 32
  continuer, q(uitter ? c
  element [12..40] ? 33
  continuer, q(uitter ? q

$157C:$0260          $41 $41 $41 $41 $41 : AAAAA
$157C:$0268 $41 $41 $41 $41 $41 $41 $41 : AAAAAAAA
$157C:$0270 $41 $41 $41 $41 $41 : AAA
$157C:$0270          $10 $00 $00 $03 $00 : .....
$157C:$0278 $42 $42 $42 $42 $42 $42 $42 :BBBBBBBB
$157C:$0280 $42 $42 $42 $42 $42 $42 $42 :BBBBBBBB

00001000 00000000 00000000 11000000 00000000

e(nsemble, q(uitte ? q

```

Fig. 4.43 Un ensemble n'utilisant pas les 32 octets.

Et voici comment TURBO traite les éléments situés en dehors de l'intervalle:

```

e(nsemble, q(uitte ? e
  element [12..40] ? 1
  continuer, q(uitter ? q

$157C:$0260          $41 $41 $41 $41 $41 : AAAAA
$157C:$0268 $41 $41 $41 $41 $41 $41 $41 : AAAAAAAA
$157C:$0270 $41 $41 $41 $41 $41 : AAA
$157C:$0270          $00 $00 $00 $00 $00 : .....
$157C:$0278 $42 $42 $42 $42 $42 $42 $42 :BBBBBBBB
$157C:$0280 $42 $42 $42 $42 $42 $42 $42 :BBBBBBBB

00000000 00000000 00000000 00000000 00000000

```

```

e(nsemble, q(uitte ? e
  element [12..40] ? 12
  continuer, q(uitter ? q

$157C:$0260          $41 $41 $41 $41 $41 : AAAAA
$157C:$0268 $41 $41 $41 $41 $41 $41 $41 : AAAAAAAA
$157C:$0270 $41 $41 $41 $41 $41 : AAA
$157C:$0270          $01 $00 $00 $00 $00 : .....
$157C:$0278 $42 $42 $42 $42 $42 $42 $42 :BBBBBBBB
$157C:$0280 $42 $42 $42 $42 $42 $42 $42 :BBBBBBBB

10000000 00000000 00000000 00000000 00000000

e(nsemble, q(uitte ? e
  element [12..40] ? 47
  continuer, q(uitter ? q

$157C:$0260          $41 $41 $41 $41 $41 : AAAAA
$157C:$0268 $41 $41 $41 $41 $41 $41 $41 : AAAAAAAA
$157C:$0270 $41 $41 $41 $41 $41 : AAA
$157C:$0270          $00 $00 $00 $00 $80 : .....
$157C:$0278 $42 $42 $42 $42 $42 $42 $42 :BBBBBBBB
$157C:$0280 $42 $42 $42 $42 $42 $42 $42 :BBBBBBBB
00000000 00000000 00000000 00000000 00000001

e(nsemble, q(uitte ? e
  element [12..40] ? 1234
  continuer, q(uitter ? q

$157C:$0260          $41 $41 $41 $41 $41 : AAAAA
$157C:$0268 $41 $41 $41 $41 $41 $41 $41 : AAAAAAAA
$157C:$0270 $41 $41 $41 $41 $41 : AAA
$157C:$0270          $00 $00 $00 $00 $00 : .....
$157C:$0278 $42 $42 $42 $42 $42 $42 $42 :BBBBBBBB
$157C:$0280 $42 $42 $42 $42 $42 $42 $42 :BBBBBBBB
00000000 00000000 00000000 00000000 00000000

e(nsemble, q(uitte ? q

```

Fig. 4.44 Ajout de valeurs au delà de l'ensemble

4.7.5 IMPLEMENTATION D'ENSEMBLES NON NUMERIQUES

Pour les autres types simples scalaires (caractères, booléens, énumérés), TURBO utilise simplement la représentation de ces types pour construire l'ensemble:

- pour les ensembles de caractères (SET OF CHAR) les 256 caractères du code ASCII étendu sont des éléments possibles. Cet ensemble nécessite donc 256 bits soit 32 octets
- pour les ensembles d'intervalles de caractères (SET OF 'A'..'B' par exemple), la technique de représentation par octets complets est utilisée. Pour les lettres majuscules, il faut donc 4 octets
- pour les ensembles de booléens (SET OF BOOLEAN), il faut un octet

- pour les ensembles de valeurs de type énuméré, le nombre de bits utilisés correspond au nombre de valeurs de l'ensemble. Ainsi, si nous déclarons:

TYPE *t_jour* = (Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche);
SET OF *Mardi.Samedi*;

nous utiliserons 5 bits, qui seront contenus dans un octet.

4.7.6 RESUME

Le type ensemble:

- permet de tester l'appartenance d'une ou plusieurs valeurs à une structure.
 - est implémenté en utilisant la représentation binaire des éléments de l'ensemble.
- Le plus grand ensemble possible est l'ensemble des valeurs dans l'intervalle 0..255. Cet ensemble utilise 32 octets.

Les ensembles d'un sous-intervalle de 0..255 sont représentés en n'utilisant que les octets strictement nécessaires pour placer les éléments du sous-intervalle (les octets 0 à premier_élément DIV 8 et les octets dernier_élément DIV 8 + 1 à 31 ne sont pas nécessaires)

- les octets d'un ensemble sont disposés en mémoire dans l'ordre croissant

4.8 LE TYPE POINTEUR

4.8.1 DEFINITION

Du point de vue de l'implémentation, le type pointeur correspond à une mécanique d'adressage.

Un variable de type pointeur est une adresse qui permet d'accéder à une variable pointée.

TURBO PASCAL place les objets pointés dans l'espace situé entre la fin des variables globales et la fin de la pile des variables locales. Cet espace peut dépasser 64K, et TURBO utilise par conséquent des adresses segmentées. Chaque pointeur contient donc le segment et le décalage du début de la variable pointée.

4.8.2 LE MECANISME D'ALLOCATION

Pour expliquer la manière utilisée pour implémenter les pointeurs et les variables pointées, nous allons d'abord récapituler la mécanique utilisée pour les pointeurs.

Lorsque vous déclarez une variable de type pointeur, le compilateur réserve 4 octets pour contenir l'adresse segmentée.

En supposant que le pointeur est une variable globale, ces 4 octets seront alloués dans la pile des variables globales. Lorsque vous chargez le programme, ces 4 octets contiendront n'importe quelle valeur qui se trouvait à cet instant dans ces 4 octets. Un pointeur, pas plus que les autres variables PASCAL, n'est pas initialisé automatiquement. Le pointeur pointe donc vers n'importe quelle adresse de la mémoire.

Si par exemple vous déclarez

VAR *pointeur_nom*: ^STRING[5];

voici l'aspect de la mémoire après le chargement:

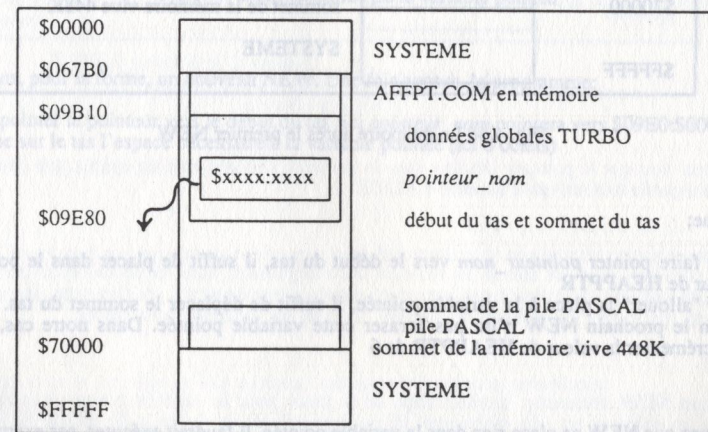


Fig. 4.45 La mémoire juste après le chargement

De plus une variable système, **HEAPPTR**, contient l'adresse du sommet du tas. C'est à cet endroit que sera placée la prochaine variable dynamique. Après le chargement, aucune variable n'a encore été placée sur le tas. Le début du tas et le sommet du tas ont la même valeur. Dans notre cas, la valeur de **HEAPPTR** est égale à \$09E8:\$0000.

Lorsque vous exécutez NEW(*pointeur_nom*), le programme:

- fait pointer le pointeur vers le début du tas (ici *pointeur_nom* pointe vers \$09E0:\$0000)
- alloue sur le tas l'espace nécessaire à la variable pointée (ici 6 octets)

Voici par conséquent la situation:

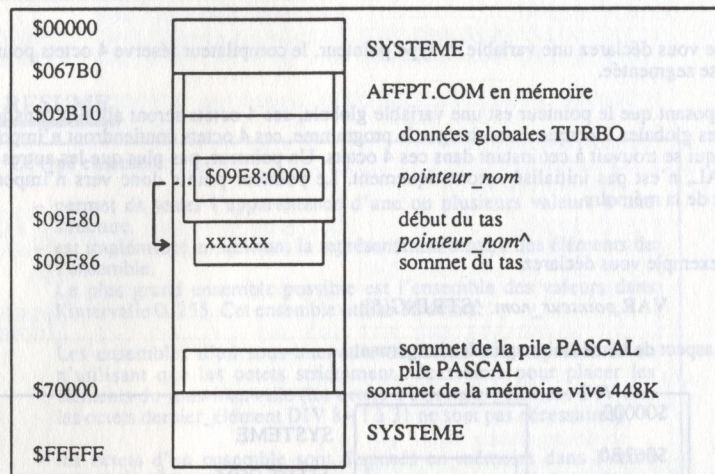


Fig. 4.46 La mémoire après le premier NEW

Notez que:

- pour faire pointer *pointeur_nom* vers le début du tas, il suffit de placer dans le pointeur la valeur de HEAPPTR
- pour "allouer" la place à la variable pointée, il suffit de déplacer le sommet du tas. De cette façon le prochain NEW n'ira pas écraser cette variable pointée. Dans notre cas, il suffit d'incrémenter la valeur de HEAPPTR de 6

Remarquez que NEW ne place rien dans la variable pointée. Il faudrait exécuter, par exemple:

pointeur_nom^:= 'ALICE';

et alors nous aurons:

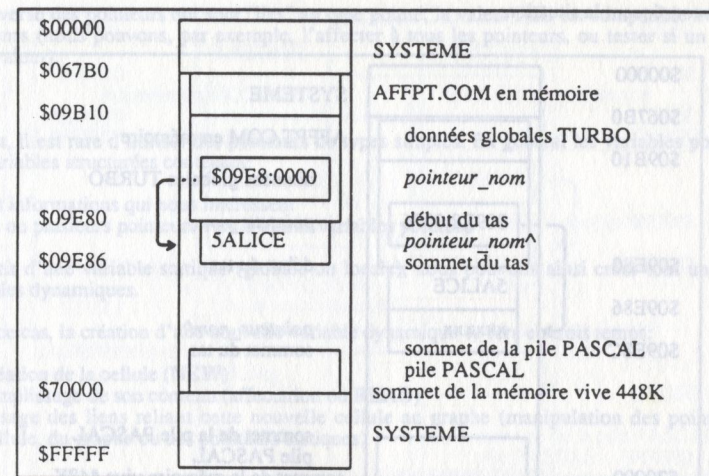


Fig. 4.47 L'initialisation de la variable pointée

Exécutons, pour la forme, un nouveau NEW. Une fois encore, le programme:

- fera pointer le pointeur vers le début du tas (ici *pointeur_nom* pointera vers \$09E0:\$0006)
- alloue sur le tas l'espace nécessaire à la variable pointée (ici 6 octets)

Voici par conséquent la situation:

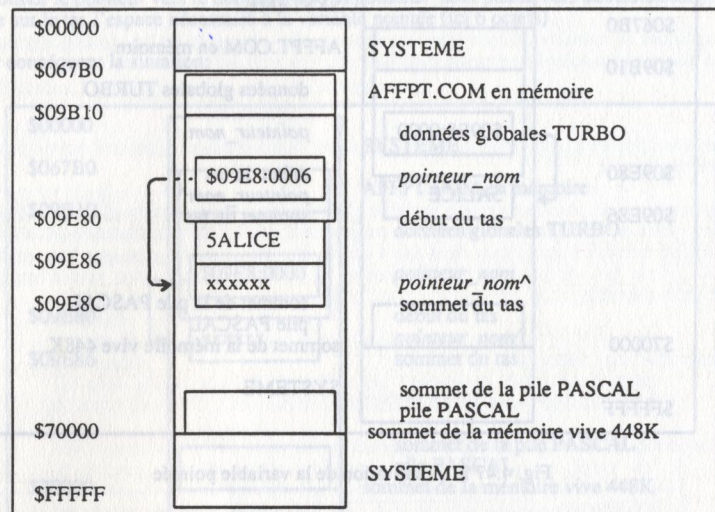


Fig. 4.48 L'effet d'un autre NEW

Remarquez bien que le pointeur désigne bien la nouvelle variable pointée, et que, par conséquent, nous avons perdu tout moyen d'accéder à ALICE.

Comme *pointeur_nom* est alloué au moment du chargement, nous la considérons comme une variable statique (l'allocation est déterminée par le programmeur). A l'inverse, les variables pointées ne sont créées qu'après le chargement. Ce sont des variables dynamiques.

Le fait que NEW déclenche la réservation de la place pour la variable pointée explique aussi pourquoi un pointeur est nécessairement lié au type pointé: un pointeur de **STRING[80]** devra réserver au moins 81 octets pour la variable pointée, alors qu'un pointeur d'**INTEGER** ne devra en réserver que 2. Il n'est donc pas possible de définir un pointeur vers n'importe quoi. Mentionnons que nous pouvons toutefois tourner cette contrainte:

- en utilisant des techniques qui nous libèrent des vérifications de type (enregistrements variants, variables **ABSOLUTE**...)
- en utilisant l'instruction **GETMEM** qui permet de préciser le nombre d'octets à allouer
- en gérant par programme le tas

A l'inverse des pointeurs qui sont "liés" au type pointé, la valeur **NIL** est compatible avec tous les pointeurs (nous pouvons, par exemple, l'affecter à tous les pointeurs, ou tester si un pointeur a cette valeur).

En fait, il est rare d'utiliser des pointeurs de types simples. En général les variables pointées sont des variables structurées contenant:

- les informations qui nous intéressent
- un ou plusieurs pointeurs vers d'autres variables pointées

A partir d'une variable statique (globale ou locale), nous pouvons ainsi créer tout un graphe de variables dynamiques.

Dans ce cas, la création d'une nouvelle variable dynamique se fera en trois temps:

- création de la cellule (**NEW**)
- remplissage de son contenu (affectation ou **READ**)
- tissage des liens reliant cette nouvelle cellule au graphe (manipulation des pointeurs de la cellule, du graphe ou des pointeurs statiques)

Par exemple, pour construire une structure de pile contenant dans chaque cellule 1 caractère, nous définissons:

```
TYPE t_pointeur = ^t_cellule;
t_cellule = RECORD
    caractere: CHAR;
    pointeur_suivante: t_pointeur;
END;
VAR racine: t_pointeur;
```

Rien n'est changé par rapport au pointeur de type élémentaire au moment du chargement:

- *racine* utilise 4 octets, alloués au moment du chargement sur la pile des variables globales

La création de la première cellule est aussi identique à la création précédente:

```
NEW(racine);
racine^.caractere := 'A';
racine^.pointeur_suivante := NIL;
```

Ici notre cellule utilise 5 octets (un pour *caractere* et 4 pour *pointeur_suivante*). Par conséquent nous aurons:

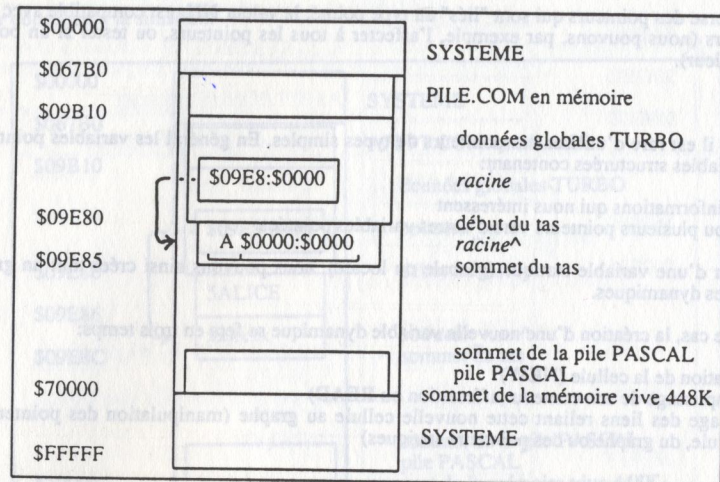


Fig. 4.49 Création de la première cellule

Pour ajouter une deuxième cellule qui pointerait vers celle-ci, nous ne pouvons utiliser `NEW(racine)`, car nous perdriions l'accès à la première cellule ! Nous devons donc utiliser une deuxième variable globale *pointeur_auxiliaire*. Le problème est un peu similaire à celui de l'échange des valeurs de 2 variables: il en faut une troisième.

Dans notre cas, la création se fait par:

```
NEW(pointeur_auxiliaire);
pointeur_auxiliaire^.caractere := 'B';
```

Soit:

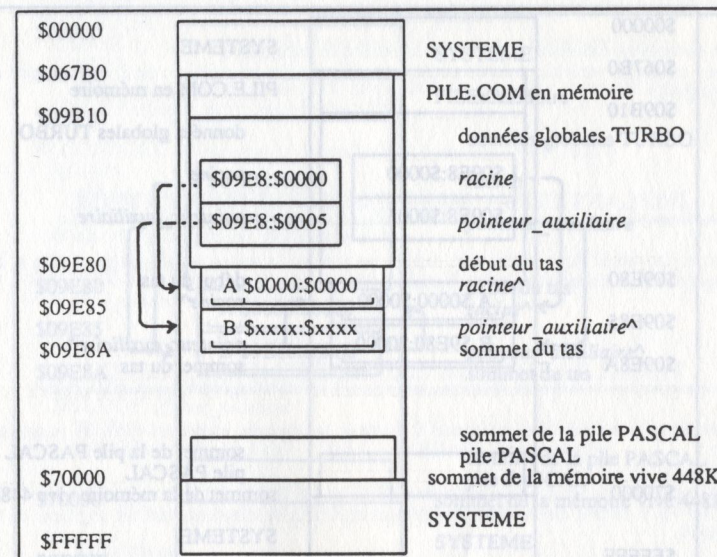


Fig. 4.50 Création de la seconde cellule

puis, nous faisons pointer la seconde cellule vers la précédente:

```
pointeur_auxiliaire^.pointeur_suivante := racine;
```

Soit:

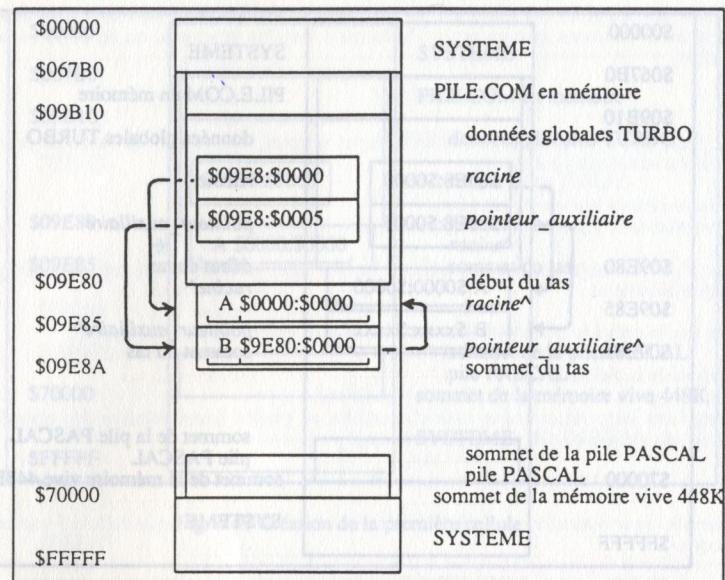
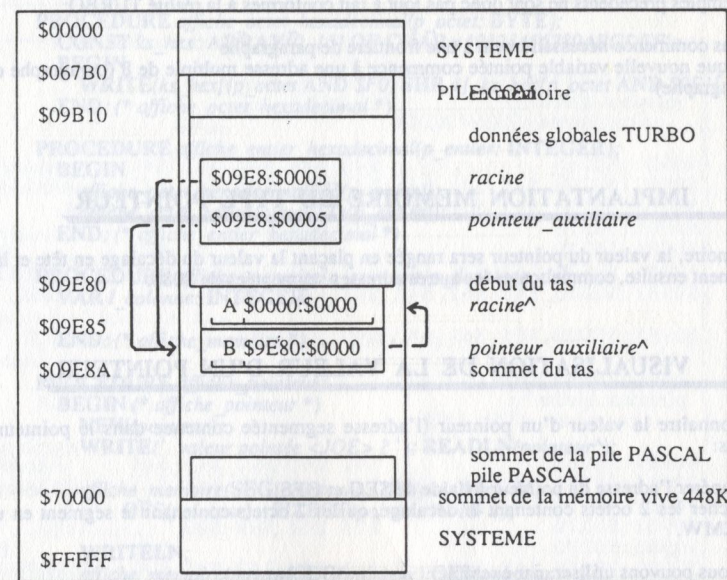


Fig. 4.51 Liaison entre les 2 cellules

Finalement nous faisons pointer *racine* vers la structure complète:

racine := *pointeur_auxiliaire*;

Soit:

Fig. 4.52 Et *racine* pointe vers la dernière cellule créée

4.8.3 IMPLEMENTATION DES POINTEURS

De façon interne, les pointeurs (aussi bien ceux qui sont statiques que ceux qui sont placés dans des variables dynamiques) correspondent à des adresses segmentées.

Ils utilisent donc 4 octets: 2 pour le segment et 2 pour le décalage. Ils peuvent désigner n'importe quelle adresse de la mémoire, de \$0000:\$0000 à \$FFFF:\$000F.

Par contre TURBO PASCAL alloue la place sur le tas par morceaux de 8 octets. La valeur d'un pointeur sera donc toujours \$xxxx:\$0000 ou \$xxxx:\$0008.

Le fait d'utiliser un décalage faible facilite l'accès aux octets d'une donnée pointée dont la taille peut ainsi atteindre 64K- 8 (il suffit de placer le segment du pointeur dans un registre et le décalage dans un autre registre).

Les exemples précédents ne sont donc pas tout à fait conformes à la réalité TURBO:

- le tas commence nécessairement à une frontière de paragraphe
- chaque nouvelle variable pointée commence à une adresse multiple de 8 (paragraphe ou demi-paragraphe)

4.8.4 IMPLANTATION MEMOIRE DU TYPE POINTEUR

En mémoire, la valeur du pointeur sera rangée en plaçant la valeur du décalage en tête et la valeur du segment ensuite, comme toutes les autres adresses segmentées du 8088.

4.8.5 VISUALISATION DE LA VALEUR D'UN POINTEUR

Pour connaître la valeur d'un pointeur (l'adresse segmentée contenue dans le pointeur), nous pouvons:

- récupérer l'adresse du pointeur à l'aide de SEG et OFS
- afficher les 2 octets contenant le décalage, ou les 2 octets contenant le segment en utilisant MEMW.

Ainsi nous pouvons utiliser, par exemple:

```
WRITE('$');
affiche_entier_hexadecimal(MEMW(SEG(pointeur):OFS(pointeur)+ 2));
WRITE(':');
affiche_entier_hexadecimal(MEMW(SEG(pointeur):OFS(pointeur)));
```

L'affichage de la valeur d'un pointeur est surtout intéressant pour la mise au point de programmes utilisant des pointeurs (lorsque le programme boucle indéfiniment par exemple).

4.8.6 EXAMEN DES POINTEURS

Voici un programme qui nous permet d'examiner un pointeur, ainsi que sa variable pointée:

```
0 (* 001 affpt *)
1 (* 08 avr 87 *)
2
3 (*$r*)
4 PROGRAM examen_pointeur;
5 TYPE t_array_16_caracteres= ARRAY[0..15] OF CHAR;
6 t_cellule= STRING[5];
7 VAR choix: CHAR;
8 avant: t_array_16_caracteres;
9 pointeur: ^t_cellule;
10 apres: t_array_16_caracteres;
```

```
11
12 PROCEDURE affiche_octet_hexadecimal(p_octet: BYTE);
13 CONST ks_hex: ARRAY[0..15] OF CHAR= '0123456789ABCDEF';
14 BEGIN
15 WRITE(ks_hex[(p_octet AND $F0) SHR 4], ks_hex[p_octet AND $0F]);
16 END; (* affiche_octet_hexadecimal *)
17
18 PROCEDURE affiche_entier_hexadecimal(p_entier: INTEGER);
19 BEGIN
20 affiche_octet_hexadecimal(HI(p_entier));
21 affiche_octet_hexadecimal(LO(p_entier));
22 END; (* affiche_entier_hexadecimal *)
23
24 PROCEDURE affiche_memoire(p_segment, p_decalage_debut, ...
25 VAR l_colonne: INTEGER;
26 ...
27 END; (* affiche_memoire *)
28
29 PROCEDURE affiche_pointeur;
30 BEGIN (* affiche_pointeur *)
31 NEW(pointeur);
32 WRITE(' valeur pointée <JOE> ? '); READLN(pointeur^);
33
34 affiche_memoire(SEG(pointeur), OFS(pointeur)- 16,
35 OFS(pointeur)+ SIZEOF(pointeur)+ 16- 1);
36
37 WRITELN;
38 affiche_memoire(memw[SEG(pointeur):OFS(pointeur)+ 2],
39 memw[SEG(pointeur):OFS(pointeur)],
40 memw[SEG(pointeur):OFS(pointeur)]+ SIZEOF(t_cellule)- 1);
41
42 END; (* affiche_pointeur *)
43
44 PROCEDURE initialise;
45 BEGIN
46 avant:= 'AAAAAAAAAAAAAAAA';
47 apres:= 'BBBBBBBBBBBBBBBB';
48 END; (* initialise *)
49
50 BEGIN (* principal *)
51 initialise;
52 REPEAT;
53 WRITELN; WRITE(' heapptr: $');
54 affiche_entier_hexadecimal(memw[SEG(heapptr): OFS(heapptr)+ 2]);
55 WRITE(':');
56 affiche_entier_hexadecimal(memw[SEG(heapptr): OFS(heapptr)]);
57 WRITELN;
58 WRITE(' p[ointeur, q(uitte ? ');
59 READ(KBD, choix); WRITELN(choix);
60 CASE choix OF
61 'p': affiche_pointeur;
62 'q': WRITELN;
63 'r': WRITELN;
64 's': WRITELN;
65 't': WRITELN;
66 'u': WRITELN;
67 'v': WRITELN;
68 'w': WRITELN;
69 'x': WRITELN;
70 'y': WRITELN;
71 'z': WRITELN;
72 '0'.. '9': WRITELN;
73 'A'.. 'Z': WRITELN;
74 'a'.. 'z': WRITELN;
75 ' ' : WRITELN;
76 '!' : WRITELN;
77 '@' : WRITELN;
78 '#' : WRITELN;
79 '$' : WRITELN;
80 '%' : WRITELN;
81 '^' : WRITELN;
82 '&' : WRITELN;
83 '*' : WRITELN;
84 '(' : WRITELN;
85 ')' : WRITELN;
86 '[' : WRITELN;
87 ']' : WRITELN;
88 '{' : WRITELN;
89 '}' : WRITELN;
90 ':' : WRITELN;
91 ';' : WRITELN;
92 ',' : WRITELN;
93 '.' : WRITELN;
94 '-' : WRITELN;
95 '_' : WRITELN;
96 '=' : WRITELN;
97 '+' : WRITELN;
98 '/' : WRITELN;
99 '*' : WRITELN;
100 ' ' : WRITELN;
101 '!' : WRITELN;
102 '@' : WRITELN;
103 '#' : WRITELN;
104 '$' : WRITELN;
105 '%' : WRITELN;
106 '^' : WRITELN;
107 '&' : WRITELN;
108 '*' : WRITELN;
109 '(' : WRITELN;
110 ')' : WRITELN;
111 '[' : WRITELN;
112 ']' : WRITELN;
113 '{' : WRITELN;
114 '}' : WRITELN;
115 ':' : WRITELN;
116 ';' : WRITELN;
117 ',' : WRITELN;
118 '.' : WRITELN;
119 '-' : WRITELN;
120 '_' : WRITELN;
121 '=' : WRITELN;
122 '+' : WRITELN;
123 '/' : WRITELN;
124 '*' : WRITELN;
125 ' ' : WRITELN;
126 '!' : WRITELN;
127 '@' : WRITELN;
128 '#' : WRITELN;
129 '$' : WRITELN;
130 '%' : WRITELN;
131 '^' : WRITELN;
132 '&' : WRITELN;
133 '*' : WRITELN;
134 '(' : WRITELN;
135 ')' : WRITELN;
136 '[' : WRITELN;
137 ']' : WRITELN;
138 '{' : WRITELN;
139 '}' : WRITELN;
140 ':' : WRITELN;
141 ';' : WRITELN;
142 ',' : WRITELN;
143 '.' : WRITELN;
144 '-' : WRITELN;
145 '_' : WRITELN;
146 '=' : WRITELN;
147 '+' : WRITELN;
148 '/' : WRITELN;
149 '*' : WRITELN;
150 ' ' : WRITELN;
151 '!' : WRITELN;
152 '@' : WRITELN;
153 '#' : WRITELN;
154 '$' : WRITELN;
155 '%' : WRITELN;
156 '^' : WRITELN;
157 '&' : WRITELN;
158 '*' : WRITELN;
159 '(' : WRITELN;
160 ')' : WRITELN;
161 '[' : WRITELN;
162 ']' : WRITELN;
163 '{' : WRITELN;
164 '}' : WRITELN;
165 ':' : WRITELN;
166 ';' : WRITELN;
167 ',' : WRITELN;
168 '.' : WRITELN;
169 '-' : WRITELN;
170 '_' : WRITELN;
171 '=' : WRITELN;
172 '+' : WRITELN;
173 '/' : WRITELN;
174 '*' : WRITELN;
175 ' ' : WRITELN;
176 '!' : WRITELN;
177 '@' : WRITELN;
178 '#' : WRITELN;
179 '$' : WRITELN;
180 '%' : WRITELN;
181 '^' : WRITELN;
182 '&' : WRITELN;
183 '*' : WRITELN;
184 '(' : WRITELN;
185 ')' : WRITELN;
186 '[' : WRITELN;
187 ']' : WRITELN;
188 '{' : WRITELN;
189 '}' : WRITELN;
190 ':' : WRITELN;
191 ';' : WRITELN;
192 ',' : WRITELN;
193 '.' : WRITELN;
194 '-' : WRITELN;
195 '_' : WRITELN;
196 '=' : WRITELN;
197 '+' : WRITELN;
198 '/' : WRITELN;
199 '*' : WRITELN;
200 ' ' : WRITELN;
201 '!' : WRITELN;
202 '@' : WRITELN;
203 '#' : WRITELN;
204 '$' : WRITELN;
205 '%' : WRITELN;
206 '^' : WRITELN;
207 '&' : WRITELN;
208 '*' : WRITELN;
209 '(' : WRITELN;
210 ')' : WRITELN;
211 '[' : WRITELN;
212 ']' : WRITELN;
213 '{' : WRITELN;
214 '}' : WRITELN;
215 ':' : WRITELN;
216 ';' : WRITELN;
217 ',' : WRITELN;
218 '.' : WRITELN;
219 '-' : WRITELN;
220 '_' : WRITELN;
221 '=' : WRITELN;
222 '+' : WRITELN;
223 '/' : WRITELN;
224 '*' : WRITELN;
225 ' ' : WRITELN;
226 '!' : WRITELN;
227 '@' : WRITELN;
228 '#' : WRITELN;
229 '$' : WRITELN;
230 '%' : WRITELN;
231 '^' : WRITELN;
232 '&' : WRITELN;
233 '*' : WRITELN;
234 '(' : WRITELN;
235 ')' : WRITELN;
236 '[' : WRITELN;
237 ']' : WRITELN;
238 '{' : WRITELN;
239 '}' : WRITELN;
240 ':' : WRITELN;
241 ';' : WRITELN;
242 ',' : WRITELN;
243 '.' : WRITELN;
244 '-' : WRITELN;
245 '_' : WRITELN;
246 '=' : WRITELN;
247 '+' : WRITELN;
248 '/' : WRITELN;
249 '*' : WRITELN;
250 ' ' : WRITELN;
251 '!' : WRITELN;
252 '@' : WRITELN;
253 '#' : WRITELN;
254 '$' : WRITELN;
255 '%' : WRITELN;
256 '^' : WRITELN;
257 '&' : WRITELN;
258 '*' : WRITELN;
259 '(' : WRITELN;
260 ')' : WRITELN;
261 '[' : WRITELN;
262 ']' : WRITELN;
263 '{' : WRITELN;
264 '}' : WRITELN;
265 ':' : WRITELN;
266 ';' : WRITELN;
267 ',' : WRITELN;
268 '.' : WRITELN;
269 '-' : WRITELN;
270 '_' : WRITELN;
271 '=' : WRITELN;
272 '+' : WRITELN;
273 '/' : WRITELN;
274 '*' : WRITELN;
275 ' ' : WRITELN;
276 '!' : WRITELN;
277 '@' : WRITELN;
278 '#' : WRITELN;
279 '$' : WRITELN;
280 '%' : WRITELN;
281 '^' : WRITELN;
282 '&' : WRITELN;
283 '*' : WRITELN;
284 '(' : WRITELN;
285 ')' : WRITELN;
286 '[' : WRITELN;
287 ']' : WRITELN;
288 '{' : WRITELN;
289 '}' : WRITELN;
290 ':' : WRITELN;
291 ';' : WRITELN;
292 ',' : WRITELN;
293 '.' : WRITELN;
294 '-' : WRITELN;
295 '_' : WRITELN;
296 '=' : WRITELN;
297 '+' : WRITELN;
298 '/' : WRITELN;
299 '*' : WRITELN;
300 ' ' : WRITELN;
301 '!' : WRITELN;
302 '@' : WRITELN;
303 '#' : WRITELN;
304 '$' : WRITELN;
305 '%' : WRITELN;
306 '^' : WRITELN;
307 '&' : WRITELN;
308 '*' : WRITELN;
309 '(' : WRITELN;
310 ')' : WRITELN;
311 '[' : WRITELN;
312 ']' : WRITELN;
313 '{' : WRITELN;
314 '}' : WRITELN;
315 ':' : WRITELN;
316 ';' : WRITELN;
317 ',' : WRITELN;
318 '.' : WRITELN;
319 '-' : WRITELN;
320 '_' : WRITELN;
321 '=' : WRITELN;
322 '+' : WRITELN;
323 '/' : WRITELN;
324 '*' : WRITELN;
325 ' ' : WRITELN;
326 '!' : WRITELN;
327 '@' : WRITELN;
328 '#' : WRITELN;
329 '$' : WRITELN;
330 '%' : WRITELN;
331 '^' : WRITELN;
332 '&' : WRITELN;
333 '*' : WRITELN;
334 '(' : WRITELN;
335 ')' : WRITELN;
336 '[' : WRITELN;
337 ']' : WRITELN;
338 '{' : WRITELN;
339 '}' : WRITELN;
340 ':' : WRITELN;
341 ';' : WRITELN;
342 ',' : WRITELN;
343 '.' : WRITELN;
344 '-' : WRITELN;
345 '_' : WRITELN;
346 '=' : WRITELN;
347 '+' : WRITELN;
348 '/' : WRITELN;
349 '*' : WRITELN;
350 ' ' : WRITELN;
351 '!' : WRITELN;
352 '@' : WRITELN;
353 '#' : WRITELN;
354 '$' : WRITELN;
355 '%' : WRITELN;
356 '^' : WRITELN;
357 '&' : WRITELN;
358 '*' : WRITELN;
359 '(' : WRITELN;
360 ')' : WRITELN;
361 '[' : WRITELN;
362 ']' : WRITELN;
363 '{' : WRITELN;
364 '}' : WRITELN;
365 ':' : WRITELN;
366 ';' : WRITELN;
367 ',' : WRITELN;
368 '.' : WRITELN;
369 '-' : WRITELN;
370 '_' : WRITELN;
371 '=' : WRITELN;
372 '+' : WRITELN;
373 '/' : WRITELN;
374 '*' : WRITELN;
375 ' ' : WRITELN;
376 '!' : WRITELN;
377 '@' : WRITELN;
378 '#' : WRITELN;
379 '$' : WRITELN;
380 '%' : WRITELN;
381 '^' : WRITELN;
382 '&' : WRITELN;
383 '*' : WRITELN;
384 '(' : WRITELN;
385 ')' : WRITELN;
386 '[' : WRITELN;
387 ']' : WRITELN;
388 '{' : WRITELN;
389 '}' : WRITELN;
390 ':' : WRITELN;
391 ';' : WRITELN;
392 ',' : WRITELN;
393 '.' : WRITELN;
394 '-' : WRITELN;
395 '_' : WRITELN;
396 '=' : WRITELN;
397 '+' : WRITELN;
398 '/' : WRITELN;
399 '*' : WRITELN;
400 ' ' : WRITELN;
401 '!' : WRITELN;
402 '@' : WRITELN;
403 '#' : WRITELN;
404 '$' : WRITELN;
405 '%' : WRITELN;
406 '^' : WRITELN;
407 '&' : WRITELN;
408 '*' : WRITELN;
409 '(' : WRITELN;
410 ')' : WRITELN;
411 '[' : WRITELN;
412 ']' : WRITELN;
413 '{' : WRITELN;
414 '}' : WRITELN;
415 ':' : WRITELN;
416 ';' : WRITELN;
417 ',' : WRITELN;
418 '.' : WRITELN;
419 '-' : WRITELN;
420 '_' : WRITELN;
421 '=' : WRITELN;
422 '+' : WRITELN;
423 '/' : WRITELN;
424 '*' : WRITELN;
425 ' ' : WRITELN;
426 '!' : WRITELN;
427 '@' : WRITELN;
428 '#' : WRITELN;
429 '$' : WRITELN;
430 '%' : WRITELN;
431 '^' : WRITELN;
432 '&' : WRITELN;
433 '*' : WRITELN;
434 '(' : WRITELN;
435 ')' : WRITELN;
436 '[' : WRITELN;
437 ']' : WRITELN;
438 '{' : WRITELN;
439 '}' : WRITELN;
440 ':' : WRITELN;
441 ';' : WRITELN;
442 ',' : WRITELN;
443 '.' : WRITELN;
444 '-' : WRITELN;
445 '_' : WRITELN;
446 '=' : WRITELN;
447 '+' : WRITELN;
448 '/' : WRITELN;
449 '*' : WRITELN;
450 ' ' : WRITELN;
451 '!' : WRITELN;
452 '@' : WRITELN;
453 '#' : WRITELN;
454 '$' : WRITELN;
455 '%' : WRITELN;
456 '^' : WRITELN;
457 '&' : WRITELN;
458 '*' : WRITELN;
459 '(' : WRITELN;
460 ')' : WRITELN;
461 '[' : WRITELN;
462 ']' : WRITELN;
463 '{' : WRITELN;
464 '}' : WRITELN;
465 ':' : WRITELN;
466 ';' : WRITELN;
467 ',' : WRITELN;
468 '.' : WRITELN;
469 '-' : WRITELN;
470 '_' : WRITELN;
471 '=' : WRITELN;
472 '+' : WRITELN;
473 '/' : WRITELN;
474 '*' : WRITELN;
475 ' ' : WRITELN;
476 '!' : WRITELN;
477 '@' : WRITELN;
478 '#' : WRITELN;
479 '$' : WRITELN;
480 '%' : WRITELN;
481 '^' : WRITELN;
482 '&' : WRITELN;
483 '*' : WRITELN;
484 '(' : WRITELN;
485 ')' : WRITELN;
486 '[' : WRITELN;
487 ']' : WRITELN;
488 '{' : WRITELN;
489 '}' : WRITELN;
490 ':' : WRITELN;
491 ';' : WRITELN;
492 ',' : WRITELN;
493 '.' : WRITELN;
494 '-' : WRITELN;
495 '_' : WRITELN;
496 '=' : WRITELN;
497 '+' : WRITELN;
498 '/' : WRITELN;
499 '*' : WRITELN;
500 ' ' : WRITELN;
501 '!' : WRITELN;
502 '@' : WRITELN;
503 '#' : WRITELN;
504 '$' : WRITELN;
505 '%' : WRITELN;
506 '^' : WRITELN;
507 '&' : WRITELN;
508 '*' : WRITELN;
509 '(' : WRITELN;
510 ')' : WRITELN;
511 '[' : WRITELN;
512 ']' : WRITELN;
513 '{' : WRITELN;
514 '}' : WRITELN;
515 ':' : WRITELN;
516 ';' : WRITELN;
517 ',' : WRITELN;
518 '.' : WRITELN;
519 '-' : WRITELN;
520 '_' : WRITELN;
521 '=' : WRITELN;
522 '+' : WRITELN;
523 '/' : WRITELN;
524 '*' : WRITELN;
525 ' ' : WRITELN;
526 '!' : WRITELN;
527 '@' : WRITELN;
528 '#' : WRITELN;
529 '$' : WRITELN;
530 '%' : WRITELN;
531 '^' : WRITELN;
532 '&' : WRITELN;
533 '*' : WRITELN;
534 '(' : WRITELN;
535 ')' : WRITELN;
536 '[' : WRITELN;
537 ']' : WRITELN;
538 '{' : WRITELN;
539 '}' : WRITELN;
540 ':' : WRITELN;
541 ';' : WRITELN;
542 ',' : WRITELN;
543 '.' : WRITELN;
544 '-' : WRITELN;
545 '_' : WRITELN;
546 '=' : WRITELN;
547 '+' : WRITELN;
548 '/' : WRITELN;
549 '*' : WRITELN;
550 ' ' : WRITELN;
551 '!' : WRITELN;
552 '@' : WRITELN;
553 '#' : WRITELN;
554 '$' : WRITELN;
555 '%' : WRITELN;
556 '^' : WRITELN;
557 '&' : WRITELN;
558 '*' : WRITELN;
559 '(' : WRITELN;
560 ')' : WRITELN;
561 '[' : WRITELN;
562 ']' : WRITELN;
563 '{' : WRITELN;
564 '}' : WRITELN;
565 ':' : WRITELN;
566 ';' : WRITELN;
567 ',' : WRITELN;
568 '.' : WRITELN;
569 '-' : WRITELN;
570 '_' : WRITELN;
571 '=' : WRITELN;
572 '+' : WRITELN;
573 '/' : WRITELN;
574 '*' : WRITELN;
575 ' ' : WRITELN;
576 '!' : WRITELN;
577 '@' : WRITELN;
578 '#' : WRITELN;
579 '$' : WRITELN;
580 '%' : WRITELN;
581 '^' : WRITELN;
582 '&' : WRITELN;
583 '*' : WRITELN;
584 '(' : WRITELN;
585 ')' : WRITELN;
586 '[' : WRITELN;
587 ']' : WRITELN;
588 '{' : WRITELN;
589 '}' : WRITELN;
590 ':' : WRITELN;
591 ';' : WRITELN;
592 ',' : WRITELN;
593 '.' : WRITELN;
594 '-' : WRITELN;
595 '_' : WRITELN;
596 '=' : WRITELN;
597 '+' : WRITELN;
598 '/' : WRITELN;
599 '*' : WRITELN;
600 ' ' : WRITELN;
601 '!' : WRITELN;
602 '@' : WRITELN;
603 '#' : WRITELN;
604 '$' : WRITELN;
605 '%' : WRITELN;
606 '^' : WRITELN;
607 '&' : WRITELN;
608 '*' : WRITELN;
609 '(' : WRITELN;
610 ')' : WRITELN;
611 '[' : WRITELN;
612 ']' : WRITELN;
613 '{' : WRITELN;
614 '}' : WRITELN;
615 ':' : WRITELN;
616 ';' : WRITELN;
617 ',' : WRITELN;
618 '.' : WRITELN;
619 '-' : WRITELN;
620 '_' : WRITELN;
621 '=' : WRITELN;
622 '+' : WRITELN;
623 '/' : WRITELN;
624 '*' : WRITELN;
625 ' ' : WRITELN;
626 '!' : WRITELN;
627 '@' : WRITELN;
628 '#' : WRITELN;
629 '$' : WRITELN;
630 '%' : WRITELN;
631 '^' : WRITELN;
632 '&' : WRITELN;
633 '*' : WRITELN;
634 '(' : WRITELN;
635 ')' : WRITELN;
636 '[' : WRITELN;
637 ']' : WRITELN;
638 '{' : WRITELN;
639 '}' : WRITELN;
640 ':' : WRITELN;
641 ';' : WRITELN;
642 ',' : WRITELN;
643 '.' : WRITELN;
644 '-' : WRITELN;
645 '_' : WRITELN;
646 '=' : WRITELN;
647 '+' : WRITELN;
648 '/' : WRITELN;
649 '*' : WRITELN;
650 ' ' : WRITELN;
651 '!' : WRITELN;
652 '@' : WRITELN;
653 '#' : WRITELN;
654 '$' : WRITELN;
655 '%' : WRITELN;
656 '^' : WRITELN;
657 '&' : WRITELN;
658 '*' : WRITELN;
659 '(' : WRITELN;
660 ')' : WRITELN;
661 '[' : WRITELN;
662 ']' : WRITELN;
663 '{' : WRITELN;
664 '}' : WRITELN;
665 ':' : WRITELN;
666 ';' : WRITELN;
667 ',' : WRITELN;
668 '.' : WRITELN;
669 '-' : WRITELN;
670 '_' : WRITELN;
671 '=' : WRITELN;
672 '+' : WRITELN;
673 '/' : WRITELN;
674 '*' : WRITELN;
675 ' ' : WRITELN;
676 '!' : WRITELN;
677 '@' : WRITELN;
678 '#' : WRITELN;
679 '$' : WRITELN;
680 '%' : WRITELN;
681 '^' : WRITELN;
682 '&' : WRITELN;
683 '*' : WRITELN;
684 '(' : WRITELN;
685 ')' : WRITELN;
686 '[' : WRITELN;
687 ']' : WRITELN;
688 '{' : WRITELN;
689 '}' : WRITELN;
690 ':' : WRITELN;
691 ';' : WRITELN;
692 ',' : WRITELN;
693 '.' : WRITELN;
694 '-' : WRITELN;
695 '_' : WRITELN;
696 '=' : WRITELN;
697 '+' : WRITELN;
698 '/' : WRITELN;
699 '*' : WRITELN;
700 ' ' : WRITELN;
701 '!' : WRITELN;
702 '@' : WRITELN;
703 '#' : WRITELN;
704 '$' : WRITELN;
705 '%' : WRITELN;
706 '^' : WRITELN;
707 '&' : WRITELN;
708 '*' : WRITELN;
709 '(' : WRITELN;
710 ')' : WRITELN;
711 '[' : WRITELN;
712 ']' : WRITELN;
713 '{' : WRITELN;
714 '}' : WRITELN;
715 ':' : WRITELN;
716 ';' : WRITELN;
717 ',' : WRITELN;
718 '.' : WRITELN;
719 '-' : WRITELN;
720 '_' : WRITELN;
721 '=' : WRITELN;
722 '+' : WRITELN;
723 '/' : WRITELN;
724 '*' : WRITELN;
725 ' ' : WRITELN;
726 '!' : WRITELN;
727 '@' : WRITELN;
728 '#' : WRITELN;
729 '$' : WRITELN;
730 '%' : WRITELN;
731 '^' : WRITELN;
732 '&' : WRITELN;
733 '*' : WRITELN;
734 '(' : WRITELN;
735 ')' : WRITELN;
736 '[' : WRITELN;
737 ']' : WRITELN;
738 '{' : WRITELN;
739 '}' : WRITELN;
740 ':' : WRITELN;
741 ';' : WRITELN;
742 ',' : WRITELN;
743 '.' : WRITELN;
744 '-' : WRITELN;
745 '_' : WRITELN;
746 '=' : WRITELN;
747 '+' : WRITELN;
748 '/' : WRITELN;
749 '*' : WRITELN;
750 ' ' : WRITELN;
751 '!' : WRITELN;
752 '@' : WRITELN;
753 '#' : WRITELN;
754 '$' : WRITELN;
755 '%' : WRITELN;
756 '^' : WRITELN;
757 '&' : WRITELN;
758 '*' : WRITELN;
759 '(' : WRITELN;
760 ')' : WRITELN;
761 '[' : WRITELN;
762 ']' : WRITELN;
763 '{' : WRITELN;
764 '}' : WRITELN;
765 ':' : WRITELN;
766 ';' : WRITELN;
767 ',' : WRITELN;
768 '.' : WRITELN;
769 '-' : WRITELN;
770 '_' : WRITELN;
771 '=' : WRITELN;
772 '+' : WRITELN;
773 '/' : WRITELN;
774 '*' : WRITELN;
775 ' ' : WRITELN;
776 '!' : WRITELN;
777 '@' : WRITELN;
778 '#' : WRITELN;
779 '$' : WRITELN;
780 '%' : WRITELN;
781 '^' : WRITELN;
782 '&' : WRITELN;
783 '*' : WRITELN;
784 '(' : WRITELN;
785 ')' : WRITELN;
786 '[' : WRITELN;
787 ']' : WRITELN;
788 '{' : WRITELN;
789 '}' : WRITELN;
790 ':' : WRITELN;
791 ';' : WRITELN;
792 ',' : WRITELN;
793 '.' : WRITELN;
794 '-' : WRITELN;
795 '_' : WRITELN;
796 '=' : WRITELN;
797 '+' : WRITELN;
798 '/' : WRITELN;
799 '*' : WRITELN;
800 ' ' : WRITELN;
801 '!' : WRITELN;
802 '@' : WRITELN;
803 '#' : WRITELN;
804 '$' : WRITELN;
805 '%' : WRITELN;
806 '^' : WRITELN;
807 '&' : WRITELN;
808 '*' : WRITELN;
809 '(' : WRITELN;
810 ')' : WRITELN;
811 '[' : WRITELN;
812 ']' : WRITELN;
813 '{' : WRITELN;
814 '}' : WRITELN;
815 ':' : WRITELN;
816 ';' : WRITELN;
817 ',' : WRITELN;
818 '.' : WRITELN;
819 '-' : WRITELN;
820 '_' : WRITELN;
821 '=' : WRITELN;
822 '+' : WRITELN;
823 '/' : WRITELN;
824 '*' : WRITELN;
825 ' ' : WRITELN;
826 '!' : WRITELN;
827 '@' : WRITELN;
828 '#' : WRITELN;
829 '$' : WRITELN;
830 '%' : WRITELN;
831 '^' : WRITELN;
832 '&' : WRITELN;
833 '*' : WRITELN;
834 '(' : WRITELN;
835 ')' : WRITELN;
836 '[' : WRITELN;
837 ']' : WRITELN;
838 '{' : WRITELN;
839 '}' : WRITELN;
840 ':' : WRITELN;
841 ';' : WRITELN;
842 ',' : WRITELN;
843 '.' : WRITELN;
844 '-' : WRITELN;
845 '_' : WRITELN;
846 '=' : WRITELN;
847 '+' : WRITELN;
848 '/' : WRITELN;
849 '*' : WRITELN;
850 ' ' : WRITELN;
851 '!' : WRITELN;
852 '@' : WRITELN;
853 '#' : WRITELN;
854 '$' : WRITELN;
855 '%' : WRITELN;
856 '^' : WRITELN;
857 '&' : WRITELN;
858 '*' : WRITELN;
859 '(' : WRITELN;
860 ')' : WRITELN;
861 '[' : WRITELN;
862 ']' : WRITELN;
863 '{' : WRITELN;
864 '}' : WRITELN;
865 ':' : WRITELN;
866 ';' : WRITELN;
867 ',' : WRITELN;
868 '.' : WRITELN;
869 '-' : WRITELN;
870 '_' : WRITELN;
871 '=' : WRITELN;
872 '+' : WRITELN;
873 '/' : WRITELN;
874 '*' : WRITELN;
875 ' ' : WRITELN;
876 '!' : WRITELN;
877 '@' : WRITELN;
878 '#' : WRITELN;
879 '$' : WRITELN;
880 '%' : WRITELN;
881 '^' : WRITELN;
882 '&' : WRITELN;
883 '*' : WRITELN;
884 '(' : WRITELN;
885 ')' : WRITELN;
886 '[' : WRITELN;
887 ']' : WRITELN;
888 '{' : WRITELN;
889 '}' : WRITELN;
890 ':' : WRITELN;
891 ';' : WRITELN;
892 ',' : WRITELN;
893 '.' : WRITELN;
894 '-' : WRITELN;
895 '_' : WRITELN;
896 '=' : WRITELN;
897 '+' : WRITELN;
898 '/' : WRITELN;
899 '*' : WRITELN;
900 ' ' : WRITELN;
901 '!' : WRITELN;
902 '@' : WRITELN;
903 '#' : WRITELN;
904 '$' : WRITELN;
905 '%' : WRITELN;
906 '^' : WRITELN;
907 '&' : WRITELN;
908 '*' : WRITELN;
909 '(' : WRITELN;
910 ')' : WRITELN;
911 '[' : WRITELN;
912 ']' : WRITELN;
913 '{' : WRITELN;
914 '}' : WRITELN;
915 ':' : WRITELN;
916 ';' : WRITELN;
917 ',' : WRITELN;
918 '.' : WRITELN;
919 '-' : WRITELN;
920 '_' : WRITELN;
921 '=' : WRITELN;
922 '+' : WRITELN;
923 '/' : WRITELN;
924 '*' : WRITELN;
925 ' ' : WRITELN;
926 '!' : WRITELN;
927 '@' : WRITELN;
928 '#' : WRITELN;
929 '$' : WRITELN;
930 '%' : WRITELN;
931 '^' : WRITELN;
932 '&' : WRITELN;
933 '*' : WRITELN;
934 '(' : WRITELN;
935 ')' : WRITELN;
936 '[' : WRITELN;
937 ']' : WRITELN;
938 '{' : WRITELN;
939 '}' : WRITELN;
940 ':' : WRITELN;
941 ';' : WRITELN;
942 ',' : WRITELN;
943 '.' : WRITELN;
944 '-' : WRITELN;
945 '_' : WRITELN;
946 '=' : WRITELN;
947 '+' : WRITELN;
948 '/' : WRITELN;
949 '*' : WRITELN;
950 ' ' : WRITELN;
951 '!' : WRITELN;
952 '@' : WRITELN;
953 '#' : WRITELN;
954 '$' : WRITELN;
955 '%' : WRITELN;
956 '^' : WRITELN;
957 '&' : WRITELN;
958 '*' : WRITELN;
959 '(' : WRITELN;
960 ')' : WRITELN;
961 '[' : WRITELN;
962 ']' : WRITELN;
963 '{' : WRITELN;
964 '}' : WRITELN;
965 ':' : WRITELN;
966 ';' : WRITELN;
967 ',' : WRITELN;
968 '.' : WRITELN;
969 '-' : WRITELN;
970 '_' : WRITELN;
971 '=' : WRITELN;
972 '+' : WRITELN;
973 '/' : WRITELN;
974 '*' : WRITELN;
975 ' ' : WRITELN;
976 '!' : WRITELN;
977 '@' : WRITELN;
978 '#' : WRITELN;
979 '$' : WRITELN;
980 '%' : WRITELN;
981 '^' : WRITELN;
982 '&' : WRITELN;
983 '*' : WRITELN;
984 '(' : WRITEL
```


Et voici un exemple d'exécution:

```

heapptr: $09E8:$0000
p(ointeur, q(uite ? p
  valeur pointée <OE> ? ALICE

$09B1:$0260          $41 $41 $41 $41 $41 : AAAAA
$09B1:$0268 $41 $41 $41 $41 $41 $41 $41 : AAAAAAA
$09B1:$0270 $41 $41 $41 $41 $41 $41 $41 : AAA
$09B1:$0270          $00 $00 $E8 $09 : ....
$09B1:$0270          $42 : B
$09B1:$0278 $42 $42 $42 $42 $42 $42 $42 :BBBBBBBB
$09B1:$0280 $42 $42 $42 $42 $42 $42 $42 :BBBBBBBB

$09E8:$0000 $05 $41 $4C $49 $43 $45 :.ALICE

heapptr: $09E8:$0008
p(ointeur, q(uite ? p
  valeur pointée <OE> ? JOE

$09B1:$0260          $41 $41 $41 $41 $41 : AAAAA
$09B1:$0268 $41 $41 $41 $41 $41 $41 $41 : AAAAAAA
$09B1:$0270 $41 $41 $41 $41 $41 $41 $41 : AAA
$09B1:$0270          $08 $00 $E8 $09 : ....
$09B1:$0270          $42 : B
$09B1:$0278 $42 $42 $42 $42 $42 $42 $42 :BBBBBBBB
$09B1:$0280 $42 $42 $42 $42 $42 $42 $42 :BBBBBBBB

$09E8:$0008 $03 $4A $4F $45 $00 $00 :.JOE..

heapptr: $09E9:$0000
09B1
p(ointeur, q(uite ? q

```

Fig. 4.54 Un pointeur en mémoire

Notez:

- l'espace inutilisé entre la fin des variables globales et le début du tas (pour aligner le tas sur une frontière de paragraphe)
- l'espace inutilisé après chacune de nos variables pointées (pour aligner le début d'une variable pointée sur une adresse multiple de 8)

4.8.7 LES POINTEURS D'ENREGISTREMENTS VARIANTS

Si la variable pointée est un enregistrement variant, WIRTH avait prévu de permettre au programmeur de préciser au moment de l'exécution de NEW la valeur du sélecteur. Le programme devait alors réserver sur le tas exactement la place dont a besoin la variante correspondante.

Si nous déclarons:

```

TYPE t_cellule= RECORD
  CASE INTEGER OF
    1 : (numero: INTEGER);
    2 : (nom: STRING(7));
  END;
VAR racine: ^t_cellule;

```

alors:

- l'exécution de:

```
NEW(racine, 1);
```

devait allouer 2 octets sur le tas

- l'exécution de:

```
NEW(racine, 2);
```

devait allouer 8 octets

- et l'exécution de:

```
NEW(racine)
```

devait allouer, comme dans le cas général des variants, la place maximale nécessaire pour la plus grande des variantes, ici 8 octets.

Cette possibilité n'a pas été implémentée par TURBO PASCAL.

4.8.8 RESUME

Le type pointeur:

- peut prendre les valeurs correspondant aux adresses possibles des variables dynamiques (ici 0..\$FFFF)
- utilise 4 octets, les 2 premiers contenant le décalage de la variable pointée, les 2 suivants son segment
- NEW réserve la place nécessaire à la variable pointée, plus les octets nécessaires pour que la prochaine variable pointée débute à une adresse multiple de 8
- une variable pointée ne peut pas dépasser 64K
- le tas débute après les variables globales et peut atteindre le sommet de la pile. Sa taille peut dépasser 64K.
- la valeur NIL, compatible avec tous les pointeurs, a la valeur numérique 0

4.9 FORMAT DES FICHIERS SUR DISQUETTE

4.9.1 INTRODUCTION

Nous appelons ici fichier sur disquette toute structure conservée de façon permanente sur la disquette.

Comme nous allons dans les chapitres qui viennent utiliser des fichiers, nous allons détailler ici leur format sur les disquettes.

Nous allons ici examiner le contenu de 3 types de fichiers:

- les fichiers qui contiennent le code généré par le compilateur TURBO (les fichiers .COM)
- les fichiers de données que nous gérons par nos programmes PASCAL (fichiers de données)
- les fichiers qui contiennent le texte de nos programmes (les fichiers .PAS)

Nous ne sommes concernés ici que par la structure des octets de ces fichiers. Nous n'examinerons pas ici:

- comment organiser ou gérer des fichiers de données en PASCAL (choix d'index, répartition des données entre plusieurs fichiers...)
- comment TURBO PASCAL gère les fichiers en mémoire (quels tampons sont utilisés, comment EOF est détecté)
- comment MS-DOS gère les fichiers (dans quels secteurs sont placées les données, quelle est la structure de la table des matières, de la disquette, quelle est la correspondance entre les secteurs physiques et les secteurs logiques)

4.9.2 LES FICHIERS DE CODE

Lorsque nous compilons sur disque, TURBO crée un fichier .COM dans lequel est placé:

- de l'octet 0 à l'octet \$2D4C- \$0100 la librairie de TURBO PASCAL
- de l'octet \$2D4D- \$0100 à \$xxxx- \$0100 les octets correspondant à vos instructions

Lorsque vous lancerez l'exécution de ce programme, MS-DOS:

- créera et initialisera en mémoire une zone de \$0000 à \$00FF pour gérer ce programme (cette zone est appelée Program Segment Prefix)
- chargera de \$0100 à \$xxxx le programme
- donnera la main à l'instruction située au décalage \$0100

Le premier octet du fichier .COM sera donc placé au décalage \$0100 du segment de code en mémoire. A part cela, le fichier .COM est une copie parfaite du programme en mémoire. Bien sûr, une fois que MS-DOS donne la main à notre programme, celui-ci effectuera un certain nombre d'initialisations, dont certaines modifieront le code-même. Nous examinerons le fonctionnement d'un programme TURBO en mémoire dans le chapitre suivant.

RESUME

Un fichier .COM résultant d'une compilation par TURBO PASCAL contient de l'octet \$0000 à \$xxxx le code qui sera chargé par MS-DOS en mémoire à partir du décalage \$0100 du segment de code alloué à ce programme.

4.9.3 LES FICHIERS DE DONNEES

Considérons à présent un fichier de données usuel, déclaré par **FILE OF**. Les octets sur la disquette sont exactement ceux contenus dans chaque fiche, ces fiches étant placées les unes derrière les autres sur la disquette.

Voici un programme qui place quelques fiches sur la disquette:

```

0 (* 001 fichier *)
1 (* 23 avr 87 *)
2
3 (*$r+*)
4 PROGRAM examen_fichier_donnees;
5   TYPE t_fiche= RECORD
6     nom: STRING[3];
7     age: BYTE;
8   END;
9   VAR choix: CHAR;
10    fichier: FILE OF t_fiche;
11    fiche_courante: t_fiche;
12 BEGIN
13   ASSIGN(fichier, 'noms.dta');
14   REWRITE(fichier);
15
16   REPEAT
17     WITH fiche_courante DO
18     BEGIN
19       WRITE('nom <JOE> ? '); READLN(nom);
20       WRITE('age <33> ? '); READLN(age);
21     END;
22     WRITE(fichier, fiche_courante);
23
24     WRITE('continuer, q(uitter) ? ');
25     READ(KBD, choix); WRITELN(choix);
26   UNTIL choix= 'q';
27
28   CLOSE(fichier);
29 END.
```

Fig. 4.55 Le programme *examen_fichier_donnees*.

Voici un exemple d'exécution:

```
nom <JOE> ? JOE
age <33> ? 33
continuer, q(uitter ? c
nom <JOE> ? SAM
age <33> ? 34
continuer, q(uitter ? c
nom <JOE> ? GUS
age <33> ? 35
continuer, q(uitter ? q
```

Fig. 4.56 La création du fichier

La table des matières nous indique que ce fichier utilise 15 octets de la disquette. Ceci correspond bien à la place nécessaire aux trois fiches. Nous utilisons alors le programme d'examen du contenu des secteurs d'un fichier, présenté au début du chapitre. Voici le résultat de cet examen:

```
???
n(om, a(ffiche, q(uitter ? n
  nom fichier <affich.pas> ? noms.dta

noms.dta
n(om, a(ffiche, q(uitter ? a
  secteur    <3> ? 0

$0000- $0000:  $03 $4A $4F $45 $21      .JOE!
               :      $03 $53 $41      .SA
$0000- $0008:  $4D $22                    M"
               :      $03 $47 $55 $53 $23      .GUS#
               :                      $00
$0000- $0010:  $00 $00 $00 $00 $00 $00 $00 .....
noms.dta
n(om, a(ffiche, q(uitter ? q
```

Fig. 4.57 Un fichier de données

RESUME

Un fichier de données déclaré par **FILE OF**:

- utilise sur la disquette pour chaque fiche le nombre d'octets nécessaires au type déclaré après **OF**
- place les fiches les unes derrière les autres (la première étant la numéro 0)

4.9.4 LES FICHIERS DE BLOCS

Nous pouvons aussi utiliser des fichiers de données constitués de blocs d'octets, sans avoir à préciser un type de fiche (en fait ces fiches sont des tableaux d'octets). Ces fichiers:

- se déclarent par **FILE** (sans préciser **OF**). Par exemple:

```
VAR fichier_de_blocs: FILE;
```

- doivent être créés par **ASSIGN** et **REWRITE**, réouverts par **ASSIGN** et **RESET**, et fermés par **CLOSE**
- effectuent des transferts disquette / mémoire par blocs d'octets. Par défaut la taille de ces blocs est de 128 octets, mais peut être spécifiée dans **RESET**:

```
RESET(fichier_de_blocs, 512);
```

- les échanges avec la mémoire s'effectuent par le biais d'un tampon, dont la taille par défaut doit être 128, ou doit correspondre au paramètre optionnel de **REWRITE** ou **RESET**

```
VAR tampon: ARRAY[0..511] OF BYTE;
```

- le transfert est effectué à l'aide des instructions **BLOCKREAD** et **BLOCKWRITE**:

```
BLOCKREAD(fichier_de_blocs, tampon, nombre_blocs, resultat);
```

Au point de vue format cependant, ce type de fichier se comporte exactement comme les fichiers **FILE OF**. Les octets produits sont identiques à ceux qu'aurait produit un fichier déclaré par:

```
TYPE t_bloc = ARRAY[0..511] OF BYTE;
VAR fichier_de_blocs: FILE OF t_bloc;
```

Seule est modifiée la vitesse des transferts mémoire / disque.

4.9.5 LES FICHIERS DE TEXTE

L'EDITEUR TURBO place sur la disquette le texte de nos programmes dans des fichiers qui ont par défaut le suffixe **.PAS**.

Le format de ces fichiers est le suivant:

- zéro ou plusieurs lignes séparées par **RETURN** (code ASCII \$0D) et **INTERLIGNE** (code ASCII \$0A)
- le caractère **CTRL-Z** (code ASCII \$1A)

Voici un exemple d'un tel texte;

```
0 (* 001 PROGRAM *)
1 (* 23 avr 87 *)
2
3 PROGRAM examen_programme;
4 BEGIN
5   WRITELN('OK');
6 END.
```

Fig. 4.58 Le programme *examen_programme*.

et voici le contenu du fichier sur disque:

```
???
n(om, a(ffiche, q(uite ? n
  nom fichier <affich.pas> ? program.pas
program.pas

n(om, a(ffiche, q(uite ? a
  secteur <3> ? 0

$0000- $0000:  $28 $2A $20 $30 $30 $31 $20 $70 (* 001 p
$0000- $0008:  $72 $6F $67 $72 $61 $6D $20 $2A rogram *
$0000- $0010:  $29 $0D $0A $28 $2A $20 $32 $33 )..(* 23
$0000- $0018:  $20 $61 $76 $72 $20 $38 $37 $20 avr 87
$0000- $0020:  $2A $29 $0D $0A $0D $0A $70 $72 )....pr
$0000- $0028:  $6F $67 $72 $61 $6D $20 $65 $78 ogram ex
$0000- $0030:  $61 $6D $65 $6E $5F $70 $72 $6F amen_pro
$0000- $0038:  $67 $72 $61 $6D $6D $65 $3B $0D gramme;..
$0000- $0040:  $0A $20 $20 $62 $65 $67 $69 $6E . begin
$0000- $0048:  $0D $0A $20 $20 $20 $20 $77 $72 .. wr
$0000- $0050:  $69 $74 $65 $6C $6E $28 $27 $4F iteln('O
$0000- $0058:  $4B $27 $29 $3B $0D $0A $20 $20 K');...
$0000- $0060:  $65 $6E $64 $2E $1A $00 $00 $00 end.....
$0000- $0068:  $00 $00 $00 $00 $00 $00 $00 .....

...

program.pas
n(om, a(ffiche, q(uite ? q
```

Fig. 4.59 Un fichier créé par l'EDITEUR

Notez que:

- le plus petit texte possible comporte uniquement le CTRL-Z
- la dernière ligne ne comporte pas nécessairement de RETURN INTERLIGNE
- les espaces figurant en début de ligne sont tous dans le fichier (aucun mécanisme de compression d'espace n'est utilisé)

Précisons toutefois que le COMPILATEUR TURBO accepte certains caractères de compression (comme le code ASCII 9 correspondant à la touche TAB), même si l'EDITEUR ne sait pas les exploiter pour l'affichage (le i en vidéo faible est affiché).

Nous pouvons créer des fichiers ayant ce format à l'aide de variable de type TEXT. Voici un programme qui écrit quelques lignes dans un tel fichier:

```
0 (* 001 texte *)
1 (* 23 avr 87 *)
2
3 PROGRAM fichier_texte;
4 VAR texte: TEXT;
5     caractere: CHAR;
6 BEGIN
7   ASSIGN(texte, 'phrases.txt');
8   REWRITE(texte);
9
10  WRITE(texte, caractere);
11  WRITELN(texte);
12  WRITE(texte, 2, 5* 3, 17: 6, 1/ 3, 'fin');
13
14  CLOSE(texte);
15 END.
```

Fig. 4.60 Le programme *fichier_texte*.

et voici le contenu du fichier créé par ce programme:

```
???
n(om, a(ffiche, q(uite ? n
  nom fichier <affich.pas> ? phrases.txt

phrases.txt
n(om, a(ffiche, q(uite ? a
  secteur <3> ? 0

$0000- $0000:  $43 $0D $0A $32 $31 $35 $20 $20 C..215
$0000- $0008:  $20 $20 $31 $37 $20 $20 $33 $2E 17 3.
$0000- $0010:  $33 $33 $33 $33 $33 $33 $33 33333333
$0000- $0018:  $33 $33 $45 $2D $30 $31 $66 $69 33E-01fi
$0000- $0020:  $6E $00 $00 $00 $00 $00 $00 $00 n.....

...

phrases.txt
n(om, a(ffiche, q(uite ? q
```

Fig. 4.61 Le fichier texte généré

Notez que:

- dans **WRITE** nous pouvons placer non seulement des variables de type caractère, mais tout ce qui est permis de placer normalement dans un **WRITE**:
 - des valeurs numériques
 - des expressions numériques
 - des instructions de formatage
 - des chaînes littérales
- le fichier ne comporte pas de CTRL-Z. TURBO PASCAL utilisera donc la taille du fichier pour savoir où il se termine. Rien n'interdit cependant de placer un CTRL-Z par programme avant de fermer le fichier. Si par contre un CTRL-Z traîne au milieu du fichier, READ considérera que la fin a été atteinte lorsque ce CTRL-Z sera atteint.

La possibilité de pouvoir lire ou écrire nos propres fichiers de texte est mise à profit:

- pour réaliser des fichiers séquentiels ayant des fiches de taille variable
- pour rediriger les entrées du clavier (jeux de test)
- pour générer des textes de programme, "comme si" un programmeur les avait tapés (générateurs de programme)

RESUME

Un fichier créé par l'EDITEUR TURBO PASCAL comporte

- zéro ou plusieurs lignes séparées par RETURN (code ASCII \$0D) et INTERLIGNE (code ASCII \$0A)
- le caractère CTRL-Z (code ASCII \$1A)

Des fichiers ayant le même format peuvent être lus ou écrits à l'aide de fichiers de type **TEXT**. **WRITE** offre alors toutes les possibilités habituelles (conversions numériques / caractère, formatage). Ces fichiers ne comportent cependant pas de CTRL-Z à la fin.

4.A CONCLUSION

Dans ce chapitre nous avons examiné l'implantation des types PASCAL:

- types simples
- types structurés
- fichiers sur disquette

4.B BIBLIOGRAPHIE

La saisie contrôlée de réels a été présentée dans:

PASCALISSIME - Numéro 23 - SAISIE DE REELS - John COLIBRI

Les routines d'affichage et de conversion de réels figurent dans:

COLIBRI John - TOPIQUES PASCAL

A l'issue de ce chapitre, vous serez en mesure:

- d'interpréter un listing du code généré par TURBO PASCAL

Le chapitre 6 présentera les options de compilation, qui modifient le comportement du compilateur.

Le chapitre 7 présentera le contenu de la bibliothèque TURBO PASCAL.

CHAPITRE 5

- le traitement des variables simples
- le traitement des variables structurées
- les fichiers et les instructions de lecture et d'écriture
- les instructions de contrôle
- les procédures et les appels de procédures
- les constantes typées
- les variables absolues

A l'issue de ce chapitre, vous serez en mesure:

- d'interpréter un listing du code généré par TURBO PASCAL

Le chapitre 6 présentera les options de compilation, qui modifient le comportement du compilateur.

Le chapitre 7 présentera le contenu de la librairie TURBO PASCAL.

5.1 L'ANALYSE D'UN PROGRAMME COMPILE**5.1.1 L'ANALYSE DU CODE**

Nous allons nous intéresser dans ce chapitre au fonctionnement d'un programme compilé avec TURBO PASCAL.

Pour cela, nous allons compiler plusieurs programmes, et analyser le contenu et le fonctionnement du code généré par le compilateur TURBO.

La meilleure compréhension du code généré permet:

- de connaître la cause de certaines caractéristiques de TURBO PASCAL (lenteur des calculs réels par exemple)
- de traiter correctement certaines nouveautés TURBO (comme les constantes typées)
- d'interfacer correctement un programme compilé avec TURBO PASCAL avec d'autres programmes ou de mieux utiliser des instructions INLINE

Notre analyse provient de l'examen du code après désassemblage. Il n'est cependant pas nécessaire que vous compreniez le détail de ces listings de désassemblage. Ces listings sont là uniquement pour illustrer l'analyse.

Le désassemblage a été effectué avec le désassembleur 8088 de l'INSTITUT PASCAL. C'est un désassembleur "intelligent":

- il ne désassemble que le code correspondant à un programme (et pas la librairie TURBO)
- il tient compte de la structure du code (paramètres du début, taille du code)
- il reconnaît automatiquement le type de désassemblage à effectuer (code 8088, zone ASCII, zone hexadécimale)
- il contient les adresses symboliques de la librairie TURBO. La liste complète de ces adresses symbolique figure au chapitre 7. Ces labels sont tous préfixés par TU_. Par exemple la routine de la librairie chargée de réaliser un WRITELN est désignée par TU_WRITELN.

5.1.2 L'ANALYSE D'UN PROGRAMME SIMPLE

Examinons pour commencer le code du programme PASCAL le plus simple possible:

```
0 PROGRAM vide;
1 BEGIN
2 END.
```

Fig. 5.1 Le programme vide.

Voici le listing de désassemblage correspondant:

```
2D7C 2D4D E849DD E857DD call TU_PARAGRAPHS 0AD6
2D7F 2D50 0600 0600 hex 0600 ; options
2D7F 2D52 7B06 7122 hex 7B06 ; cs
2D7F 2D54 3F10 312C hex 3F10 ; ds
2D7F 2D56 D802 DB02 hex D802 ; code
2D7F 2D58 2700 2600 hex 2700 ; globales
2D7F 2D5A 0004 0004 hex 0004 ; min piles
2D7F 2D5C 00A0 00A0 hex 00A0 ; max piles
2D7F 2D5E 1000 1000 hex 1000 ; fichiers
2D7F 2D60 0000 0000 hex 0000 ; input
2D7F 2D62 0000 0000 hex 0000 ; output

2D93 2D64 8BEC mov bp, sp
2D93 2D66 E814DF call TU_TAILLE_CODE 38CC
2D93 2D69 0F00 hex 0F00 ; taille
2D93 2D6B 6D2D adr SAUT_PRINCIPAL 2D0C

2D93 2D6D E90000 SAUT_PRINCIPAL jmp PRINCIPAL
2D93 2D70 E90000 PRINCIPAL jmp FIN
2D93 2D73 33C0 FIN xor ax, ax
2D93 2D75 E8C8DE DEE2 call TU_HALT 0C89
2D93 2D78 0000 hex 0000
```

Fig. 5.2 Le désassemblage du programme vide

Nous notons:

- en 2D4D un appel à TU_PARAGRAPHS qui est la routine de la librairie chargée d'initialiser l'exécution en utilisant les données situées de 2D50 à 2D53
- en 2D66 un appel à TU_TAILLE_CODE qui effectue une autre série d'initialisations à l'aide des valeurs situées de 2D69 à 2D6C
- en 2D6D se trouve un saut au début du programme principal. Si le programme comportait des procédures, leur code se trouverait entre ce saut et le code du programme principal
- en 2D70 se trouve le saut vers les instructions de fin de programme
- en 2D73 l'appel de fin de programme

Schématiquement nous avons par conséquent:

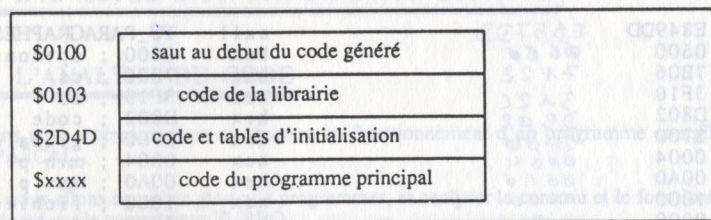


Fig. 5.3 Structure générale d'un programme TURBO PASCAL

5.1.3 LA STRATEGIE DE COMPILATION

La présence des deux sauts à l'adresse qui suit (JMP PRINCIPAL et JMP FIN) s'explique par la stratégie de compilation.

En effet WIRTH a conçu le langage PASCAL de façon à permettre une compilation en une seule passe. A chaque instant, le compilateur a analysé les éléments avant de devoir les utiliser ("déclarer avant d'utiliser"). Si le compilateur rencontre un nom sans avoir analysé sa définition, une erreur de compilation est provoquée.

TURBO PASCAL utilise cette propriété de PASCAL. A aucun moment le compilateur ne revient en arrière pour réanalyser le texte, ou pour modifier des instructions 8088 déjà générées.

Par conséquent le compilateur se place toujours dans le cas le plus général, quitte à générer des temps en temps du code inutile (mais correct).

Ainsi, l'instruction JMP PRINCIPAL permet de sauter par dessus le code d'éventuelles procédures. Voici par exemple un programme avec une procédure:

```

0 (* 001 proc *)
1 (* 16 aou 87 *)
2
3 PROGRAM avec_procedure;
4
5 PROCEDURE deux;
6 BEGIN
7   END; (* deux *)
8
9 BEGIN (* principal *)
10  END. (* principal *)

```

Fig. 5.4 Le programme avec_procedure.

dont le code est le suivant:

```

2D4D E849DD call TU_PARAGRAPHE
2D50 0600 hex 0600 ; options
2D52 780B hex 780B ; cs
2D54 3C15 hex 3C15 ; ds
2D56 D902 hex D902 ; code
2D58 2700 hex 2700 ; globales
2D5A 0004 hex 0004 ; min piles
2D5C 00A0 hex 00A0 ; max piles
2D5E 1000 hex 1000 ; fichiers
2D60 0000 hex 0000 ; input
2D62 0000 hex 0000 ; output

2D64 8BEC mov bp, sp
2D66 E814DF call TU_TAILLE_CODE
2D69 1D00 hex 1D00 ; taille
2D6B 6D2D adr SAUT_PRINCIPAL

2D6D E90E00 SAUT_PRINCIPAL jmp PRINCIPAL

2D70 55 DEUX push bp
2D71 8BEC mov bp, sp
2D73 55 push bp
2D74 E90000 jmp DEBUT_PROCEDURE

2D77 E90000 DEBUT_PROCEDURE jmp FIN_PROCEDURE

2D7A 8BE5 FIN_PROCEDURE mov sp, bp
2D7C 5D pop bp
2D7D C3 ret

2D7E E90000 PRINCIPAL jmp FIN

2D81 33C0 FIN xor ax, ax
2D83 E8BADE call TU_HALT
2D86 0000 hex 0000

```

Fig. 5.5 Un programme avec une procédure

Par conséquent:

- lorsque le compilateur rencontre le ; après le nom du programme (ligne 3), il génère le code d'initialisation et le saut au code du programme principal. Comme la valeur du saut n'est pas connue, le compilateur place \$0000 pour le moment après le JMP
- la procédure est compilée (lignes 4 à 8, générant le code de \$2D64 à \$2D7D)
- lorsque le compilateur rencontre le BEGIN du programme principal (ligne 9), il calcule la valeur du saut au début du programme principal et place cette valeur en \$2D6E
- les lignes du programme principal (ligne 9 et 10) sont alors compilées.

Sans trop anticiper sur ce qui sera présenté plus en détail ci-dessous, nous pouvons alors préciser le contenu du code généré par le compilateur:

- en \$2D4D un appel à TU_PARAGRAPHERS qui utilise les paramètres permettant la mise en place du programme en mémoire avant son exécution. Les paramètres sont les suivants:
 - *options*: code les options globales de compilation (I, D, C et U)
 - *cs*: contient le segment de code de TURBO.COM (utilisé pour retourner dans TURBO après avoir exécuté un programme compilé en mémoire)
 - *ds*: contient le segment de donnée de TURBO.COM (utilisé pour restaurer le segment de donnée de TURBO après l'exécution d'un programme compilé en mémoire)
 - *taille code, taille globales, min stack/heap, max stack/heap* contiennent le nombre de paragraphes utilisés par le code, les données globales, le tas et la pile. Ces valeurs sont calculées par le compilateur après la compilation, ou déterminées par le programmeur en utilisant les options du menu
 - *fichiers*: contient le nombre de fichiers qui peuvent être ouverts à un instant donné (option F)
 - *input* et *output* permettent de rediriger les fichiers d'entrée et de sortie par défaut (option G et P)
- en \$2D66 le compilateur place un appel à TU_TAILLE_CODE. Cette routine de la librairie est chargée de faire la place dans le code pour les overlays. En effet le fichier .COM ne contient que le code des procédures qui ne sont pas des overlays. Ceci permet d'économiser de la place sur disque et du temps au chargement. Par contre au moment de l'exécution, il faut ménager des trous dans le code chargé. Ces trous sont effectués en partant de la fin du code. Les 2 paramètres en \$2D69 et \$2D6B permettent d'effectuer cette manœuvre, en suivant une liste chaînée de trous à ouvrir. Lorsque votre programme ne contient pas d'overlay, cette procédure ne fait rien.
- en \$2D6D se trouve le saut à la première instruction du programme principal.
- entre le saut au programme principal et la première instruction de ce programme se trouvent:
 - les constantes typées si vous en avez défini
 - le code des procédures que vous avez déclarées
- vient ensuite le code du programme principal
- après votre dernière instruction, le compilateur place un saut vers les instructions de fin d'exécution
- entre le saut aux instructions de fin d'exécution et ces instructions, le compilateur place du code correspondant à des GOTO ou des EXIT hors de boucles FOR
- finalement le compilateur génère l'appel à la routine TU_HALT. Cette routine rend la main soit à TURBO (compilation mémoire), soit à MS-DOS compilation .COM).

Le code d'un programme TURBO PASCAL a donc l'allure générale suivante:

\$0100	saut au debut du code généré
\$0103	code de la librairie
\$2D4D \$2D66	code et tables d'initialisation création des espaces d'overlays
\$2D6D	saut au programme principal
\$2D70	constantes typées et procédures
\$xxxx	code du programme principal
\$yyyy	saut au code de fin de programme
\$zzzz	GOTO hors de boucles FOR
\$vvvv	code de fin de programme

Fig. 5.6 Structure détaillée d'un programme TURBO PASCAL

5.2 LES PHASES DE COMPILATION

5.2.1 LES ELEMENTS D'UN PROGRAMME

Lorsque le compilateur analyse le texte d'un programme, il peut:

- soit générer des instructions 8088 correspondant au texte. C'est le cas, par exemple, lorsqu'il rencontre une instruction telle que:

```
total:= 5;
```

- soit mettre à jour ses tables internes (appelées tables d'attributs), pour pouvoir générer du code par la suite. C'est le cas, par exemple, lorsqu'il rencontre une définition de type:

```
TYPE jour= 1..31;
```


Nous allons analyser rapidement ce que fait le compilateur en fonction des éléments d'un programme PASCAL. Nous distinguons:

- les définitions de labels
- les définitions de constantes
- les définitions de type
- les déclarations de variables
- les déclarations de procédures ou fonctions
- les instructions

Voyons donc comment le compilateur traite chacun de ces éléments.

5.2.2 LES DEFINITIONS DE LABELS

Les définitions de labels ont uniquement une vocation syntactique. Ces définitions sont uniquement là pour rester cohérent avec le langage (il faut tout définir ou déclarer).

En effet:

- lorsque le compilateur rencontre le mot **LABEL**, il stocke simplement le nom du label dans une de ses tables. Aucun code n'est généré à ce moment.
- lorsque le label est rencontré, le compilateur vérifie que celui-ci a bien été déclaré à ce niveau. Si ce label n'est pas connu, une erreur de compilation est provoquée.

Bien sûr, le compilateur stocke aussi à ce moment la valeur du compteur programme (CP), et cette valeur est utilisée pour calculer les **GOTO** utilisant ce label.

5.2.3 LES DEFINITIONS DE TYPES

Les définitions de types permettent de désigner par un nom de nouvelles données.

Rappelons qu'en PASCAL un type correspond:

- à un ensemble de valeurs possibles
- à un ensemble d'opérateurs, de procédures et de fonctions

Ainsi le type **INTEGER** correspond aux valeurs entières (sans décimales) comprises entre -32768 et 32767, avec les opérations +, -, *, DIV et MOD, et des procédures et fonctions telles que **ABS**, **STR** ou **ODD**.

Les types **INTEGER**, **BYTE**, **REAL**, **CHAR** et **BOOLEAN** sont prédéfinis: le compilateur connaît déjà les valeurs possibles, les opérateurs les procédures et les fonctions utilisables.

Si nous souhaitons utiliser des données ayant des structures autres que celles des types prédéfinis, nous devons indiquer au compilateur quel est leur type.

Les instructions que nous plaçons après **TYPE** servent donc uniquement à garnir les tables d'attributs du compilateur. Lorsque les variables ayant les types correspondants seront utilisées dans des instructions, le compilateur pourra:

- vérifier que ces instructions sont correctes (pas de multiplications de chaînes par des réels par exemple)
- générer le code permettant d'allouer la place en fonction de la taille nécessaire pour le type
- générer le code pour accéder aux données

Les définitions de type ne génèrent donc aucun code. Il s'agit d'une sorte de dictionnaire.

5.2.4 LES DEFINITIONS DE CONSTANTES SIMPLES

Normalement, les constantes sont utilisées:

- pour améliorer la lisibilité d'un programme
- pour améliorer la portabilité d'un programme

En effet, les valeurs littérales placées directement dans le texte peuvent être ambiguës. Par exemple:

```
i := 7;
```

peut aussi bien correspondre au code ASCII de la sonnette, au nombre de jours de la semaine, à la valeur maximale que nous pouvons coder avec 3 bits... L'utilisation de:

```
CONST k_nombre_jours = 7;  
      k_code_sonnette = 7;  
      k_bit_max = 7;
```

permet de savoir dans le texte du programme quel objet nous utilisons.

Certains programmes utilisent aussi des objets dont la taille ou le nombre peuvent dépendre de la configuration. Par exemple un tampon mémoire, ou un nombre de lecteurs. Il est alors de bonne pratique de définir les valeurs maximales dans des constantes:

```
CONST k_taille_max = $4000;  
      k_nombre_lecteurs = 2;
```

```
VAR tampon: ARRAY[1..k_taille_max] OF BYTE;
```

```
...
```

```
FOR indice := 0 to k_taille_max DO
```

```
...
```

```
  lecteur := 1;
```

```
  WHILE lecteur <= k_nombre_lecteurs DO
```

```
    ...
```


Cette pratique a deux avantages:

- elle permet de mettre le programme au point en utilisant de petites valeurs (il est plus simple d'examiner un tampon de 10 octets qu'un tampon de 16000 octets)
- elle permet d'adapter facilement le programme lorsque la configuration est modifiée: il suffit de modifier la valeur des constantes et recompiler le programme.

Au moment de la compilation:

- lorsque le compilateur compile la partie CONST, il place le nom du symbole et sa valeur dans sa table d'attributs
- lorsque le compilateur rencontre le nom de la constante dans des déclarations ou des instructions, il remplace ce symbole par sa valeur.

Comme pour les types, les définitions de constantes servent donc uniquement à garnir la table d'attributs. La définition ne génère aucun code. Par contre l'utilisation de la constante dans une expression utilise la valeur de la constante pour générer le code qui calcule l'expression.

TURBO permet d'utiliser comme constantes:

- tous les types simples (entiers, caractères, booléens, énumérés) et leurs intervalles
- les réels
- les chaînes de caractères (telles que: 'ABCD')

Pour les constantes simples, les réels et les chaînes, le code généré en utilisant des constantes est identique au code généré en utilisant des valeurs littérales (constantes dans une instruction). Par exemple:

```

0 (* 001 CONST *)
1 (* 10 avr 87 *)
2
3 (*$r+*)
4 PROGRAM desassemble_constantes;
5   CONST k_entier= $77;
6         k_chaine= 'ABCD';
7   BEGIN
8     WRITE(k_entier);
9     WRITE(k_chaine);
10  END.
```

Fig. 5.7 Le programme *desassemble_constantes*.

génère exactement le même code que:

```

0 (* 001 literal *)
1 (* 10 avr 87 *)
2
3 (*$r+*)
4 PROGRAM desassemble_literal;
5   BEGIN
6     WRITE($77);
7     WRITE('ABCD');
8   END.
```

Fig. 5.8 Le programme *desassemble_literal*.

Nous examineront le traitement des constantes littérales, sans reprendre le traitement des constantes simples qui génère le même code.

5.2.5 LES VARIABLES

Lorsque le compilateur analyse une déclaration d'une partie VAR:

- il place dans ses tables d'attributs les caractéristiques de cette variable (pour effectuer les vérifications de type)
- il calcule à quel endroit se trouvera cette variable au moment de l'exécution (gestion de l'allocation des variables)

Une variable n'est en effet qu'une zone mémoire d'une certaine taille placée à une certaine adresse. En fonction du type de la variable, le compilateur prévoit de placer ces variables à des adresses différentes.

Nous avons vu que dans notre version la première variable globale sera placée à DS:\$0262.

Si nous déclarons alors:

```

VAR total: INTEGER;
    reel: REAL;
    montant: INTEGER;
```

alors le compilateur sait que:

- *total* utilisera les 2 octets DS:\$0262 et DS:\$0263
- *reel* occupera les 6 octets de DS:\$0264 à DS:\$0269
- *montant* occupera les 2 octets DS:\$026A et DS:\$026B

Ces adresses seront alors utilisées pour accéder à ces données. Par exemple l'instruction:

```
montant:= total+ $77;
```

sera compilé en:

```

"chercher la valeur en DS:$0262"
"additionner $77 à cette valeur"
"stocker le résultat à l'adresse DS:$026A"
```


Voici un programme qui effectue une addition:

```

0 (* 001 alloue *)
1 (* 24 avr 87 *)
2
3 (*$r+*)
4 PROGRAM desassemble_alloue;
5   VAR total: INTEGER;
6       reel: REAL;
7       montant: INTEGER;
8 BEGIN
9   montant:= total+ $77;
10 END.
```

Fig. 5.9 Le programme *alloue*.

et voici le code du programme:

```

2D4D E849DD      call    TU_PARAGRAPHS
2D50 0600        hex     0600 ; options
2D52 7B06        hex     7B06 ; cs
2D54 3F10        hex     3F10 ; ds
2D56 D902        hex     D902 ; code
2D58 2700        hex     2700 ; globales
2D5A 0004        hex     0004 ; min piles
2D5C 00A0        hex     00A0 ; max piles
2D5E 1000        hex     1000 ; fichiers
2D60 0000        hex     0000 ; input
2D62 0000        hex     0000 ; output

2D64 8BEC        mov     bp, sp
2D66 E814DF      call    TU_TAILLE_CODE
2D69 1800        hex     1800 ; taille
2D6B 6D2D        adr     SAUT_PRINCIPAL

2D6D E90000      SAUT_PRINCIPAL jmp    PRINCIPAL

2D70 A16202      PRINCIPAL   mov     ax, [$0262]
2D73 057700      add     ax, $0077
2D76 A36A02      mov     [$026A], ax
2D79 E90000      jmp     FIN

2D7C 33C0        FIN        xor     ax, ax
2D7E E8BFDE      call    TU_HALT
```

Fig. 5.10 Le code du programme *alloue*.

Notez que:

- la taille utilisée par chaque type dépend de l'implémentation, et a été décrite en détail au chapitre 2.

Les déclarations de variables ne génèrent donc aucun code. L'utilisation de ces variables dans des instructions (des expressions, des affectations ou des appels de procédures) utilise alors les attributs pour générer le code.

5.2.6 LES INSTRUCTIONS PASCAL

Les instructions PASCAL sont:

- l'affectation
- les instructions de contrôle (tests, répétitions, sauts)
- les appels de procédures et fonctions utilisateur
- les appels de procédures et fonctions prédéfinies

Le code généré par le compilateur pour chaque type d'instruction est une combinaison:

- d'instructions 8088
- d'appels aux routines de la librairie TURBO PASCAL (qui appellent éventuellement des routines MS-DOS ou des routines du BIOS)

Nous allons donc examiner ces combinaisons pour les différentes classes d'instructions et les différents types de données. Nous présenterons dans le détail:

- le traitement des variables simples
- le traitement des variables structurées
- le traitement des fichiers
- les instructions de contrôle

Dans chaque cas, nous présenterons les instructions qui permettent de comprendre le code généré par le compilateur. Lorsque plusieurs types d'instructions génèrent le même code, nous n'en présenterons qu'une. Par exemple l'addition et la soustraction de réels ne diffèrent que par la routine de la librairie utilisée (routine d'addition ou routine de soustraction). Toutes les routines de la librairie seront par contre présentées au chapitre 8.

5.3 ENTIERS, CARACTERES, BOOLEENS, ENUMERES, REELS

5.3.1 LES ENTIERS

5.3.1.1 ARITHMETIQUE ENTIERE

Voici un programme manipulant des entiers: