Chapitre 1: **Introduction.**

I. Structure d'un programme.

- L'en-tête
- Les déclarations (var, const, types, procedures, functions..)
- **begin**
- le programme
- **end.**

Exercice 1 : écrire un programme écrivant bonjour à l'écran.

II. Les types de variables.

- **Shortint :** entier compris entre -127 et 128
- **Byte :** entier compris entre 0 et 255
- **Integer :** entier compris entre -32 768 et 32 767
- **Word :** entier compris entre 0 et 65535
- **Longint :** entier long compris entre – 2 147 483 648 et 2 147 483 647
- **Real :** réel compris entre 2,9 e 10^{-39} et 1,7 e 10^{38} avec 11 décimales
- **Double :** réel double précision compris entre 5,0e10^{-324} et 1,7e10^{308} avec 15 décimales
- **Extended :** réel compris entre 1,9e 10^{--4951} et 1,1e 10^{-4932}
- **Char :** caractère alphanumérique
- **String :** chaîne de caractères.
- **Boolean :** valeurs logiques égales à **TRUE** ou **FALSE**

Exercice 2 :

- Calculer la somme de deux nombres entrés au clavier
- Calculer la somme de trois nombres entrés au clavier
- Calculer la somme de cinq nombres entrés au clavier en n'utilisant que deux variables.

III. Affectation.

Instruction : **:=**

A:=3 -----> A prend la valeur 3

B:=A -----> B prend la valeur de A, ici 3

Exercice 3 : a) Indiquer ce que contiennent les cases mémoires associées aux variables alpha, beta, gamma, I et n au cours de l'exécution du programme suivant : (UTILISER LE MODE PAS À PAS)

```
Program escalope;
var alpha, beta, gamma : integer;
I,n : boolean;
begin
      alpha:=1;
      beta:=alpha+1;
      gamma:=2*beta-3;
      beta:=beta+1;
      alpha:=alpha div 2;
      I:=true;
      n:=false;
      n:=(true)or(false);
```

```
            write('alpha ',alpha,' beta ',beta,' gamma ',gamma,' I ',I,' n ',n)
      end.
```

b) écrire un programme permutant deux variables a et b.

c) écrire un programme calculant l'aire d'un disque, et son périmètre dont on fournit le rayon (noter qu'en Turbo-Pascal, il existe une constante prédéfinie pi).

IV. Les entrées sorties conversationnelles.

1°) write et writeln.

Exemple 2 : taper le programme suivant et observer les sorties écran

```
program Affichage_formate ;
var n, p : integer ;
x, y : real ;
cl, c2 : char ;
ok : boolean ;
begin

      n := 3 ; p := 125 ;
      x :=1.23456e2 ; y := 2.0 ;
      cl := 'e' ; c2 := 'i' ;
      ok := false ;
      {1} writeln ('nombre', n:4) ;
      {2} writeln (n:3, p:5) ;
      {3} writeln (c1:3, c2:6, ok:7) ;
      {4} writeln (p:2) ;
      {5} writeln ('bonjour':3, ok:2) ;
      {6} writeln (x:20) ;
      {7} writeln (x:10) ;
      {8} writeln (x:2) ;
      {9} writeln (x:12:4) ;
      {10} writeln (x:10:1) ;
      {11} writeln (x:8:5)
end.
```

Exemple 3 :

```
Program tartenpion;

begin
      write('je ','ne ');
      write('vais pas à la ligne');
      writeln;
      writeln('je ','vais ');
      write('à la ligne')
end.
```

2°) Read et readln :

cf programme précédent.

Exercice 3 : écrire un programme demandant nom, prénom, âge, taille en cm, et répondant : "bonjour prénom nom, tu as, âge ans, et tu mesures taille m".

Chapitre 2 : Les tests

I. Exemple n°1 :

```
program Exemple_d_instruction_if_2 ;
var n, p, max : integer ;
begin
      writeln ('donnez 2 nombres entiers') ;
      readln (n, p) ;
      if n < p then
            begin
            max := p ;
            writeln ('croissant')
            end
      else
            begin
            max := n ;
            writeln ('décroissant')
            end;
      writeln ('le maximum est ', max)
end.
```

Remarque : dans toute la suite, on entend par instruction, une instruction simple, ou une suite d'instructions commençant par **begin** et finissant par **end**.

II. Syntaxe générale de l'instruction if :

Elle peut prendre une de ces deux formes :

```
If expression_booléenne   then instruction

If expression_booléenne   then instruction_1
                          else instruction_2
```

Un **else** se rapporte toujours au dernier **then** rencontré auquel un **else** n'a pas encore été attribué.

III. Exemple n°2 :

```
program Exemple_d_instruction_case_l
var n : integer
begin
      write ('donnez un entier ') ;
      readln (n)
      case n of
            1, 2 : writeln ('petit') ;
            3..10 : writeln ('moyen') ;
            11..50 : writeln ('grand')
      else
            writeln ('très grand')
      end ;
      write ('au revoir')
end.
```

IV. Syntaxe générale de l'instruction case of :

Elle peut prendre une de ces deux formes :

```
case variable of
          domaine_1: instruction_1;
          domaine_2: instruction_2;
          ...
          domaine_n: instruction_n
end

case variable of
          domaine_1: instruction_1;
          domaine_2: instruction_2;
          ...
          domaine_n: instruction_n;
     else instruction_bis
end
```

V. Exercices :

1) Quelles erreurs ont été commises dans chacune des instructions suivantes:

a) if a<b then x := x+1 ; else x := x-l

b) if a<b then x := x+1 ; y := b end else x := x-1 ; y := a end c) if n := 0 then p := 1

2) Que fait cette partie de programme:
if a<b then writeln ('merci') ; writeln ('croissant')

3) Soient trois variables réelles a, b et c et une variable booléenne nommée ordre. Placer dans ordre la valeur true si les valeurs de a, b et c prises dans cet ordre sont rangés par valeurs croissantes (a <= b <= c) et la valeur false dans le cas contraire. On cherchera deux solutions :

a) l'une employant une instruction if,

b) l'autre n'employant pas d'instruction if.

4) Écrire un programme réalisant la facturation d'un article livré en un ou plusieurs exemplaires. On fournira en données le nombre d'articles et leur prix unitaire hors-taxe. Le taux de TVA sera toujours de 20,6%. Si le montant TTC dépasse 1000F, on établira une remise de 5%. On cherchera à ce que le dialogue se présente ainsi:

nombre d'articles : 27
prix unitaire ht : 248.65
montant TTC : 8096,54
remise : 404,83
net à payer : 7691,71

5) Dire si chacune des instructions **case** schématisées ci-dessous est correcte ou non et, le cas échéant, identifier l'erreur commise. Nous supposerons pour chaque cas ces déclarations:
const nb = 100 ; var n, p : integer ;

a)
```
case n of
     1..3 : ....
     4..nb : ....
end
```

b)
```
case n of
     1..5 : ....
     p : ....
end
```

c)
```
case n of
     1..nb : ....
     nb+1..maxint : ....
end
```

d)
```
case n of
     -nb..0 :  ....
     nb,nb+1 : ....
end
```

6) Que fait cette instruction **case** lorsque la variable entière *n* contient l'une de ces valeurs :

a) 2 b) 5 c) 12 d) 20 c) 30

```
case n of   1..3, 11..19, 21 : write ('premier lot') ;
            4..20 : write ('deuxième lot')
            else write ('autre')
end
```

7) Écrire un programme qui lit trois notes d'examen et leur coefficient, puis affiche la moyenne en précisant "éliminé" si la moyenne est inférieure à 10, "admissible" dans la cas contraire.

8) Écrire un programme permettant la résolution de $ax^2+bx+c=0$ dans $\mathbb{R}$.

Chapitre 3 Les structures de répétitions

I. Exemples :

```
     1°)
program utilisation_du_compteur1;
var n : integer;
begin
     for n:=1 to 8 do
          writeln(n,' a pour triple ',3*n)
end.
     2°)
program utilisation_du_compteur1;
var n : integer;
begin
     for n:=8 downto 1 do
          writeln(n,' a pour triple ',3*n)
end.
```

II. Syntaxe de l'instruction for :

```
for compteur:= début  to fin do instruction

for compteur:= fin  downto début do instruction
```

Exercices :

1°) Écrire un programme affichant l'alphabet complet.
2°) Écrire un programme affichant tous les nombres de 1 à 41 de 5 en 5.
3°) Écrire un programme calculant la puissance entière relative d'un nombre réel quelconque.

III. Exemple :

```
Program exemple_boucle_jusqu_a;
var n : integer;
begin
     repeat
          write('Donner un entier positif : ');
          readln(n);
     until n<0;
     writeln('vous venez de taper un nombre négatif')
end.
```

IV. Syntaxe de l'instruction repeat ... until :

```
repeat
     instruction_1;
     instruction_2;
     ...
     instruction_n;
until expression_booléenne
```

Exercice : 4°) Écrire un programme calculant la puissance entière relative d'un nombre réel quelconque.

V. Exemple :

```
program Exemple_boucle tant_que ;
var  somme : integer ;
          nombre : integer ;
begin
somme := 0 ;
     while somme < 100 do
          begin
               write ('donnez un nombre ') ;
               readln (nombre) ;
               somme := somme + nombre
          end ;
writeln ('somme obtenue : ', somme)
end.
```

VI. Syntaxe de l'instruction while ... do :

```
while expression_booléenne  do instruction
```

Exemple : 5°) Écrire un programme calculant la puissance entière relative d'un nombre réel quelconque.

VII. Remarque importante : POUR INTERROMPRE UN PROGRAMME

Avec les structures de boucle, le risque existe d'écrire, par mégarde, un programme qui ne s'arrête pas ou qui ne fonctionne pas comme prévu et que l'on aimerait pouvoir interrompre.

La combinaison de touches Ctrl/Pause permet cette interruption (dans certains cas, il faudra la presser deux fois). Elle vous propose une boîte de dialogue vous informant de l'arrêt du programme ; une simple validation vous ramène alors en fenêtre d'édition. Notez bien toutefois que, si vous cherchez à relancer votre programme, son exécution se poursuivra à partir de l'endroit où il avait été interrompu; elle ne reprendra pas depuis le début. Pour éviter cela, il vous suffit de faire appel à la commande Run/Program Reset avant de relancer votre programme.

La combinaison de touches Ctrl/C permet, dans certains cas seulement, d'interrompre définitivement votre programme. Cependant, elle ne peut être prise en compte que lors d'entrées-sorties, ce qui signifie que si votre programme "boucle" sur des calculs, sans rien afficher, votre frappe de Ctr/C ne sera jamais prise en compte.

N'oubliez pas que, de toute façon, certains "plantages" de votre programme ne vous laisseront pas d'autres ressources que d'effectuer un "redémarrage à chaud" de votre machine (touches Ctrl/Alt/del). Ceci confirme l'intérêt qu'il y a d'effectuer systématiquement une sauvegarde d'un programme (modifié) avant de l'exécuter.

VIII. Exercices :

6°) Calculer la moyenne de notes fournies au clavier avec un "dialogue" se présentant ainsi:
combien de notes : 4
note 1 : 12
note 2 : 15.25
note 3 : 13.5
note 4 : 8.75
moyenne de ces 4 notes : 12.37

7°) Calculer la somme d'un nombre quelconque de nombres positifs entrés au clavier, le dernier entré, non compté, étant -1

8°) Calculer la somme des entiers compris entre 1 et N.

9°) Calculer la somme des entiers pairs compris entre 1 et N.(facultatif)

10°) Vérifier qu'un nombre est premier.

11°) Calculer le nombre M tel que $\sum_{k=1}^{M} \frac{1}{k} > N$

12°) Deux joueurs lancent chacun un dé. Le joueur qui a le plus grand résultat marque un point. Le jeu s'arrête quand un des joueurs atteint un total de 11 points.

Écrire un programme simulant ce jeu et indiquant le gagnant (l'ordinateur simulera les valeurs obtenues par chaque joueur à chaque coup).

Modifiez le programme précédent pour que l'on puisse recommencer ou arrêter suivant la réponse à la question « voulez-vous recommencer ?».

Chapitre 4 :

Les types définis par l'utilisateur.

I. Introduction :

Dans le chapitre 1, nous avons introduit la notion de type scalaire (ou simple) et nous vous avons présenté quatre types scalaires prédéfinis (integer, real, char et boolean). Nous allons voir qu'il est possible de définir ses propres types scalaires.

Pourquoi, direz-vous, définir vos propres types scalaires ? En fait, les types prédéfinis permettent de manipuler aisément des nombres ou des caractères. Mais, vous pouvez être amené à traiter d'autres sortes d'informations, par exemple: des mois de l'année (janvier, février...), les jours de la semaine (lundi, mardi...), des notes de musique, des noms de cartes à jouer (as, roi, dame...), des marques de voiture, etc. Bien entendu, vous pouvez toujours dans ce cas "coder" I'information correspondante, par exemple 1 pour janvier, 2 pour février... ou encore 'A' pour as, 'R' pour roi... Mais Pascal vous permet de manipuler ces informations d'une manière plus naturelle en leur attribuant un nom de votre choix tel que janvier ou février. Pour ce faire, il vous suffira de fabriquer ce que l'on appelle un type défini par énumération (ou plus brièvement type énuméré), c'est-à-dire un type dans lequel vous "énumérez" chacune des valeurs possibles.

Par ailleurs, il arrive fréquemment que l'on ait à manipuler des informations dont les valeurs ne peuvent couvrir qu'un intervalle restreint de valeurs. Par exemple, un âge pourrait être un entier compris entre 0 et, disons... 150. Une note d'élève pourra être un entier compris entre 0 et 20. Une lettre minuscule sera un caractère compris entre 'a' et 'z'. Lorsque l'ensemble des valeurs possibles est ainsi restreint, il peut être intéressant de pouvoir en tenir compte dans le programme. Pascal vous autorise ainsi à définir de nouveaux types comme intervalles d'un autre type scalaire, œ qui vous permet de bénéficier:

– d'une meilleure lisibilité du programme,

– d'un contrôle des valeurs qui seront affectées aux variables de ce type,

– d'un gain de place mémoire lié au fait que, la plupart du temps, les variables d'un type intervalle occuperont moins de place que les variables du type initial (dit type "hôte" ou type "de base").

II. Types énumérés

```
program Exemple_de_type_enumere ;
type jour = (lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche);
var date : jour ;

begin
   for date:= lundi to dimanche do
      begin
         writeln ('Voici un nouveau jour');
         if date = mercredi then
            writeln ('--- les enfants sont en congé');
         if date = vendredi then
            writeln ('--- dernier jour de travail');
         if (date=samedi) or (date=dimanche) then
            writeln ('--- on se repose')
      end
end
```

Règle concernant la déclaration d'un type énuméré :

```
type nom_du_type = (identificateur_1,identificateur_2, ....,
identificateur_n)
```

Bien entendu, vous avez toute latitude dans le choix des identificateurs, aussi bien pour le type que pour les différentes valeurs de ce type.

Néanmoins, ne perdez pas de vue les remarques suivantes:

1) Un identificateur ne peut être un mot réservé; ainsi, pour déclarer un type "note de musique", vous ne pouvez pas écrire:

type note = (do, re, mi, fa, sol, la, si)

car do est un mot réservé (vous pourrez utiliser, par exemple, doo).

2) Un même identificateur ne peut pas désigner plusieurs choses différentes. Supposez que vous souhaitiez définir en plus du type jour déjà rencontré, un type jour_travail correspondant aux jours de la semaine. Vous songez (peut-être) à procéder ainsi:

type jour = (lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche) ;
jour travail = (lundi, mardi, mercredi, jeudi, vendredi) ;

Cela serait rejeté par Pascal puisqu'un identificateur tel que mardi représenterait deux entités de types différents. Bien entendu, dans votre esprit, le type jour_travail n'est pas totalement différent du type jour; il est plutôt "inclus" dedans (mais Pascal ne peut pas le deviner !). Vous verrez que le type intervalle vous permettra de venir à bout de ce problème.

3) Une constante n'est pas un identificateur. Autrement dit, il n'est pas question de déclarer un type par une énumération de nombres comme:

type impair = (1, 3, 5, 7, 9, 11)

ou même un type voyelle par:

type voyelle - ('a' , 'e', 'i', 'o', 'u', 'y')

4) En Pascal, tout identificateur doit avoir été déclaré avant d'être utilisé. Autrement dit, la déclaration de type doit être effectuée avant la déclaration des variables de ce type.

III. Type intervalle :

Abordons maintenant la deuxième sorte de type défini par l'utilisateur, à savoir le type intervalle.

Voici, par exemple, comment déclarer un type nommé age dont les valeurs seraient des entiers compris entre 0 et 150 :

type age = 1 .. 150 ;

ou encore, en utilisant une constante définie avant la déclaration de type:

const age_max = 150 ;
. . .
type age = 1 .. age_max ;

Il est alors ensuite possible de déclarer des variables du type age, par exemple:

var age_pere, age_mere, courant : age :

Notez bien que le type age n'est plus, à proprement parler, un nouveau type; ses valeurs appartiennent simplement à un intervalle d'un type déjà défini (et même ici prédéfini), à savoir le type entier. Ainsi, des variables du type age pourront être manipulées de la même manière que des variables entières, c'est-à-dire être lues ou écrites ou intervenir dans des calculs.

Voici des exemples corrects (n étant supposée entière):

```
read (age_pere, age_mere) ;
n := age_pere + age_mere ;
courant := age_pere + 10 ;
```

Notez bien que age_pere + age_mere est une expression de type entier. Sa valeur peut dépasser 150, sans que cela ne pose de problème.

Une seule restriction est imposée aux variables du type âge : elles ne pourront se voir affecter de valeurs sortant de l'intervalle prévu.

Ainsi, en principe, l'affectation courant := age_pere + 10 entraînera une erreur si la valeur de age_pere + 10 est supérieure à 150. Il en irait de même si, en réponse à l'instruction read précédente, l'utilisateur fournissait une valeur en dehors des limites 0 .. 150.

L'exemple précèdent définissait un type intervalle à partir d'un type entier. On dit que le type hôte (ou type de base) est le type entier. Mais le type hôte peut être n'importe quel type ordinal.

Voici deux exemples de types intervalle définis à partir d'un type énumération, à savoir le type jour du paragraphe 2:

```
type_jour=(lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche) ;
type jour_travail=lundi .. vendredi ;
type week_end=samedi .. dimanche ;
```

Ainsi, les valeurs du type jour_travail sont les 5 constantes: lundi, mardi, mercredi, jeudi et vendredi. Celles du type week_end sont les deux constantes samedi et dimanche.

Là encore, les variables déclarées de type jour_travail ou week_end pourront être manipulées comme n'importe quelle valeur du type jour. Ainsi, en déclarant:

var aujourd_hui : jour_travail ; courant : jour ;

ces affectations seront correctes:

aujourd_hui:= vendredi ;

courant:= succ (aujourd_hui)

Notez bien que courant prendra la valeur samedi. En effet, bien que aujourd_hui soit de type jour_travail, le résultat de succ(aujourd hui) est du type hôte du type jour_travail, à savoir, du type jour et donc, samedi est une valeur acceptable pour cette expression ainsi que pour la variable courant.

Règle concernant la déclaration d'un type intervalle :

```
type nom_du_type = début_de_l_intervalle...fin_de_l_intervalle
```

IV. Les types tableau :

Supposez que nous souhaitions déterminer, à partir de 20 notes fournies en donnée, combien d'élèves ont une note supérieure à la moyenne de la classe. Pour parvenir à un tel résultat, nous devons:

– déterminer la moyenne des 20 notes, ce qui demande de les lire toutes,

– déterminer combien, parmi ces 20 notes, sont supérieures à la moyenne précédemment obtenue.

Vous constatez que si nous ne voulons pas être obligé de demander deux fois les notes à l'utilisateur, il nous faut les conserver en mémoire. Pour ce faire, il paraît peu raisonnable de prévoir 20 variables différentes (méthode qui, de toute manière, serait difficilement transposable à un nombre important de notes). Le type tableau va nous offrir une solution convenable à ce problème, à savoir:

– par des déclarations appropriées, nous choisirons un identificateur unique (par exemple notes) pour repérer notre ensemble (dit alors tableau) de 20 notes;

– nous pourrons accéder individuellement à chacune des valeurs de ce tableau, en la repérant par un "indice" (ou index) précisant sa position dans le tableau.

Voici le programme complet résolvant le problème posé:

```
program Exemple tableau 1 ;

const nb_eleves - 7 ;
type tab_notes = array [1..nb_eleves] of real ;
var  notes:tab_notes ;
     i,nombre:integer ;
     somme,moyenne:real ;
begin
     writeln ('donnez vos ',nb_eleves,' notes')
     for i:=1 to nb_eleves do readln(notes[i]);
     somme:= 0.0;
     for i:= 1 to nb_eleves do somme:= somme + notes[i] ;
     moyenne:= somme/nb_eleves;
     nombre:= 0;
     for i:= 1 to nb_eleves do if notes[i]> moyenne then nombre:= nombre+1
     writeln('moyenne de ces ', nb eleves, ' notes', moyenne:8:2 );
     writeln(nombre,' eleves ont plus de cette moyenne')
end.

     donnez vos 7 notes :
     11  12.5  13  5  9  13.5  10
     moyenne de ces 7 notes 10.57
     4 élèves ont plus de cette moyenne
```

Syntaxe de la déclaration d'un type tableau :

```
type nom_du_tableau =  array[type_indices]  of type_éléments
```

Cas des tableaux à plusieurs indices :

```
array[type_indice_1] of [type_indice_2] of [type_indice_2] of type_element

array[type_indice_1, type_indice_2,type_indice_3 ] of type_element
```

Deux façons de désigner un éléments d'un tableau à plusieurs indices :

```
identificateur [indice_1,indice_2,indice_3]

identificateur [indice_1][indice_2][indice_3]
```

V. Exercices :

1°) Remplir un tableau avec des nombres aléatoires, puis calculer le maximum et le minimum de ce tableau.

2°) Même chose que le précédent avec en plus les numéros des cases du tableau où se trouvent le max et le min.

3°) Écrire un programme permettant de déterminer les k premiers nombres premiers (1 exclu), la valeur de k étant fixé dans le programme par une instruction const. On conservera les nombres premiers dans un tableau, au fur et à mesure de leur découverte, et on utilisera la remarque suivante pour décider si un nombre entier est premier : n est premier s'il n'est divisible par aucun nombre premier (1 exclu) inférieur ou égal à la racine carrée de n.

5°) Réaliser un programme de "tri alphabétique" de mots fournis au clavier. Le nombre de mots sera prévu dans une instruction const et chacun d'entre eux ne pourra comporter plus de 20 caractères.

6°) Réaliser un programme remplissant aléatoirement un tableau 10*10 des 6 nombres ,1 à 6, puis calculer la moyenne de sortie de chaque nombre.

7°) Écrire un programme permettant de calculer la multiplication de 2 tableaux de même dimension n*n; la multiplication étant définie ainsi :

cij étant l'élément du tableau (ième ligne j ième colonne) résultat de la

multiplication des deux tableaux de base.

Chapitre 5 :

Les procédures et les fonctions

Comme tous les langages, Pascal permet de découper un programme en plusieurs parties (nommées souvent "modules"). Cette "programmation modulaire" se justifie pour différentes raisons:

- Un programme écrit d'un seul tenant devient difficile à comprendre dès qu'il dépasse une ou deux pages de texte. Une écriture modulaire permet de le scinder en plusieurs parties et de regrouper dans le "programme principal" les instructions en décrivant les enchaînements. Chacune de ces parties peut d'ailleurs, si nécessaire, être décomposée en modules plus élémentaires; ce processus de décomposition pouvant être répété autant de fois que nécessaire. Il est à noter que les méthodes de "programmation structurée" conduisent tout naturellement à ce découpage.

- La programmation modulaire permet d'éviter des séquences d'instructions répétitives. En particulier, nous verrons comment la notion d'argument permet de "paramétrer" certains modules.

- La programmation modulaire permet le partage d'outils communs qu'il suffit d'avoir écrits et mis au point une seule fois.

I. LA NOTION DE PROCEDURE

Vous avez déjà été amené à utiliser des "fonctions prédefinies" telles que sqr,sqrt... Vous avez pu constater que cette notion de fonction en Pascal est très proche de la notion mathématique classique. En particulier, une fonction possède généralement un ou plusieurs arguments et elle fournit un résultat. Quant à l'aspect modulaire, on peut dire que le fait de mentionner le nom d'une fonction dans un programme entraîne l'appel de tout un ensemble d'instructions (module) en réalisant le calcul.

La procédure, quant à elle, n'est rien d'autre qu'une généralisation de la notion de fonction. Elle peut, elle aussi, posséder des arguments; en revanche, elle peut, en retour, fournir un ou plusieurs résultats, ou même aucun. Mais ces résultats (lorsqu'ils existent) ne constituent pas nécessairement le seul "travail" de la procédure; par exemple, cette dernière peut imprimer un message. On peut dire de la procédure qu'elle réalise une "action", terme plus général que calcul. Notez que nous avons déjà été amené à utiliser des procédures "prédefinies" pour les entrées-sorties (read, write) et pour les manipulations de chaînes.

En ce qui concerne la notion d'argument, nous verrons que celle-ci est plus générale qu'en mathématiques; en particulier, elle interfère avec les notions de "variables globales et locales". C'est pourquoi, nous allons commencer par vous présenter une succession d'exemples introduisant progressivement les notions fondamentales (variables globales, variables locales, arguments muets et effectifs, arguments transmis par valeur, arguments transmis par adresse). Nous parlerons des procédures en premier, les fonctions apparaissant ensuite comme un cas particulier.

II PREMIER EXEMPLE DE PROCEDURE

Considérez ce programme:

```
program Exemple procedure_1 ;
var i : integer ;
procedure pessimist ;
begin
writeln ('il ne fait jamais beau')
```

```
end ;

begin for i := 1 to 5 do pessimist ;
writeln ('du moins, on le croit')
end.
```

Après l'en-tête du programme et les déclarations usuelles (limitées ici à la déclaration de i), nous trouvons ce que l'on appelle une "déclaration de procédure":

```
procedure pessimist ;
begin
writeln ('il ne fait jamais beau')
end ;
```

Cette déclaration définit à la fois:

- le nom de la procédure à l'aide d'un en-tête ressemblant à un en-tête de programme,
- son action, à l'aide d'un "bloc" d'instructions; ici, ce bloc comporte simplement:

```
writeln ('il ne fait jamais beau')
```

Le programme proprement dit (dit parfois "programme principal") vient ensuite, écrit comme à l'accoutumée sous forme d'un bloc terminé par un point. La seule nouveauté réside dans le fait qu'il est possible d'y utiliser la procédure pessimist préalablement définie. Pour ce faire, il suffit d'en citer le nom pour former une nouvelle instruction Pascal. Ainsi, l'instruction pessimist demande simplement d'exécuter les instructions mentionnées dans la définition de la procédure pessimist (ici, il n'y a qu'une instruction...).

Comme cette instruction dite "instruction d'appel de procédure" apparaît ici au sein d'une boucle for, elle est répétée 5 fois, d'où l'affichage de 5 fois le message: "il ne fait jamais beau".

III - LES VARIABLES GLOBALES

Notre précédente procédure pessimist réalisait toujours la même chose. Or, comme vous le devinez, en général, l'intérêt d'une procédure résidera dans son aptitude à réaliser une action dépendant de certains "paramètres". Ces paramètres seront définis, non plus lors de l'écriture de la procédure elle-même, mais lors de son appel par le "programme appelant".

En Pascal, il existe deux méthodes radicalement différentes pour paramétrer une procédure: les variables globales d'une part, les arguments d'autre part. Dans l'exemple suivant, la (nouvelle) procédure optimist voit son action paramétrée par l'intermédiaire d'une variable globale nommée i.

```
program Exemple procedure_2;
var i : i integer ;

procedure optimist;
begin
writeln ('il ne fait jamais beau');
writeln (‘---- , i,  ‘fois';
end;
begin
```

```
for i := 1 to 5 do
optimist ;
writeln ('du moins, on le croit')
end .
```

Le programme principal est ici resté le même, tant au niveau de ses déclarations que de sa partie exécutable. La seule différence réside dans la présence, au sein de la procédure, de l'instruction:

```
writeln ('---- ', i, ' fois')
```

Vous constatez que la variable i a été déclarée dans la partie déclaration du programme principal. Pour l'instant, d'ailleurs, la définition de notre procédure ne comporte aucune déclaration qui lui soit propre (mais nous verrons bientôt que cela serait possible). Dans ces conditions, il vous paraît peut-être normal que l'identificateur i, lorsqu'il est employé dans la procédure optimist, désigne bien la variable entière i connue du programme principal.

Il en va bien ainsi en Pascal. Plus précisément, à partir du moment où un identificateur a été déclaré dans le programme principal, il est connu de toutes les procédures qui sont éventuellement déclarées par la suite.

Ici, donc, nous avons pu paramétrer l'action de la procédure pessimist par l'intermédiaire de la variable i.

Remarque: On dit souvent qu'une variable telle que i, connue à la fois du programme principal et de la procédure optimist, est une "variable globale". En fait, ce terme de global est quelque peu ambigu, dès que l'on utilise plusieurs niveaux de procédures car il est, "relatif". Nous y reviendrons.

IV LES VARIABLES LOCALES

En fait, dans notre précédent exemple, la variable i était considérée comme "globale" parce qu'aucune déclaration concernant i n'avait été place dans la procédure elle-même.

Or, il est tout à fait possible d'effectuer au sein d'une procédure, des déclarations (classiques) de types, de variables, d'étiquettes, de procédures... Dans ces conditions, les nouvelles "entités" (variables, types, étiquettes, procédures) ainsi déclarées ne sont connues qu'à l'intérieur de la procédure dans laquelle elles ont été déclarées. On dit qu'elles sont "locales" à la procédure, ou encore que leur "portée est limitée à la procédure".

Voyez cet exemple:

```
program exemple procedure_3 ;
var cl, c2 : char ;
procedure tricar ;
     var c : char ;
     begin
     if c1>c2 then begin
               c := c1;
               c1:=c2;
               c2:=c;
               end;
end ;
begin
     write ('donnez 2 caractères : ') ;
     readln (cl, c2) ;
```

```
     tricar ;
     write ('caractères tries : ') ;
     writeln (cl,c2)
end.
```

Le rôle de la procédure tricar est simplement de ranger par ordre alphabétique les caractères contenus dans les deux variables globales cl et c2 en procédant, si nécessaire, à un échange de leurs valeurs. Pour ce faire, elle utilise la variable c comme variable intermédiaire. Or, vous constatez que c a été déclarée au sein de la procédure tricar. Cette fois, c n'est connue qu'au sein de tricar; on dit que sa "portée est limitée à la procédure tricar" ou encore que "c est locale à tricar".

Que se passerait-il si l'on cherchait à utiliser c en dehors de tricar, par exemple, en ajoutant une simple instruction telle que:

```
     write (c)
```

après l'appel de tricar dans le programme principal. Le compilateur signalerait simplement que l'identificateur c n'est pas connu à ce niveau.

Si, en revanche, nous déclarions, en plus de la déclaration faite dans tricar, une variable c dans le programme principal, celle-ci serait indépendante de la variable c connue dans tricar. Voyez cet exemple qui reprend le précèdent auquel nous avons ajouté la déclaration:

```
     var c : char
```

ainsi que les deux instructions:

```
     c := 'A' ;
     writeln ('et dans c : ', c)
```

qui permettent d'affecter une valeur à la variable globale c avant l'appel de tricar et de l'afficher après cet appel, afin de montrer qu'elle n'a effectivement pas été modifiée:

```
program exemple_procedure 4 ;
var cl, c2 : char ;
     c : char ;

procedure tricar ;
var c : char ;
     begin
     if c1 > c2 then begin
               c := c1; c1 := c2 ; c2 := c ;
               end;
end ;

begin
     c := 'A';
     write ('donnez 2 caracteres : ') ;
     readln (c1, c2) ;
     tricar ;
     write ('caracteres tries     :');
     writeln (c1,c2) ;
     writeln (' et dans c : ', c)
end.
```

V LES ARGUMENTS TRANSMIS PAR VALEUR

Lorsque nous avons évoqué la possibilité de paramétrage d'une procédure, nous avons précisé qu'il existait deux façons de le faire en Pascal: à l'aide de variables globales ou à l'aide d'arguments. Nous venons de voir ce que sont les variables globales et nous vous proposons maintenant d'examiner la notion d'argument. Voyez ce nouvel exemple:

```
program exemple_procedure_5 ;
type note = (ut, re, mi, fa, sol, la, si) ;
var n1 : note ;
     i : integer ;

procedure imprime_note(n : note) ;
       begin
          case n of
          ut : write ('do ') ;
          re : write ('re ') ;
          mi : write ('mi ') ;
          fa : write ('fa ') ;
          sol : write ('sol ') ;
          la : write ('la ') ;
          si : write ('si ')
          end ;
     end ;

begin
     writeln ('voici les notes de la gamme') ;
     for n1 := ut to si do
               imprime note (nl) ;
     writeln ;
     write ('donnez un numero : ') ;
     readln (i) ;
     write ('la ', i, ' eme note est ') ;
     imprime_note (note(i)) ;
     writeln
end.
```

Avant de l'examiner en détail, portons d'abord notre attention sur la définition de la procédure imprime_note, et plus particulièrement sur son en-tête:

```
procedure imprime note (n : note) ;
```

Cette fois, celle-ci comporte, en plus du nom de procédure (imprime_note), une indication signifiant que cette procédure possède un **argument** nommé n, de type note (notez que ce type est en fait "défini" dans le programme principal). Cela signifie que lorsque cette procédure sera appelée, on lui transmettra une valeur de type note. Quant à l'usage qu'elle doit faire de cette valeur, celui-ci lui est spécifié au sein des instructions mêmes, sachant que c'est l'identificateur n qui désigne la dite valeur.

Notez bien que n n'a aucune signification en dehors de la procédure, ni aucun rapport avec une éventuelle variable globale de même nom. On peut dire que n se comporte un peu comme une variable locale à la procédure, avec cette différence importante que sa valeur proviendra de l'extérieur. On peut dire aussi que n est un "**argument muet**", autrement dit que ce nom en soi n'a aucune importance et que la même procédure pourrait être définie en remplaçant n par n'importe quel autre identificateur (à chaque endroit où apparaît n, bien sûr, et non seulement dans l'en-tête). Il

s'agit là du même principe que lorsque nous définissons une fonction en mathématiques: nous pouvons indifféremment définir f par $f(x) = ax^2 + bx = c$ ou par $f(u) = au^2 + bu + c$; x ou u sont des variables muettes.
Quant au rôle de notre procédure, il se limite à l'affichage "en clair" de la valeur de type note qu'on lui transmet.

En ce qui concerne la manière d'utiliser la procédure imprime_note, vous constatez que nous faisons suivre son nom d'une expression de type note entre parenthèses. Par exemple:

```
imprime_note (n1)
```

appelle la procédure imprime_note, en lui transmettant la valeur de n1. De même:

```
imprime_note (fa)
```

appelle la même procédure en lui transmettant la valeur (constante cette fois) fa.

Cette fois, nous dirons que n1 ou fa sont les arguments effectifs de l'appel de la procédure. Ces arguments ne sont plus muets comme dans la définition de la procédure puisqu'ils ne représentent plus quelque chose d'indéfini mais, qu'au contraire, ils ont une valeur bien précise.

VI LES DEUX MODES DE TRANSMISSION D'ARGUMENTS: PAR VALEUR OU PAR ADRESSE

Dans notre précédent exemple, nous avons indiqué que la valeur de l'argument était transmise à la procédure, mais nous n'avons pas précisé quel était le mécanisme de cette transmission. En fait, il existe en Pascal deux techniques différentes d'échange d'informations par arguments, à savoir la **transmission par valeur** et la **transmission par adresse**.
La transmission par valeur consiste à recopier la valeur à transférer au sein de la procédure elle-même. Cette dernière travaille alors avec cette copie, sans toucher en quelque sorte à la valeur d'origine.

Lorsque l'on ne spécifie rien de particulier, la transmission d'arguments se fait par valeur. Ainsi, dans notre précédent exemple, où l'en-tête de procédure était:

```
procedure imprime_note (n : note)
```

nous pouvons considérer que n représentait un emplacement de la procédure dans lequel se trouvait recopiée, à chaque appel, la valeur de l'argument effectif correspondant. Au fil du déroulement du programme, nous trouvions donc successivement à cet endroit: 7 fois la valeur de n1 (ut puis ré, puis mi...) et 1 fois la valeur fa.

Il faut bien noter que cette transmission par valeur est "à sens unique": s'il y a bien transfert d'informations lors de l'appel de la procédure, il n'y a, en revanche, aucun échange lors du retour de la procédure. Pour illustrer cette restriction, supposez que nous ayons modifié ainsi la procédure tricar du premier exemple du paragraphe 3:

```
procedure tricar (ca, cb : char) ;
      var c : char ;
      begin
          if ca < cb then begin
          c := ca ; ca := cb ; cb := c
```

```
            end
end;
```

Dans ce cas, lors de l'appel de tricar par:

```
tricar (c1, c2)
```

il y aura transfert des variables c1 et c2 dans les emplacements de la procédure. La "mise en ordre" se fera alors sur les valeurs de ces emplacements "locaux". Ainsi si avant de sortir de tricar, nous écrivions les valeurs de ca et cb, celles-ci seraient convenablement ordonnées. En revanche, lors du retour dans le programme principal, aucun échange symétrique du précédent n'aura lieu, de sorte que cl et c2 contiendront toujours leurs "anciennes" valeurs.

En résumé, donc, la transmission par valeur permet de fournir de l'information à la procédure, mais en aucun cas de recueillir un quelconque résultat.

En Pascal, la transmission par adresse va nous permettre de résoudre ce problème. Dans ce cas, en effet, lors de l'appel de la procédure, il y aura, non plus recopie d'une valeur, mais transmission de son "adresse", donc de son emplacement dans le programme appelant. Dans ces conditions, la procédure travaillera, non plus sur une copie, mais sur la valeur d'origine elle-même. Elle peut donc éventuellement la modifier.
Voyons maintenant comment indiquer à Pascal que la transmission de certains arguments doit se faire par adresse plutôt que par valeur.

VII LES ARGUMENTS TRANSMIS PAR ADRESSE

Supposons que nous souhaitions réaliser une procédure calculant la somme des valeurs d'un tableau. Les arguments en seraient donc:

- le tableau,
- la somme de ses valeurs.

En ce qui concerne le tableau, nous pouvons nous contenter d'une transmission par valeur puisque la procédure n'a pas à le modifier. Pour ce qui est de la somme, en revanche, il nous faut utiliser une transmission par adresse. Supposons que, dans le programme principal, nous ayons les déclarations suivantes:

```
const nb_valeurs = 10 ;
type ligne  = array [1..nb_valeurs] of integer ;
var a, b : ligne ;
```

Voici comment nous pourrions écrire notre en-tête de procédure:

```
procedure somme (t : ligne, var som : integer ) :
```

Vous voyez que, pour préciser que l'argument som est transmis par adresse, il nous suffit d'en faire précéder la déclaration du mot **var**.

Voici maintenant la liste complète de notre procédure, et deux exemples de son utilisation pour calculer la somme des dix premiers entiers, ainsi que celle de leurs carrés:

```
program Exemple_procedure 6 ;
const n_max = 10 ;
type ligne = array [1..n_max] of integer ;
```

```
var a, b : ligne ;
     i : integer ;
     sa, Sb : integer ;

procedure somme (t : ligne var som : integer ) ;
var i : integer
     begin
     som := 0;
     for i := 1 to n_max do
               som := som + t[i] ;
     end;

     begin
          for i := 1 to n_max do begin
               a[i] := i ; b[i] := sqr(i) ;
               end ;
     somme ( a, sa ) ;
     somme ( b, Sb ) ;
     writeln (' somme des ', n_max, ' premiers entiers : ', sa ) ;
     writeln (' somme des ', n_max, ' carres des premiers entiers : ', Sb)
end.
```

somme des 10 premiers entiers : 55
somme des 10 carres des premiers entiers : 385

Cette fois, lors de l'appel de la procédure somme, par exemple par:

```
somme (a, sa)
```

il y a :

- d'une part, recopie des valeurs du tableau a au sein de la procédure, dans l'emplacement désigné par t,

- d'autre part, transmission de l'adresse de la variable sa.. Ainsi, une instruction de la procédure telle que som: = O travaillera en fait directement sur la variable sa.

Remarques:

1) Dans l'en-tête de somme, nous avons utilisé l'identificateur ligne pour déclarer le type de t. Nous verrons que c'est d'ailleurs le seul moyen de déclarer le type d'un argument formel, lorsqu'il ne s'agit pas d'un type prédéfini. En effet, Pascal refuserait:

```
procedure somme (t : array [1..n_max] of integer, ... { incorrect }
```

2) Notre procédure somme utilise finalement:
- une variable globale: n_max,
- un argument transmis par valeur: t,
- un argument transmis par adresse: som.

3) La transmission par adresse n'est pas, à proprement parler, le symétrique de la transmission par valeur. On pourrait simplement dire qu'elle est plus générale puisqu'elle permet toujours, le cas échéant, de transférer une information qui ne serait pas modifiée par la procédure, donc de réaliser l'équivalent d'une transmission par valeur. Dans certains langages, en revanche, il existe une

technique de transmission parfaitement symétrique de la transmission par valeur, à savoir la transmission par résultat (il y a recopie d'un résultat lors du retour de la procédure).

4) Telle qu'elle se présente, notre procédure somme ne peut travailler que sur des tableaux du type ligne, c'est-à-dire sur des tableaux de 10 entiers. On pourrait souhaiter disposer d'une procédure capable de faire la somme de tableaux d'entiers de tailles différentes. Ceci n'est pas prévu par le Pascal standard.

VIII LA FONCTION: CAS PARTICULIER DE LA PROCEDURE

Considérons l'utilisation d'une fonction prédéfinie telle que sqr dans l'instruction:

```
a := sqr (b) + 3
```

La notation sqr(b) peut faire penser à un appel de procédure qui se nommerait sqr, à laquelle on fournirait en argument un valeur b. Cependant, si nous avions cherché à écrire une procédure réalisant le même calcul, nous aurions dû y prévoir un argument (transmis par adresse) pour recueillir le résultat. Voici, par exemple, une procédure nommée proc_sqr effectuant un tel travail:

```
procedure proc_sqr (nombre : integer ; var carre : integer ) ;
     begin
     carre := nombre * nombre
end ;
```

Si nous cherchons alors à réaliser l'équivalent de l'affectation précédente, nous voyons que nous sommes amenés à écrire:

```
proc_sqr (b, b2) ;
a := b2 + 3 ;
```

Si, donc, nous revenons à la notation sqr (b), nous constatons qu'elle désigne "implicitement" la valeur du résultat. Cela n'est bien sûr possible que parce que ce résultat est unique. Il est alors plus pratique, dans ce cas, d'utiliser une fonction plutôt qu'une procédure, puisque la première peut éventuellement être insérée dans une expression.

Nous venons de raisonner sur une fonction prédéfinie. En fait, de même que nous pouvions définir nos propres procédures, nous pouvons définir nos propres fonctions.

Nous pouvons dire que lorsqu'un "module" ne comporte qu'un seul résultat de type scalaire, nous avons le choix entre l'écrire sous forme d'une procédure ou d'une fonction.

IX EXEMPLE D'UTILISATION D'UNE FONCTION

Voici, à titre d'exemple, l'utilisation d'une fonction calculant la somme des valeurs d'un tableau; vous pouvez le comparer utilement à celui du paragraphe 7 qui résolvait le même problème à l'aide d'une procédure..

```
program Exemple_procedure 7 ;
const n max = 10 ;
type ligne = array [1..n_max] of integer ;
var a, b : ligne ;
     i : integer ;
```

```
     sa, Sb : integer ;
function somme ( t : ligne ) : integer ;
var i : integer ;
     som : integer ;
          begin
          som := 0 ;
          for i := 1 to n_max do
               som := som + t[i] ;
          somme := som ;
end ;

begin
     for i := 1 to n_max do begin
               a[i] := i ;
               b[i] := sqr(i) ;
               end ;
     sa := somme ( a ) ;
     Sb := somme ( b ) ;
     writeln (' somme des ', n_max, ' premiers entiers : ', sa ) ;
     writeln (' somme des ', n_ max, ' carres des premiers entiers : ', Sb)
end.
```

```
somme des 10 premiers entiers : 55
somme des 10 carres des premiers entiers : 385
```

Vous constatez tout d'abord que l'en-tête de procédure est devenu un en-tête de fonction:

```
function somme ( t: ligne ): integer;
```

Le terme procedure a été remplacé par celui de function. En ce qui concerne les arguments, vous remarquez que som n'apparaît plus dans la liste; en revanche, le mot integer a été ajouté à la suite. Cet en-tête précise, en définitive, le nom de la fonction, les arguments d'"entrée", qui correspondent aux valeurs qui lui seront transmises en argument et, enfin, le "type de la fonction" (ici integer), c'est-à-dire le type de l'unique résultat qu'elle fournira..

En ce qui concerne la description de la fonction elle-même, vous constatez la présence d'une variable locale nommée som (dans la procédure, il s'agissait d'un argument): elle servira au calcul de la somme désirée.

Enfin, l'affectation somme := som est quelque peu spéciale puisqu'on y trouve, à gauche du signe :=, le nom même de la fonction. Par convention, cette affectation sert à définir la valeur du résultat de la fonction. L'identificateur somme apparaît ainsi à la fois comme identificateur de fonction et comme identificateur de variable . Toutefois, dans ce dernier cas, il ne peut apparaître qu'à gauche d'une instruction d'affectation, et en aucun cas dans une expression. En particulier, nous n'aurions pas pu utiliser directement l'identificateur somme pour y cumuler progressivement la somme des éléments de notre tableau en écrivant:

```
somme : = somme + t [ i ].
```

X REGLES GÉNÉRALES D'ÉCRITURE DES PROCÉDURES ET DES FONCTIONS

Nous reprenons ici en les généralisant l'ensemble des règles qui président à l'écriture des procédures ou des fonctions. Celles relatives à leur utilisation seront étudiées dans le paragraphe suivant.

11.1 Structure générale

Chaque procédure est définie (déclarée) dans la partie déclaration du programme principal ou d'une autre procédure ou fonction. En Pascal standard, cette déclaration de procédure ne peut apparaître qu'après les autres déclarations (label, const, type, var). En Turbo Pascal, comme nous l'avons déjà mentionné, cette contrainte n'existe plus; néanmoins, ne perdez pas de vue qu'aucun identificateur ne peut être utilisé (donc en particulier, dans une déclaration de procédure) avant d'avoir été défini.
Par ailleurs, la portée d'un identificateur est limitée à la procédure où il est défini, ainsi qu'à celles qui sont internes à cette dernière. Bien entendu, les identificateurs définis dans le programme principal, sont connus partout.

11.2 L'en-tête et les arguments formels

L'en-tête d'une procédure ou d'une fonction précise:

- son nom,
- les arguments formels,
- leur type,
- leur mode de transmission (par valeur ou par adresse).

De plus, pour les fonctions, il spécifie le type du résultat.

Voici quelques exemples d'en-têtes corrects :

```
procedure ex1(n, p : integer ; c : real);

procedure ex2(n : integer ; var p : integer ; var x, y : real ; q : integer) ;
```

En revanche, ces en-têtes sont incorrects :

```
procedure ex3(n : 0..100; var x : real)
```

le type de n est explicité dans l'en-tête ; il faudrait le définir dans la procédure englobant et placer ici son identificateur.

```
function ex4(x : real ; c : char)
```

il manque le type de la fonction.

XII EXERCICES

1) Écrire une procédure permettant de déterminer si un nombre est premier. Elle comportera deux arguments : le nombre à examiner, un indicateur booléen précisant si ce nombre est premier ou non.

2) Écrire la procédure précédente sous forme de fonction.(facultatif)

3) Écrire une fonction calculant la norme d'un vecteur à 3 composantes réelles.

4) Écrire la fonction précédente sous forme d'une procédure.

5) Écrire une procédure supprimant tous les espaces d'une chaîne de longueur maximale de 50 caractères.

6) Écrire une fonction calculant le produit vectoriel de deux vecteurs à 3 composantes réelles.

7) Écrire une procédure simulant le lancé d'un dé à six faces.(facultatif)

8) Écrire un programme du jeu de 421 auquel joue deux joueurs en utilisant un maximum de procédures ou fonctions.

Chapitre 6

Les types Enregistrement

Parmi les types structurés dont dispose Pascal, nous avons déjà étudié le type tableau. Celui-ci nous permettait de réunir, au sein d'une même structure, des éléments de même type ayant un certain rapport entre eux; cela nous autorisait, au bout du compte, soit à manipuler globalement l'ensemble du tableau (dans des affectations), soit à appliquer un même traitement à chacun de ses éléments (grâce à la notion d'indice).

Mais, on peut également souhaiter regrouper au sein d'une même structure des informations n'ayant pas nécessairement toutes le même type; par exemple, les différentes informations (nom, prénom, sexe, nombre d'enfants...) relatives à un employé d'une entreprise.

En Pascal, le type enregistrement va nous permettre d'y parvenir. Ses différents éléments, nommés alors champs, pourront être de type quelconque; par exemple, un enregistrement correspondant à un employé pourra comporter les informations suivantes:

- nom, de type string[30],
- prénom, de type string[20]],
- sexe, de type boolean,
- nombre d'enfants, de type 0..50.

Comme les tableaux, les enregistrements pourront être manipulés soit globalement, soit élément par élément. En revanche, nous ne retrouverons pas la possibilité qu'offrait le tableau de répéter un même traitement sur les différents champs (celle-ci n'aurait d'ailleurs en général aucun sens, à partir du moment où les champs sont de nature différente).

Comme nous le verrons dans le prochain chapitre, le type enregistrement est fréquemment associé aux fichiers. Néanmoins, ce n'est pas là une règle générale: il existe des variables de type enregistrement n'ayant pas de rapport avec des fichiers et il existe des fichiers ne faisant pas appel à

un type enregistrement. C'est pour cette raison que nous vous proposons d'étudier ici la notion d'enregistrement indépendamment de la notion de fichier.

1 - EXEMPLES INTRODUCTIFS

Voyez ces déclarations:

```
type personne = record
     nom : string [30] ;
     prénom : string [20] ;
     masculin : boolean ;
     nbenfants : 0..50
end ;
var employe, courant : personne
```

Elles correspondent à l'exemple évoqué en introduction. Elles définissent tout d'abord un type nommé personne comme étant formé de 4 champs nommés nom, prénom, masculin et nb_enfants et ayant le type spécifié. Ensuite de quoi elles déclarent deux variables nommées employe et courant comme étant de ce type personne.

Ces variables pourront être manipulées champ par champ, de la même manière que n'importe quelle information ayant le type du champ correspondant. Pour ce faire, un champ d'une variable de type enregistrement est désigné par le nom de la variable, suivi d'un point et du nom du champ concerné. Par exemple:

`employe.nom` désigne le champ nom de l'enregistrement employe (il s'agit donc d'une information de type string[30])),

`courant.masculin` désigne le champ masculin de l'enregistrement courant (il s'agit donc d'une information de type boolean).

Voici quelques exemples d'instructions faisant référence à certains champs de nos enregistrements. personne et courant:

```
readln (employe.nom)
employe.nbenfants := 3
if courant.masculin then ...
```

Par ailleurs, les variables employe et courant pourront être manipulées globalement; ainsi:

```
courant := employe
```

recopie toutes les valeurs des différents champs de emplaye dans les champs correspondants de courant. Elle remplace (avantageusement) les 4 affectations:

```
courant.nom := employe.nom
courant.prenom := employe.prenom
courant.masculin := employe.masculin
courant.nbenfants := employe.nbenfants
```

Notez que, comme pour les tableaux, ces affectations globales ne seront possibles que pour des enregistrements déclarés explicitement du même type.

II - LA SYNTAXE DE LA DECLARATION D'UN TYPE ENREGISTREMENT

Au sein d'une déclaration type ou var, la définition d'un type enregistrement peut se faire comme suit :

```
record
     liste_d identificateurs 1 : description_de type 1
     liste_d_identificateurs_2 : description de type_2

     liste d_identificateurs_n : description_de_type n

end
```

Avec:

Le type enregistrement (sans variante)

liste d_identificateurs: un ou plusieurs identificateurs séparés par des virgules.

Notez que le mot end termine la description de l'enregistrement; celle-ci est comparable au contenu d'une instruction var avec ses listes d'identificateurs et ses descriptions de types; ces derniers pouvant être aussi bien des identificateurs de type que des descriptions effectives.

Jusqu'ici, nous avons vu qu'un même identificateur ne pouvait être défini qu'une seule fois. Le type enregistrement apporte en quelque sorte une exception (logique) à cette règle car il est possible de donner le même nom à des champs de deux types enregistrement différents; en effet, dans un tel cas, Pascal est en mesure de lever l'ambiguïté grâce au "préfixe" (nom d'enregistrement) précédant ce nom de champ. Néanmoins, pour des raisons de clarté des programmes, il est conseillé de ne pas abuser de cette possibilité (laquelle peut, au demeurant, entraîner quelques ambiguïtés en cas d'utilisation d'instructions with multiples .

Aucun restriction n'est apportée à la nature des différents champs qui peuvent être à leur tour, de type structuré, de sorte que l'on peut très bien définir des enregistrements comportant eux-mêmes des enregistrements ou des tableaux. Nous avions déjà fait une remarque analogue à propos des tableaux; maintenant, vous voyez que l'on peut aussi définir des tableaux d'enregistrements. Examinons dès maintenant un exemple d'enregistrements d'enregistrements. Nous rencontrerons ultérieurement un exemple de tableaux d'enregistrements.

III EXEMPLES D'ENREGISTREMENTS D'ENREGISTREMENTS

Supposons qu'à l'intérieur de nos enregistrements de type personne, nous ayons besoin d'introduire deux dates: la date d'embauche (date embauche) et la date d'entrée dans le dernier poste occupé (date poste); ces dates étant elles-mêmes formées de plusieurs champs:

jour_s (jour de la semaine)
jour (quantième du mois)
mois
annee.

Nous pourrions effectuer les déclarations suivantes:

```
type date = record
     jour_s : (lun, mar, mer, jeu, ven, sam, dim) ;
     jour : 1..31 ;
     mois : 1..12 ;
     annee : 0..99
end ;

type personne = record
     nom  : string [30]
     prenom   : string [20]
     masculin : boolean ;
     date_embauche : date ;
     date_poste : date ;
     nbenfants : 0..50
end ;

var employe, courant : personne ;
```

Voici quelques exemples de références à des champs de la variable employe:

`employe.date_embauche.annee` désigne l'année d'embauche de l'enregistrement employe (cette information est de type 0..99),

`employe.date_embauche` désigne la date d'embauche de l'enregistrement employe (il s'agit cette fois d'une information de type date)

L'instruction:

```
if employe.date embauche.jour_s = lun
```

permet de tester si un employé a été embauché un lundi.

IV - L'INSTRUCTION WITH

4.1 Exemples

La manipulation globale d'enregistrements n'est possible que dans le cas d'affectations (et à condition que l'on manipule tous les champs en même temps). Supposons que l'on souhaite écrire les informations de l'enregistrement employe. Nous pourrions procéder ainsi (en supposant que c est de type char):

```
writeln ('nom : ', employe.nom) ;
writeln ('prenom : ', employe.prenom) ;
if employe.masculin then c := 'M'
                                        else c := 'F' ;
writeln ('sexe : ', c, ' nbre enfants' , employe.nbenfants) ;
```

En fait, I'instruction with nous permet de simplifier les choses:

```
with employe do begin
     writeln ('nom : ', nom);
     writeln ('prenom : ', prenom) ;
     if masculin then c := 'M' else c := 'F';
      writeln ('sexe : '. c, ' nbre enfants' , nbenfants);
end;
```

Vous constatez que nous avons pu omettre le nom d'enregistrement à l'intérieur de:

```
with employe do begin

end
```

L'instruction with employe demande en fait au compilateur de "préfixer" par employe tous les idenficateurs qui correspondent à un nom de champ de l'enregistrement employe. Sa portée est naturellement limitée à l'instruction (souvent composée) mentionnée à la suite du do. Bien entendu, un identificateur ne correspondant pas à un nom de champ de l'enregistrement employe, n'est pas affecté par cette instruction.

4.2 Imbrication des instructions with

Avec les déclarations des types date et personne et la variable courant du paragraphe précédent, nous pourrions écrire ces instructions utilisant deux with imbriqués:

```
with courant do begin
     nom := ...
     prenom :=

with date_embauche do begin
     jour_s := ...
     jour := ...
      mois := ...
     annee := ...
     end ;

end ;
```

On obtiendrait le même résultat avec:

```
with courant do
     with date embauche do begin
               nom := ...
     prenom := ...
     jour_s := ...
     jour := ...
     mois := ...
     annee := ...
     ...
end ;
```

En effet, les noms de champ appartenant à date_embauche (et seulement ceux-là) seront préfixés par date_embauche. Puis, les champs appartenant à courant (y compris ceux éventuellement préfixés par date_embauche) seront préfixés par courant.

Il faut cependant noter que ce mécanisme d'imbrication des with risque d'entraîner quelques ambiguïtés lorsque des noms de champ identiques sont définis au sein d'enregistrements différents. En fait dans ce cas, Pascal, qui se trouve confronté à deux préfixes possibles pour un nom de champ, choisit celui qui a été cité en dernier (donc celui de niveau le plus imbriqué). Nous vous conseillons d'éviter ce genre de situation, bien compromettante pour la compréhension du programme.

Signalons enfin une troisième possibilité, équivalente aux deux précèdentes:

```
with courant, date embauche do begin
     nom := ...
     prenom := ...
     jour s := ...
     jour := ...
      mois := ...
     annee := ...
. . .
end;
```

V Exercices.

1) Ecrire un programme créant un annuaire de 10 personnes à trier par ordre alphabétique du nom.