

# Le langage Pascal

par Patrick Trau

Date de publication : Janvier 1992

Dernière mise à jour : 14 mai 2009

Ce document présente (de manière claire je crois, d'après ce qu'en ont dit certains) le langage **Pascal**. Il a été publié dans APTEP - INFO il y a quelques années. Depuis, bien que je maîtrisais bien ce langage et avais développé de nombreux programmes en Pascal, je suis passé au C. Le C est d'après moi plus souple et plus efficace à condition de bien en comprendre les mécanismes (il faut d'après moi comprendre à la fois l'Assembleur et les objets pour bien programmer en C). De plus, le C est plus utilisé dans mon environnement d'informaticiens. Néanmoins, le Pascal reste bien meilleur pour le débutant et le programmeur moyen : parfaitement structuré et clair, il conduit rapidement à un programme de bonne qualité et assez facilement maintenable (ce qui l'est moins en C, qui possède trop souvent trop de possibilités pour résoudre le même problème).

De plus, on trouve des **compilateurs Pascal** très conviviaux, avec aide en ligne, débogage, ...

Ce document reste donc d'actualité : le Pascal est d'après moi une très bonne manière d'aborder la programmation. C'est pourquoi j'ai choisi de mettre à disposition ce document sur Internet, qu'il serve à qui en a besoin !

Introduction.....	4
Les logiciels.....	4
Langages de programmation.....	4
Un premier petit programme.....	5
Constantes.....	7
Instruction d'affectation.....	8
Les types de variables standard simples et opérateurs associés.....	9
Entiers.....	9
Réels.....	9
Booléens.....	10
Caractères.....	10
Les fonctions standard.....	11
Instruction.....	12
Structures de contrôle.....	13
Boucle WHILE - DO (tant que - faire).....	13
Boucle REPEAT - UNTIL (répéter - jusqu'à ce que).....	13
Boucle FOR - DO (pour - faire).....	14
Instruction IF - THEN - ELSE (si - alors - sinon).....	14
La structure CASE - OF (cas - parmi).....	15
Types énumérés non standards.....	17
Les types intervalles.....	18
Tableaux.....	19
Tableaux unidimensionnels.....	19
Les chaînes de caractères.....	19
Tableaux de tableaux.....	20
Tableaux compactés.....	20
Enregistrements.....	22
Déclarations.....	22
Utilisation des enregistrements.....	22
La structure WITH - DO (avec - faire).....	23
Enregistrements avec variantes.....	23
Procédures et Fonctions.....	25
Généralités.....	25
Structure d'une entité de programme (routine).....	25
Portées des déclarations.....	26
Arguments(ou paramètres).....	26
Les fonctions.....	27
Récursivité.....	27
Les entrées / sorties.....	29
Sur la console.....	29
Sur fichier.....	29
Les fichiers texte.....	30
Extensions non standard.....	30
Accès direct.....	30
Imprimante.....	30
Autres.....	31
Ensembles.....	32
Pointeurs.....	35
Les listes chaînées et arbres.....	35
Les pointeurs en Pascal.....	35
Correction des exercices.....	37
Ex ex_puiss.....	37
Ex ex_jeu.....	37
Ex ex_moy.....	37
Ex ex_jeu_bis.....	38
Ex ex_calc.....	38
Ex moy_a.....	38
Ex rot_b.....	39

Ex clas_c.....	39
Ex ex_str.....	40
Ex mat.....	40
Ex tel.....	41
Ex rec.....	42
Ex fichier.....	43
Ex pointeurs.....	43

## Introduction

## Les logiciels

Dans la majorité des cas, on achète des programmes (logiciels) tout faits qui correspondent plus ou moins au besoin :

- Traitement de texte - P.A.O : avec mise en page, justification, numérotation chapitres-pages, table des matières, dictionnaire...
- Tableur : tableau de nombres à 2 dimensions et calculs
- Base de données : ensemble de fiches (nom, adresse...) et recherche par rubrique, publipostage...
- C.A.O, Dessin par ordinateur : propre, modification aisée, archivage...
- Gestion : paye, facturation, stock...
- Communication : transfert de programmes par modem et ligne téléphonique, serveur minitel...

Un Intégré regroupe plusieurs de ces possibilités.

Soit on achète un logiciel général : très bon niveau, parfaitement testé, documentation, formation... mais trop général (fonctions inutiles, fonctions utiles avec trop de paramètres ou difficilement accessibles); soit on fait (ou fait faire) un logiciel particulier : plus pratique, mais plus hasardeux (erreurs, SAV, doc...). Le cahier des charges doit être très précis.

## Langages de programmation

Un ordinateur est une machine bête, ne sachant qu'obéir, et à très peu de choses :

- Addition, soustraction, multiplication en binaire, uniquement sur des entiers;
- Sortir un résultat ou lire une valeur binaire (dans une mémoire par exemple);
- Comparer des nombres.

Sa puissance vient du fait qu'il peut être **programmé**, c'est à dire que l'on peut lui donner, à l'avance, la séquence (la suite ordonnée) des ordres à effectuer l'un après l'autre. Le grand avantage de l'ordinateur est sa rapidité. Par contre, c'est le programmeur qui doit **tout** faire. L'ordinateur ne comprenant que des ordres codés en binaire (le langage machine), des langages dits "évolués" ont été mis au point pour faciliter la programmation.

Le **Pascal**, créé par **Wirth** au début des années 70, possède des instructions assez claires (si vous comprenez l'anglais) et favorise une approche méthodique et disciplinée (on dit "structurée").

Le Pascal est un langage **compilé**, c'est-à-dire qu'il faut :

- Entrer un texte dans l'ordinateur (à l'aide d'un programme appelé **éditeur**);
- Le traduire en langage machine (c'est-à-dire en codes binaires compréhensibles par l'ordinateur) : c'est la **compilation** et éventuellement l'**édition de liens (Link)**;
- L'exécuter.

Contrairement à un Basic interprété, l'exécution sera beaucoup plus rapide puisqu'il n'y a plus de traduction à effectuer.

Bien que le langage soit normalisé, un certain nombre de points dépendent de la machine et du compilateur utilisé (par exemple, comment appeler le compilateur). Ces indications ne seront pas données ici. Si vous avez le choix, je vous conseille **Turbo Pascal**, le plus pratique d'emploi (en particulier parce qu'il possède son propre éditeur de texte). <sup>(1)</sup>

(1) Note de la rédaction : A l'heure actuelle, Turbo Pascal est devenu obsolète et peut avantageusement être remplacé par, par exemple, **Free Pascal** voire **Lazarus**.

## Un premier petit programme

Un programme Pascal est composé d'une **entête**, des **déclarations** et des **instructions** (délimitées par BEGIN et END).

```
PROGRAM cercle (input,output); (* entête *)
VAR
  perimetre, diametre : REAL; (* déclarations *)
BEGIN
  readln(diametre); (* instruction *)
  perimetre := 3.141592 * diametre; (* instruction *)
  writeln(diametre,perimetre) (* instruction *)
END.
```

L'**entête** est composée du mot PROGRAM, suivi du nom du programme (*cercle*), et d'indications sur les Entrées/Sorties (ici le clavier et l'écran).

La partie déclarative de notre programme est limitée à la déclaration de deux variables (mot-clef VAR). Une **variable** est une "case" mémoire de l'ordinateur, à laquelle on donne ici un nom. Chaque case peut contenir une valeur. On a précisé ici que nos deux variables *perimetre* et *diametre* contiendraient des réels.

Les types simples connus en Pascal sont :

- REAL;
- INTEGER (entier naturel);
- CHAR (contient un est un seul caractère);
- BOOLEAN (booléen, c'est-à-dire qui peut valoir soit TRUE (vrai) soit FALSE (faux)).

En Turbo Pascal, les **entiers** admissibles sont compris entre **-32768** et **+32767**. <sup>(2)</sup> Tous les compilateurs Pascal possèdent une constante prédéclarée MAXINT, qui donne le plus grand entier admissible.

Les **réels** doivent être compris en Turbo Pascal entre **+ et -1.7E37** (c'est-à-dire 1,7 fois 10 puissance 37), avec 11 chiffres significatifs. <sup>(3)</sup> La virgule décimale est toujours représentée par un point en informatique.

Un **identificateur** (tout nom que vous choisissez : variable, programme...) peut être formé de lettres (A à Z), de chiffres et (pas sur toutes les versions de Pascal) du signe \_ (souligné). Turbo Pascal accepte des noms de 127 caractères maximum, certains Pascal sont plus limités (31 caractères par ex). Le premier caractère doit être une lettre. Par exemple, *valeur1* ou *prem\_valeur* sont possibles mais pas *1ere\_valeur*. En Pascal, **les minuscules sont traitées comme des majuscules** (SURface et surFACE désignent la même case mémoire). Je n'utilise les majuscules que pour faire ressortir les mots importants. Les accents et autres ç ne sont pas autorisés (*var diamètre : real* est interdit à cause de l'accent). Un blanc dans un identificateur est également interdit (utilisez \_ pour séparer des mots dans un même identificateur).

Toute variable utilisée dans un programme doit être **déclarée**.

Les instructions de notre programme sont :

- Lecture sur le clavier : le programme s'arrête, attend que l'on donne une valeur à l'aide du clavier, met cette valeur dans la case *diametre* et continue lorsque l'on appuie sur la touche "**ENTREE**" ou "**RETURN**".
- Calcul et affectation : on multiplie le contenu de la case *diametre* par PI et on met le résultat dans la case *perimetre*. Le := symbolise une flèche à gauche. Ce n'est PAS une égalité au sens mathématique, mais la copie d'une valeur dans une mémoire.
- Ecriture sur l'écran : on affiche sur l'écran le contenu des cases *diametre* (que l'on connaissait puisque nous l'avions donné) et *perimetre* (qui nous intéresse un peu plus). Ce programme affiche donc deux chiffres.

<sup>(2)</sup> Les entiers signés 32 bits de Free Pascal sont compris entre -2 147 483 648 et 2 147 483 647; les entiers non signés entre 0 et 4 294 967 295.

<sup>(3)</sup> Sous Free Pascal, les réels sont compris respectivement :

- Single : 1.5E-45 et 3.4E38
- Double : 5.0E-324 et 1.7E308
- Extended : 1.9E-4932 et 1.1E4932

Selon la plateforme, le type Real sera soit un Single, soit un Double.

Il serait plus parlant d'afficher également des commentaires par l'instruction :

```
writeln('Diamètre : ',diametre,', Périimètre : ',perimetre);
```

Les textes doivent être entourés de cotes ('). Les majuscules/minuscules sont significatives. Pour afficher une apostrophe, utiliser deux cotes (*"exemple"*). Pour sauter une ligne, utiliser WRITELN seul.

Les instructions doivent toujours être séparées par des " ; " (j'ai dit "séparées", pas "terminées"). Le fait de passer à la ligne n'est interprété par l'ordinateur que comme un blanc. On aurait donc pu écrire notre programme sur une seule ligne (peut-être un peu longue pour l'éditeur).

Le programme doit toujours se terminer par un point.

## Constantes

Nous avons déjà utilisé des **variables** (on donne un nom à une mémoire). Mais on peut également utiliser des **constantes**, qui sont des valeurs qui restent fixes tout au long du programme et pour chaque exécution. On déclare les constantes avant de déclarer les variables, par :

```
CONST nom = valeur;
```

Exemple :

```
CONST taux_tva = 18.6;
```

MAXINT (plus grand entier possible) est une constante prédéfinie.  
On peut également avoir des constantes de type "chaîne de caractères" :

```
PROGRAM bonjour (output); { on n'a pas besoin ici du clavier }
CONST
  message1 = 'salut, ça va?';
  message2 = 'moi ça baigne';
BEGIN
  writeln(message1);
  writeln(message2) { notez l'absence de ; devant le END }
END.
```

Le type de la constante est déterminé automatiquement par le compilateur (entier si nombre sans point, réel si nombre avec point, caractères si entre cotes).

Remarque 1 : on peut toujours utiliser un entier à la place d'un réel mais pas l'inverse.

Remarque 2 : si on utilise un point décimal, il faut l'encadrer de chiffres (0.3 et non .3, 10 ou 10.0 et non 10.).

## Instruction d'affectation

On appelle **affectation** la mise d'une valeur dans une variable. Celle-ci peut être sous forme directe ( $A := B$ ) ou sous forme d'un calcul ( $A := B * C$ ). Le signe  $:=$  représente une flèche à gauche, et signifie "*mettre la **valeur** à droite du  $:=$  dans la mémoire désignée à gauche*" (mettre le contenu de B dans A ou mettre le résultat du calcul (contenu de B) fois (contenu de C) dans A). Une affectation du type  $B * C := A$  est donc **impossible**.

Une affectation ne peut se faire qu'entre une variable et une expression de même type (si A est réel, impossible de faire  $A := 'xyz'$ ). La seule exception est de mettre un entier dans un réel (le .0 est rajouté automatiquement), mais l'inverse est impossible directement.

Soient I entier et X réel, pour mettre X dans I il faut utiliser  $I := \text{Round}(X)$  (arrondi) ou  $I := \text{Trunc}(X)$  (partie entière).

On peut également utiliser des affectations et expressions booléennes. Par exemple, en ayant déclaré :

```
VAR
  test : boolean;
  a, b : real;
```

on peut écrire :

```
test := (a < b) and (a > 0);
```



## Les types de variables standard simples et opérateurs associés

Tous ces types, excepté les réels, sont dits **scalaires**. <sup>(4)</sup>

### Entiers

Déclaration :

```
VAR variable1, variable2, ..., variableN : INTEGER;
```

Opérations sur entiers :

- + (addition)
- - (soustraction)
- \* (multiplication)
- div (division)
- mod (reste de la division)

Elles sont toutes à résultat entier et nécessitent deux arguments entiers.

Les entiers sont compris entre  $-(\text{MAXINT} + 1)$  et  $+\text{MAXINT}$ , qui est une constante standard prédéfinie (sa valeur dépend par contre du compilateur, 32767 pour Turbo Pascal).

### Réels

Déclaration :

```
VAR variable1, variable2, ..., variableN : REAL;
```

Opérations :

- + (addition)
- - (soustraction)
- \* (multiplication)
- / (division)

Quand une opération comprend un argument réel et un entier, le résultat est réel. / donne toujours un résultat réel, même si les deux arguments sont entiers.

\* et / sont de priorité supérieure à + et -, mais entre \* et / tout dépend du compilateur (en général de gauche à droite). En cas d'ambiguïté, utilisez des parenthèses (il n'y a aucun inconvénient à mettre plus de parenthèses que nécessaire). Exemples d'expressions numériques (soit  $A = 3$ ,  $B = 4$ ,  $C = 2$ ) :

- $A + B / C = A + (B / C) = 5$
- $(A + B) / C = 3.5$
- $A / B * C = (A / B) * C$  (1.5) dans certains cas,  $A / (B * C)$  (0.375) dans d'autres
- $A / BC$  = valeur de A sur valeur de la variable de nom BC et non A sur  $B * C$
- $B * A - 5 * C = (B * A) - (5 * C) = 2$

<sup>(4)</sup> Note de la rédaction : Les types scalaires (*ordinal types* en anglais) sont une catégorie de types qui définissent des ensembles de valeurs ordonnées, à l'exception des types réels (qui forment une classe à part). On les appelle ainsi parce qu'on peut déterminer leur rang, qui est leur valeur entière.

## Booléens

Déclaration :

```
VAR variable1, variable2, ..., variableN : BOOLEAN;
```

Ces variables peuvent prendre soit la valeur TRUE (vrai), soit la valeur FALSE (faux).

Opérations booléennes :

- AND
- OR
- NOT
- XOR (ou exclusif)

Ces opérations nécessitent des arguments booléens.

Opérations à valeur booléenne :

- > (supérieur)
- < (inférieur)
- >= (supérieur ou égal)
- <= (inférieur ou égal)
- = (égal)
- <> (différent)

Ces opérations comparent tous éléments de type simple (les 2 arguments doivent être de même type, sauf entiers et réels qui peuvent être comparés entre eux), et renvoient un booléen. Les caractères sont comparés suivant l'ordre du code ASCII.

AND (et), OR (ou), NOT (non), sont de priorité supérieure aux précédents et ne peuvent opérer que sur des booléens : A > B et C doit être écrit : (A > B) and (A > C). Les parenthèses sont obligatoires pour ne pas faire en premier B and A.

## Caractères

Déclaration :

```
VAR variable1, variable2, ..., variableN : CHAR;
```

Ces variables contiennent **un** caractère. Ceux-ci sont classés suivant un ordre précis: le code ASCII, qui suit l'ordre suivant :

- Les chiffres '0' à '9' par ordre croissant;
- Les majuscules 'A' à 'Z' par ordre alphabétique;
- Les minuscules 'a' à 'z'.

Dans le code ASCII, chaque caractère possible a un numéro de code. Par exemple A a pour code 65. En déclarant C comme variable caractère, on peut mettre le signe A dans C par C := 'A' ou C := Chr(65). Dans le premier cas, il faut mettre les cotes pour différencier 'A' de la variable A. Pour mettre une cote dans C, on peut faire C := Chr(39) ou C := "" : la 1ère cote pour dire qu'il va y avoir un caractère, les 2 suivantes qui symbolisent la cote (car une seule cote voudrait dire fin du caractère), la dernière qui signifie fin du caractère.

## Les fonctions standard

On peut utiliser comme une variable des fonctions (qui peuvent soit être connues par le compilateur, soit définies par vous-même). Une fonction est un "module" ou "routine" qui renvoie une valeur au programme. Par exemple,  $A := \text{sqrt}(B * C)$  met dans A la racine carrée de B fois C.  $B * C$  est appelé **argument** de la fonction.

Les principales fonctions standard connues par tous les compilateurs sont :

- ABS : renvoie la valeur absolue
- SQR : renvoie le carré
- SQRT : racine carrée
- EX : exponentielle
- LN : log népérien
- SIN : sinus
- COS : cosinus
- ARCTAN : arc tangente
- SUCC : variable énumérée suivante
- PRED : précédent
- ROUND : arrondi à l'entier le plus proche
- TRUNC : partie entière (permet de mettre un réel dans un entier :  $\text{trunc}(4.5) = 4$ )

Comme toute variable, une fonction possède un type (entier, réel,...) défini, et ne peut donc être utilisée que comme une variable de ce type.

## Instruction

On appelle **instruction simple** soit :

- Une affectation;
- Un appel à une procédure (une procédure est un ensemble d'instructions regroupées sous un nom, par exemple READLN);
- Une structure de contrôle (voir plus bas).

On appelle **instruction composée** le regroupement de plusieurs instructions sous la forme :

```
BEGIN instruction1; instruction2; ... ; instructionN END
```

On ne met pas de ; après BEGIN ni avant END (puisque le ; sépare deux instructions). Par contre, si l'instruction composée est *suivie* d'une autre instruction, on mettra un ; après le END.

Remarque : La lisibilité du programme sera meilleure en mettant une instruction par ligne, et en décalant à droite les instructions comprises entre un BEGIN et un END :

```
BEGIN
  instruction1;
  instruction2;

  instructionN
END
```

On appelle **instruction** une instruction soit simple soit composée.

## Structures de contrôle

Nos connaissances actuelles ne nous permettent pas de faire des programmes utilisant la capacité de l'ordinateur de répéter rapidement et sans erreur beaucoup de calculs. Nous allons donc remédier immédiatement à cela. Chaque structure de contrôle forme une instruction (qui peut donc être utilisée dans une autre structure de contrôle).

### Boucle WHILE - DO (tant que - faire)

Structure :

```
WHILE expression booléenne DO instruction
```

Elle permet de répéter l'instruction tant que l'expression (ou la variable) booléenne est vraie.

```
PROGRAM racine_a_deux_decimales (input, output);
VAR
  nombre, racine : REAL;
BEGIN
  writeln('Entrez un réel entre 0 et 10');
  readln(nombre);
  racine := 0;
  WHILE racine * racine < nombre DO
    racine := racine + 0.01;
  writeln('La racine de ', nombre, ' vaut à peu près', racine)
END.
```

Il faut noter que si l'expression booléenne est fausse dès le début, l'instruction n'est jamais exécutée (ici si nombre = 0). Attention, Pascal n'initialise pas automatiquement les variables à 0, c'est-à-dire que sans l'instruction `racine := 0`, le programme risquerait de donner une réponse fausse (racine valant n'importe quoi, il sera en général très supérieur à la racine cherchée).

On ne peut répéter qu'une **seule** instruction. Mais celle-ci peut être simple (comme dans l'exemple précédent) ou composée (begin - end).

#### **Exercice ex\_puiss :**

*Faire un programme qui affiche les puissances de 2 jusqu'à une valeur maxi donnée par l'utilisateur (par multiplication successive par 2).*

**Voir la correction**

### Boucle REPEAT - UNTIL (répéter - jusqu'à ce que)

Structure :

```
REPEAT
  instruction1;
  instruction2;
  ...etc...
  instructionN
UNTIL condition
```

Les N instructions sont répétées jusqu'à ce que la condition soit vérifiée. Même si la condition est vraie dès le début, elles sont au moins exécutées une fois.

```
PROGRAM jeu_simpliste (input,output);
VAR
```

```

a : integer;
BEGIN
writeln('Entrez le nombre 482');
REPEAT
  readln(a)
UNTIL a = 482;
writeln('C''est gentil de m''avoir obéi')
END.

```

Quelle que soit la valeur initiale de A (même 482), la question sera au moins posée une fois (plus si vous désobéissez).

### **Exercice ex\_jeu :**

Faire un jeu qui demande de trouver le nombre entre 0 et 10 choisi par l'ordinateur (en comptant les coups).

On utilisera la fonction Random(N) (non standard, disponible en Turbo Pascal) qui renvoie un entier entre 0 et N-1 compris, par l'instruction valeur\_choisie := Random(11).

**Voir la correction**

## Boucle FOR - DO (pour - faire)

Structure :

```

FOR variable := valeur_début TO valeur_fin DO instruction

```

La variable\_énumérée (non réelle) prend la valeur\_début, et l'instruction est exécutée. Puis elle est incrémentée (on passe à la suivante, c'est-à-dire que, si elle est entière, on ajoute 1), et ce jusqu'à valeur\_fin (comprise).

L'instruction sera donc exécutée (valeur\_fin - valeur\_début + 1) fois. Si valeur\_fin est inférieure à valeur\_début, l'instruction n'est jamais exécutée. Cette forme de boucle est utilisée chaque fois que l'on connaît le nombre de boucles à effectuer.

On peut utiliser un **pas dégressif** en remplaçant TO par DOWNTO :

```

for lettre := 'Z' downto 'A' do
  writeln(lettre)

```

Dans cet exemple, on écrit l'alphabet à l'envers (en déclarant lettre du type CHAR).

La variable peut être utilisée (mais pas modifiée) dans l'instruction (simple ou composée). Elle est souvent appelée **indice** de la boucle. Sa valeur est perdue dès que l'on sort de la boucle.

### **Exercice ex\_moy :**

Faire un programme qui calcule la moyenne de N nombres. N doit être demandé par un READLN.

(initialiser une variable à 0, y ajouter progressivement chaque note puis diviser par N).

**Voir la correction**

## Instruction IF - THEN - ELSE (si - alors - sinon)

Structure :

```

IF condition THEN instruction1 (* CAS 1 *)
{ ou }
IF condition THEN instruction1 ELSE
  instruction2 (* CAS 2 *)

```

Si la condition est vraie, alors on exécute l'instruction1 (simple ou composée). Sinon, on passe à la suite (cas 1), ou on exécute l'instruction2 (cas 2).

Remarquez qu'il n'y a pas de ; devant le ELSE.

### Exercice ex\_jeu\_bis :

Modifier le jeu précédent (ex\_jeu) en aidant le joueur (en précisant si c'est plus ou c'est moins).

[Voir la correction](#)

L'instruction2 peut être composée ou entre autres être une instruction IF :

```
IF condition1 THEN
  instruction1
ELSE IF condition2 THEN
  instruction2
ELSE IF condition3 THEN
  instruction3
.....
ELSE instructionN
```

Un ELSE correspond toujours au dernier IF rencontré (mais dont on n'a pas encore utilisé le ELSE).

```
IF cond1 THEN
  IF cond2 THEN
    inst1 { cond1 et cond2 }
  ELSE inst2 { cond1 et pas cond2 }
ELSE IF cond3 THEN
  inst3 { pas cond1 mais cond3 }
ELSE inst4 { ni cond1 ni cond3 }
```

Si on désire autre chose, utiliser BEGIN et END :

```
IF cond1 THEN
BEGIN
  if cond2 then
    inst1
END { le prochain ELSE se rapporte à COND1 puisque l'instruction (composée) suivant THEN est terminée }

ELSE inst2
```

## La structure CASE - OF (cas - parmi)

Elle évite d'utiliser une trop grande suite de ELSE IF.

Structure :

```
CASE expression OF { regardez bien où j'ai mis les ; }
  liste_de_cas1 : instruction1;
  liste_de_cas2 : instruction2;
  .....
  liste_de_casN : instructionN
END
```

L'instruction *i* sera exécutée si l'expression appartient à la *liste\_de\_cas i*. Les autres ne seront pas exécutées (on passe directement au END). L'expression doit être de type scalaire (pas de réels).

En Turbo Pascal, on accepte une *liste\_de\_cas* particulière qui est ELSE (et doit être placée en dernier), pour prévoir le cas où expression n'appartient à aucun des cas cités au dessus. En MS-Pascal on utilise de même OTHERWISE.

```
CASE a * b OF { avec a et b déclarés entiers }
```

```
0 : writeln('un des nombres est nul');  
1, 10, 100, 1000, 10000 :  
writeln('le produit est une puissance de 10'); { 100000 est impossible en Turbo Pascal car supérieur à MAXINT }  
  
END
```

Attention, certains compilateurs n'acceptent pas de passer sur un CASE avec une valeur prévue dans aucune liste de cas.

**Exercice ex\_calc :**

*Faire un programme simulant une calculatrice à 4 opérations en utilisant CASE pour le choix de l'opération à effectuer.*

***Voir la correction***



## Types énumérés non standards

Si les types prédéfinis mentionnés précédemment ne vous suffisent pas, vous pouvez déclarer (donc créer) d'autres types. Il faut d'abord définir le type que vous désirez créer en donnant la liste ordonnée de toutes les valeurs possibles.  
Exemple :

```
TYPE tjour = (lundi,mardi,mercredi,jeudi,vendredi,samedi,dimanche);
```

Toutes les variables de ce type ne pourront pas avoir d'autres valeurs que celles que l'on a énumérées.  
Il faut ensuite déclarer quelles variables seront de ce type.

Exemple :

```
VAR jour1, jour2 : tjour;
```

On peut alors les utiliser dans le programme grâce à :

- **Des affectations** :

```
JOUR1 := mercredi;  
JOUR2 := JOUR1;
```

- **Des fonctions prédéfinies** :

PRED (précédent), SUCC (suivant), ORD (numéro d'ordre (dans la déclaration), entre 0 et N-1).

Exemples :

```
pred(mardi) = lundi  
succ(mardi) = mercredi  
ord(mardi) = 1
```

- **Des comparaisons** :


lundi < mercredi mais attention : dimanche > lundi (suivant l'ordre donné dans la déclaration).

- **Des boucles** :

```
FOR jour1 := lundi TO vendredi DO ...
```

- **Des sélections de cas** :

```
CASE jour2 OF  
  lundi : ...;  
  samedi,dimanche : ...  
END
```

 *READLN et WRITELN ne fonctionnent pas en standard pour les types définis par l'utilisateur, mais plusieurs compilateurs l'acceptent (mais pas Turbo Pascal).*

## Les types intervalles

On peut également définir un type énuméré comme sous-ensemble d'un autre type énuméré, en en donnant les bornes. En voici quelques exemples :

```
TYPE
jourtravail = lundi..vendredi;
    {si on a déjà défini tjour }
mois = 1..12;
    {sous-ensemble du type INTEGER}
byte = 0..255;
    {prédéfini en Turbo Pascal, prend 1 octet en mémoire}
minuscules = 'a'..'z';
```

Ces variables s'utilisent comme des variables du type dont est issu leur intervalle (on peut utiliser une variable mois comme tout autre entier) mais peuvent utiliser moins de mémoire.

## Tableaux

### Tableaux unidimensionnels

On a souvent besoin de regrouper dans une seule variable (et donc un seul nom) plusieurs variables (exemple 3 coordonnées d'un même vecteur). On utilise pour cela les **tableaux**. La manière la plus simple des les définir est :

```
VAR nom_tableau : ARRAY [type_index] OF type_composantes

{Exemple}
CONST dimension = 3;
VAR vecteur1, vecteur2 : ARRAY [1..dimension] OF REAL;
```

On peut utiliser le tableau complet (vecteur1 := vecteur2) pour l'affectation uniquement (vecteur1 + vecteur2 est impossible directement). Mais on peut également accéder aux différentes composantes par un index (si vecteur1 est le vecteur unitaire porté par Y, alors vecteur1[1] = 0, vecteur2[2] = 1, vecteur3[3] = 0).

L'index peut être variable:

```
PROGRAM heures (input,output);
TYPE
  tj = (lundi,mardi,mercredi,jeudi,vendredi,samedi,dimanche);
VAR
  jour : tj;
  nb_heures_cours : ARRAY[tj] OF integer;
BEGIN {il faudrait ici entrer les nombres d'heures de cours par jour}
  FOR jour := lundi TO samedi DO
    writeln(nb_heures_cours[jour])
  END.
```

On peut également définir un **type** tableau, par exemple :

```
TYPE
  typheures = ARRAY[tj] OF integer;
VAR
  nb_heures_cours : typheures;
```

#### **Exercice moy\_a :**

*Ecrire un programme qui lit une liste de N nombres, calcule et affiche la moyenne puis l'écart entre chaque note et cette moyenne.*

**Voir la correction**

#### **Exercice rot\_b :**

*Ecrire un programme qui lit une liste de N nombres, la décale d'un cran vers le haut (le 1er doit se retrouver en dernier), l'affiche puis la décale vers le bas.*

**Voir la correction**

#### **Exercice clas\_c :**

*Classer automatiquement une liste de n entiers par ordre croissant puis décroissant.*

**Voir la correction**

## Les chaînes de caractères

On appelle **chaîne de caractères** une suite de caractères regroupés dans une seule variable. En Pascal standard, il faut la déclarer comme ARRAY [1..nbcar] OF CHAR. Les tests sont possibles entre chaînes (test suivant l'ordre ASCII du 1er caractère, puis automatiquement du 2ème, etc).

Exemples : 'aaz' < 'aba' 'aa' < 'aaa' 'ZZZ' < 'aaa' (majuscules)

On peut également utiliser ces tableaux dans READLN et WRITELN.

Mais en général, les compilateurs possèdent des types plus puissants, associés à des fonctions et procédures non standard. En Turbo Pascal, on déclare une chaîne par String [nbcar]. Une fonction utile est alors Length(variable\_string), qui donne la longueur de la chaîne (inférieure ou égale à la dimension déclarée).

### Exercice ex\_str :

*Ecrire un programme qui détermine le nombre et les positions d'une chaîne dans une autre (par exemple : 'ON' dans 'FONCTION' : en position 2 et 7).*

**Voir la correction**

## Tableaux de tableaux

On peut faire des tableaux avec des composantes de tous types, y compris de tableaux. On peut voir une matrice comme un tableau de lignes qui sont elles-mêmes des tableaux de composantes.

Exemple : matrice 5 lignes, 10 colonnes :

#### TYPE

```
ligne = array [1..10] of real;
mat = array [1..5] of ligne;
```

Les écritures TYPE mat = array [1..5] of array [1..10] of real

ou même TYPE mat = array [1..5,1..10] of real

sont strictement équivalentes (sauf si la définition du type ligne est nécessaire pour d'autres variables).

Soit la déclaration

#### VAR

```
m1, m2 : mat;
i, j : integer;
```

on peut écrire :

```
m1[i,j] := 10
    {affectation de la valeur 10 en ligne i,colonne j}
m1[i] := m1[i + 1]
    {copie complète de la ligne i+1 sur la ligne i}
m2 := m1
    {copie de la matrice complète }
```

mais il est impossible d'accéder directement à une colonne.



**Rappel :** READLN et WRITELN ne s'appliquent qu'à des variables simples prédéfinies (c'est-à-dire les composantes, si celles-ci sont réelles, entières, chaînes ou booléennes).

### Exercice mat :

*Faire le calcul de multiplication d'une matrice (M lignes, L col) par (L lignes, N col) (résultat M lignes, N col).*

**Voir la correction**

## Tableaux compactés

On peut, en Pascal standard, définir des tableaux en PACKED ARRAY. Ils s'utilisent de la même manière que des tableaux normaux, mais prennent moins de place en mémoire. Sur certains ordinateurs, ceci se paie par un

ralentissement du programme dans certaines applications. En Turbo Pascal ce n'est pas le cas, et elles sont donc automatiquement toutes compactées (inutile de le préciser). <sup>(5)</sup>

```
VAR truc : PACKED ARRAY [1..10] OF typtruc;
```

On possède en standard les procédures de transfert :

```
PACK(table_non_packée,index_début,table_packée_resultat);  
UNPACK(table_packée,table_non_packée_resultat,index_début);
```

pour transférer toute la table, mettre index\_début à 1.

---

(5) Note de la rédaction : Si elle est en effet ignorée par Turbo Pascal, la directive PACKED peut être utilisée avec les compilateurs Free Pascal, Virtual Pascal ou Delphi pour compacter les types structurés sans tenir compte de l'alignement par défaut.

## Enregistrements

Dans un tableau, tous les constituants doivent être semblables. Ce n'est pas le cas des enregistrements, qui sont des variables composées de plusieurs variables (ou **champs**) de types différents.

## Déclarations

Structure d'une déclaration d'enregistrement :

```
VAR
  nom_variable : RECORD
    champ1 : type;
    champ2 : type;
    ...
  END;

{ ou }

TYPE
  nom_type = RECORD
    champ1 : type;
    champ2 : type;
    ...
  END;
```

Exemple :

```
TYPE
  date = RECORD
    jour : 1..31;
    mois : 1..12;
    an : 1980..1999
  END;
  facture = RECORD
    reference : integer;
    jour : date; {enregistrement d'enregistrement}
    client : string [100];
    total_HT : real
  END;

VAR
  date1, date2 : date;
  comptes : array [1..100] of facture;
  fact : facture;
```

Dans un enregistrement, chaque champ doit avoir un nom différent. Mais pour des enregistrements différents on peut réutiliser le même nom de champ (ici jour).

## Utilisation des enregistrements

Comme l'affectation fonctionne toujours pour des variables de même type, on peut écrire DATE1 := DATE2. On accède à un champ par NOM\_VARIABLE.NOM\_CHAMP :

```
writeln('Référence de la facture ? ');
readln(fact.reference);
writeln('Mois de la facture ? ');
readln(fact.jour.mois);
writeln('Jour de la facture ? ');
readln(fact.jour.jour);
```

```
for i := 1 to nb_fact do
  writeln(comptes[i].prix)
```

Un champ d'enregistrement s'utilise comme une variable du même type (avec les mêmes possibilités mais aussi les mêmes limitations).

## La structure WITH - DO (avec - faire)

Structure :

```
WITH nom_enregistrement DO instruction
```

Elle permet d'omettre le nom\_enregistrement dans l'instruction.

```
FOR I := 1 TO nb_fact DO
  WITH comptes[i] DO
    begin
      reference := i; { c.a.d comptes[i].reference }
      writeln('Nom du client de la facture ndeg.',i,' ?');
      readln(client);
    end
```

Dans un WITH, on peut évidemment accéder à d'autres enregistrements, il suffit de préciser le nom\_enregistrement.champ. Si l'on utilise des WITH imbriqués, en cas d'ambiguïté des noms de champs on se réfère au WITH le plus proche (intérieur).

```
WITH fact DO
  begin
    WITH date1 DO
      begin
        mois := 12;      { date1.mois }
        reference := 1;   { fact.reference car date1.reference n'existe pas }
        jour := 15;      { date1.jour car WITH DATE1 le plus à l'intérieur }
        fact.jour := date2 { fact.jour puisqu'on le précise }
      end
    end
```

## Enregistrements avec variantes

Certains champs d'un enregistrement peuvent être différents suivant la valeur d'un des champs en utilisant la structure CASE OF dans la déclaration de l'enregistrement.

```
TYPE
  statut = (celibataire,marie,divorce,veuf);
  perso = RECORD
    nom : string[20];
    CASE situation : statut OF
      celibataire : ();
      marie : (enfants : 0..10);
      divorce, veuf : (enfants : 0..10; remarie : boolean)
    END;
```

Il ne peut y avoir qu'un CASE par enregistrement. Celui-ci doit se placer en dernier, après les champs communs (si il y en a). Mais dans ce CASE on peut avoir un autre CASE imbriqué (et ainsi de suite).

Les cas où il n'y a pas de champ variant doivent être précisés (ici celibataire). Le champ discriminant (situation) s'utilise comme un autre champ commun.

R Exemples d'instructions possibles (x déclaré comme perso) :

```
x.situation := marie;  
if x.situation = veuf then  
  writeln(x.enfants)
```

**Exercice tel :**

*A l'aide d'un tableau de personnes (nom, prénom, numéro dans la rue, rue, département, ville, numéro de téléphone), faire un programme de recherche automatique de toutes les informations sur les personnes répondant à une valeur d'une rubrique donnée (tous les PATRICK, tous ceux de Saverne, etc...).*

***Voir la correction***



## Procédures et Fonctions

Nous avons déjà vu un certain nombre de **procédures** (WRITELN) et **fonctions** (SQRT, SIN...) prédéfinies par le compilateur. Mais si l'on en désire d'autres, il suffit de les définir.

### Généralités

On peut regrouper un ensemble d'instructions sous un même nom. On forme alors un **sous-programme** ou procédure. On utilise les procédures :

- Chaque fois qu'une même suite d'instructions doit être répétée plusieurs fois dans un programme;
- Quand une suite d'instructions forme une action globale. Le programme est alors plus clair et les erreurs plus facilement détectables.

Pour pouvoir utiliser une procédure, il faut d'abord la **déclarer**. La déclaration des procédures et fonctions se fait après toutes les autres déclarations.

### Structure d'une entité de programme (routine)

- Entête
- Déclaration des :
  - Labels; (pour les GOTO, déconseillé)
  - Constantes;
  - Types;
  - Variables.
- Définition des sous-routines; (sous-programmes)
- BEGIN  
... Instructions ...  
END.

Le programme principal comme les procédures et les fonctions ont toujours cette structure. L'entête est composée d'un mot clef (PROGRAM, PROCEDURE ou FUNCTION), suivi de l'identificateur (nom) de la routine, et de la liste des arguments entre parenthèses. Les arguments forment la liaison avec l'extérieur de la routine (clavier, écran et éventuellement fichiers pour le programme).

```
PROGRAM remplir (output);  
  {entête du prog principal}  
  
VAR i : integer;  
  {déclarations prog princ.}  
  
  {dont déclaration LIGNE}  
PROCEDURE ligne(n:integer);  
  {entête de la procédure}  
var j : integer;  
  {déclarations procédure}  
BEGIN  
  {corps de la procédure}  
  for j := 1 to n do write('*');  
  writeln  
END;  
  
BEGIN  
  {instructions du prog princ}  
  for i := 1 to 25 do ligne(70)  
END.
```

La procédure LIGNE écrit N caractères "\*" sur une même ligne. Le programme remplit donc l'écran (25 lignes 70 colonnes) d'étoiles.

On peut appeler une procédure déclarée dans une routine n'importe où dans cette routine en indiquant simplement son nom comme si c'était une instruction. A l'appel d'une procédure, le programme interrompt son déroulement normal, exécute les instructions de la procédure, puis retourne au programme appelant et exécute l'instruction suivante. Tout ce passe donc comme si le nom de la procédure était remplacé dans le programme par les instructions de la procédure (avec n = 70).

## Portées des déclarations

Celle-ci est symbolisée dans l'exemple ci-dessus par deux cadres : la variable I et la procédure LIGNE (avec un argument entier) sont déclarées dans REMPLIR, et donc connues dans tout le programme (rectangle extérieur). Par contre N et J sont déclarés dans LIGNE et ne sont connus (et utilisables) que dans le rectangle intérieur.

En d'autres termes :

- Une variable est **locale** pour une procédure X si elle est déclarée dans X. Elle n'existe que dans X (et dans les procédures déclarées à l'intérieur de X). La routine qui comporte la procédure X ne peut donc pas accéder à cette variable locale.
- Une variable est **globale** pour une procédure X si elle est déclarée dans une routine **englobant** la procédure X. Elle peut être utilisée dans la procédure. La modifier la modifie également dans la routine appelante (englobante).

Si l'on avait déclaré une variable I dans la procédure LIGNE (au lieu de N ou J), celle-ci aurait été locale à la procédure, c'est à dire que, dans le programme principal, I désigne une autre case mémoire que dans la procédure. Modifier la variable *locale* I ne modifie pas la variable *globale* I (momentanément inaccessible).

**Remarque** : Ceci s'applique à toutes les déclarations. En particulier, une procédure déclarée localement à l'intérieur d'une procédure est indéfinie à l'extérieur.

## Arguments(ou paramètres)

Les échanges d'informations entre une routine et une sous-routine peuvent se faire par l'intermédiaire des variables globales. Mais il est beaucoup plus intéressant d'utiliser les **paramètres** : (6)

```
PROGRAM machin (input,output);

VAR
  a, b, c, d : real;

PROCEDURE aff_somme(x,y:real);
var z : real;
begin
  z := x + y;
  writeln(x, ' + ', y, ' = ', z)
end;

BEGIN { programme principal }
  writeln('Entrez 4 valeurs :');
  readln(a,b,c,d);
  aff_somme(a,b);
  aff_somme(3,5);
  aff_somme(c+a,d)
END.
```

En appelant AFF\_SOMME(A,B), la procédure prend pour x la valeur de a, et pour y la valeur de b. On dit que les arguments sont **passés par valeur**. Mais si la procédure modifiait x ou y, a et b ne seraient pas modifiés dans le

(6) **Note de la rédaction** : La meilleure manière de penser une sous-routine est de la considérer comme un tout autonome et donc de lui fournir tout ce dont elle a besoin par l'intermédiaire de paramètres.

programme appelant. Pour répercuter les modifications des arguments, il faut les déclarer comme **variables** (ils sont alors dits **passés par adresse**).

```
PROCEDURE echange (VAR x, y : real);
var z : real;
begin
  z := x;
  x := y;
  y := z
end; {cette procédure échange les contenus des 2 arguments}
```

## Les fonctions

Tout ce qui a été dit pour les procédures s'applique également aux fonctions. La différence avec une procédure est qu'une fonction **renvoie un résultat**.

L'entête est du type :

```
FUNCTION nom_fonction (liste_parametres) : type_de_la_fonction
```

La liste des paramètres (en général passés par valeur) est de la même forme que pour une procédure, le type de la fonction étant le type du résultat retourné. On retourne le résultat par :

```
NOM_FONCTION := ...
```

Exemple :

```
PROGRAM classer (input,output);

VAR a, b, c : real;

FUNCTION max (x, y : real) : real;
begin
  if x >= y then
    max := x
  else max := y
end;

BEGIN
  writeln('Entrez deux valeurs : ');
  readln(a,b);
  c := max(a,b);
  writeln('Le plus grand est ',c)
END.
```

La fonction max a deux paramètres réels (x et y) et renvoie un réel.

## Récurtivité

C'est ainsi que l'on appelle le fait qu'une routine puisse s'appeler elle-même.

```
function factorielle (n : integer) : integer;
begin
  if n <= 1 then
    factorielle := 1
  else
    factorielle := n * factorielle(n - 1)
```

```
end;
```

Par exemple, en appelant factorielle(3), on calcule  $3 * \text{factorielle}(2)$ . Or,  $\text{factorielle}(2) = 2 * \text{factorielle}(1)$ , qui lui vaut 1. Donc  $\text{factorielle}(3) = 3 * (2 * 1)$  (ce qui me paraît juste). Faites un petit dessin, à chaque appel on recrée de nouvelles variables locales, donc on obtient 3 cases n distinctes valant 3, 2 et 1, on les supprime petit à petit en passant sur le END.

Remarque : Il faut toujours vérifier qu'en aucun cas on ne puisse avoir une boucle infinie qui bloquerait la machine. Ce serait le cas en mettant le test IF N = 1 et en appelant factorielle pour une valeur négative ou nulle.

Une procédure devant toujours être déclarée pour pouvoir être utilisée, on utilise FORWARD pour les cas de récursivité passant par deux procédures :

```
function prem (a, b : real) : boolean; FORWARD; { déclaration anticipée de l'entête }

procedure deux (x, y : real);
var vbool : boolean;
begin
    .....
    vbool := prem(x,y); { on peut utiliser PREM car déjà déclarée }
    .....
end;

function prem; { ne plus donner les arguments car déjà déclarés }
begin
    .....
    if pas_fini then
        deux(a,b); { DEUX déjà déclarée }
    .....
end;
```

### **Exercice rec :**

*Ecrire le programme qui calcule le déterminant d'une matrice carrée NxN sachant que celui-ci vaut :*

*où  $M[i, 1]$  est l'élément de la matrice (ligne i, 1ère colonne),*

*DETn-1 est le déterminant de la sous-matrice d'ordre n-1 obtenu en ôtant la ligne i et la 1ère colonne.*

*Le déterminant d'une matrice 1x1 est son seul élément.*

*On utilisera bien évidemment une fonction récursive, et l'on séparera le calcul de sous-matrice dans une procédure.*

Remarque : Il existe des méthodes numériques permettant d'accéder au résultat beaucoup plus rapidement que par cette méthode.

**Voir la correction**

## Les entrées / sorties

### Sur la console

La procédure `WRITELN` permet d'afficher des résultats sur l'écran. Les différents arguments sont affichés les uns après les autres sur la même ligne. Le curseur est ensuite automatiquement mis en début de ligne suivante. Pour éviter ceci, on peut utiliser `WRITE`, qui s'utilise comme `WRITELN` mais le curseur est laissé derrière le dernier caractère. De même, `READ` fonctionne comme `READLN`, excepté le curseur qui reste sur la ligne. <sup>(7)</sup>  
`WRITE` et `WRITELN` acceptent le "formatage" : on peut imposer le nombre de caractères utilisés pour chaque variable. Soit `I` entier, `R` réel : `WRITE(I:5, ' ', R:7:2)` écrira `I` sur 5 caractères, et `R` sur 7 caractères dont 2 après la virgule. Si les valeurs sont trop petites pour entrer dans le format, des blancs sont mis devant. Si elles sont trop grandes, le format est ignoré.

### Sur fichier

Un fichier est un ensemble de données, écrites sur un "support" lisible par l'ordinateur (disquette, cartes perforées,...), et regroupées sous un nom. Un fichier peut contenir des caractères (fichier texte), des programmes, des valeurs (fichier de données).

Etudions l'exemple suivant :

```
PROGRAM recopier (input,output);
VAR
  fic_ent, fic_sor : file of real;
  x : real;
BEGIN
  assign(fic_ent,'fichier1'); { non standard }
  reset(fic_ent);
  assign(fic_sor,'fichier2'); { non standard }
  rewrite(fic_sor);
  while not eof(fic_ent) do
    begin
      read(fic_ent,x);
      write(fic_sor,x)
    end;
  close(fic_ent);
  close(fic_sor)
END.
```

On déclare les fichiers par :

```
VAR nomfic : FILE OF type_du_contenu
```

Un fichier peut contenir des éléments de n'importe quel type (aussi compliqué soit-il, même tableaux), mais tous les enregistrements du fichier doivent être du même type.

Avant d'utiliser le fichier, il faut relier son identificateur au nom effectif du fichier par `ASSIGN`. Ici, `fic_ent` correspondra à un fichier qui aura pour nom `FICHIER1` sur la disquette. `ASSIGN` n'est pas une fonction standard, d'autres compilateurs utilisent `OPEN`.

Puis il faut préciser si l'on va lire ou écrire sur le fichier, par `RESET` ou `REWRITE`. Ceci positionne en début du fichier. On écrit dans le fichier par

```
WRITE(nomfic,liste_des_variables);
```

Idem pour `READ`.

<sup>(7)</sup> Note de la rédaction : `READ` peut poser des problèmes lors de lectures successives au clavier car elle n'interprète pas le "Enter".

Il ne faut pas oublier de fermer le fichier (CLOSE) quand on n'en a plus besoin. CLOSE est nécessaire également si l'on veut refaire un nouvel ASSIGN sur le même fichier.

La fonction standard EOF(nomfic) est true quand on arrive en fin de fichier (*End Of File*).

Remarque : Lors de la définition initiale du Pascal, on devait ouvrir les fichiers avant d'appeler le programme. Les fichiers étaient des variables globales déclarées en arguments du programme (et donc passés du système d'exploitation au programme). Nous les déclarons maintenant **dans** le programme, les deux seuls fichiers donnés en paramètres du programme restant input et output. <sup>(8)</sup>

## Les fichiers texte

Le principal problème des fichiers précédents est que tous les enregistrements sont du même type. De plus, ils ne sont pas directement imprimables ni visibles par un traitement de texte. On y remédie par les **fichiers texte** (aussi appelés *fichiers formatés*), déclarés par :

```
VAR nomfic : TEXT;
```

Ce sont des fichiers de caractères, et se comportent comme l'écran ou le clavier - qui sont d'ailleurs les fichiers texte input et output utilisés automatiquement si on ne précise pas de nomfic dans les READ(In) et WRITE(In). On a aussi les mêmes limitations (écriture d'entiers, réels et chaînes de caractères seulement).

```
PROGRAM lire (input,output,fic);
TYPE
  chaine = array [1..80] of char;
VAR
  fic : text;
  ligne : chaine;
BEGIN
  assign(fic,'texte');
  rewrite(fic);
  writeln('Tapez votre texte, il sera enregistré dans le fichier TEXTE ');
  writeln('Tapez FIN pour arrêter');
  repeat
    readln(ligne);
    writeln(fic,ligne)
  until ligne = 'FIN';
  close(fic)
END.
```

## Extensions non standard

### Accès direct

La plupart des compilateurs acceptent la procédure SEEK(nomfic,position), qui permet de se positionner n'importe où dans le fichier (sans être obligé de lire dans l'ordre tous les éléments). Ceci n'est évidemment possible que si la taille des éléments est constante, donc pour tous les fichiers exceptés ceux du type TEXT, puisque les lignes n'y sont pas de longueur constante. L'accès à une information est donc beaucoup plus rapide (à condition de connaître sa position). On appelle ceci l'**accès direct** à un fichier, par opposition à l'**accès séquentiel** prévu en standard.

SEEK est d'autant plus utile que RESET autorise la lecture et l'écriture des fichiers à accès direct (REWRITE, par contre, efface tout le contenu du fichier, on ne peut donc plus qu'écrire).

### Imprimante

Pour accéder à l'imprimante, deux solutions :

---

<sup>(8)</sup> Note de la rédaction : Cette obligation a maintenant disparu.

- Ouvrir un fichier texte sur disque, le remplir, et lorsque le programme est terminé le copier (par le DOS) sur imprimante;
- Utiliser un nom de fichier prédéfini correspondant à l'imprimante. En Turbo Pascal, c'est LST. Par exemple : `writeln(lst,'COUCOU')`. LST n'a pas besoin d'être ouvert en Turbo Pascal (tout comme output).

## Autres

Turbo Pascal permet beaucoup d'autres opérations sur les fichiers (Delete, Rename, Path...) qui ne sont pas standard. Si vous les utilisez, rappelez-vous qu'elles ne fonctionneront pas avec un autre compilateur (Microsoft par exemple) ni sous un autre système d'exploitation (Unix par exemple).

### **Exercice fichier :**

*Ecrire la procédure qui lit le fichier "annuaire" afin de rendre opérationnel l'**exercice tel**.*

*Modifier ce programme pour permettre l'entrée et la modification du fichier annuaire.*

***Voir la correction***

## Ensembles

Un ensemble est une "collection" d'éléments de même type (cf cours maths 6ème). Supposons vouloir représenter des vendeurs et leurs domaines d'action.

Le type ensemble est défini par SET OF :

```
TYPE produits = (velos,motos,autos,accessoires);
VAR vendeur1, vendeur2 : SET OF produits;
```

On "remplit" un ensemble en donnant ses éléments entre crochets :

```
vendeur1 := [velos,motos];
vendeur2 := [motos,accessoires];
```

L'ensemble vide est : [].

On peut faire les opérations suivantes :

- **Union** : vendeur1 + vendeur2 = [velos,motos,accessoires]
- **Intersection** : vendeur1 \* vendeur2 = [motos]
- **Complément** : vendeur1 - vendeur2 = [velos] vendeur2 - vendeur1 = [accessoires]

Les tests booléens possibles sont : = , <> , <= (inclus) , >= (contenant).

On teste l'appartenance d'un élément par IN : si X vaut motos, alors X IN VENDEUR1 et [motos] <= VENDEUR1 sont équivalents (IN compare un élément et un ensemble, alors que <= compare deux ensembles).

**Remarque** : En général, on ne pense pas à utiliser les ensembles (le prof de maths aurait-il oublié de nous dire à quoi ça sert ?), et l'on s'embrouille dans des programmes complexes. Voici par exemple, des idées pour programmer facilement un automatisme défini par plusieurs Grafcet complexes ([xx] se lisant "ensemble des xx") :

```
[étapes actives] := [étapes initiales]
```

Pour chaque transition :

```
si [étapes immédiatement précédentes] <= [étapes actives]
et [capteurs nécessaires] <= [capteurs]
alors [étapes actives] := [étapes actives] - [précédentes] + [suivantes]
activer [sorties] en fonction [étapes actives] et boucler
```

On peut trouver un exemple détaillé dans mon document sur la **mise en oeuvre du Grafcet**.

```
{ Ce programme correspond au GRAFCET 2 "mise en oeuvre du grafcet sur automate" }

PROGRAM grafcet_2 (input,output);

CONST
  adresse_port = $330;

TYPE
  liste_capteurs = (e1,e2,e3);
  ensemble_capteurs = SET OF liste_capteurs;
  liste_actions = (sortie1,sortie2,sortie3);
  ensemble_actions = SET OF liste_actions;

VAR { pour le programme principal }
  etape : array [1..4] of boolean;
  transition : array [1..4] of boolean;
  capteurs : ensemble_capteurs;
```



```

sorties : ensemble_actions;
i : integer;

PROCEDURE lire_capteurs (VAR etat_actuel_capteurs : ensemble_capteurs);
{ cette procédure lit les capteurs et rend un ensemble contenant les
  capteurs à 1. Cette procédure dépend du type de machine }
var { locale } etat : record
    case integer of
        1 : (compatible_port : byte);
        2 : (compatible_ensemble : ensemble_capteurs)
    end;
begin
    etat.compatible_port := port[adresse_port];
    etat_actuel_capteurs := etat.compatible_ensemble
end;

PROCEDURE affecte_sorties (etat_sorties : ensemble_actions);
{ affecte les sorties }
var etat : record
    case integer of
        1 : (compatible_port:byte);
        2 : (compatible_ensemble:ensemble_actions)
    end;
begin
    etat.compatible_ensemble := etat_sorties;
    port[adresse_port] := etat.compatible_port
end;

BEGIN { programme principal }
{initialisation}
sorties := []; { ensemble vide }
affecte_sorties(sorties);
etape[1] := true;
for i := 2 to 4 do etape[i] := false;
repeat
    lecture des entrées}
    lire_capteurs(capteurs);
    {-----}
    write('capteurs : ');
    if e1 in capteurs then write('E1');
    if e2 in capteurs then write('E2 ');
    if e3 in capteurs then write('E3 ');
    writeln;
    {-----}
    { conditions d'évolution }
    transition[1] := etape[1] and (e1 in capteurs);
    transition[2] := etape[2] and etape[3] and ([e2,e3] * capteurs = []); { intersection vide }
    transition[3] := etape[3] and (e2 in capteurs) and not (e3 in capteurs);
    transition[4] := etape[4] and not (e2 in capteurs);
    { désativation }
    if transition[1] then etape[1] := false;
    if transition[2] then begin
        etape[2] := false;
        etape[3] := false
    end;
    if transition[3] then etape[3] := false;
    if transition[4] then etape[4] := false;
    { activation }
    if transition[1] then begin
        etape[2] := true;
        etape[3] := true
    end;
    if transition[2] then etape[1] := true;
    if transition[3] then etape[4] := true;
    if transition[4] then etape[3] := true;
    { affectation sorties }
    {-----}
    write('Etapes : ');
    for i := 1 to 4 do if etape[i] then write(i, ' ');
    writeln;
    {-----}

```

```
sorties := [];  
if etape[2] then sorties := sorties + [sortie1];  
if etape[3] then sorties := sorties + [sortie2];  
if etape[4] then sorties := sorties + [sortie3];  
affecte_sorties(sorties);  
UNTIL false; { boucler jusqu'à extinction }  
END.
```

## Pointeurs

Un pointeur est une variable qui stocke **l'adresse** (c'est-à-dire la position en mémoire) d'une variable.

## Les listes chaînées et arbres

Supposons une liste de 3 entiers. On suppose connaître pour chacun l'adresse du suivant :

Si l'on veut insérer une valeur dans la liste, les modifications à apporter sont minimales :

Contrairement à l'insertion dans un tableau, il est inutile de décaler les termes suivants. Pour connaître la liste, il suffit de connaître l'adresse du premier terme.

Pour représenter un **arbre**, il suffit pour chaque élément de connaître l'adresse de chaque fils :

Remarque : si le nombre de fils n'est pas constant, on a intérêt à stocker uniquement le fils aîné, ainsi que le frère suivant.

## Les pointeurs en Pascal

En pascal, on utilise les **pointeurs** pour représenter ces objets. La déclaration se fait de la manière suivante :

```
TYPE pointeur = ^type_de_la_variable_pointée;
```

Exemple :

```
TYPE
tpoint = ^tval; { pointe sur des TVAL }
tval = record
    valeur : integer;
    suivant : tpoint
end;

VAR
    p1, p2 : tpoint;
```

Dans cet exemple, les variables de type tval contiendront un entier et l'adresse de la suivante (liste chaînée vue plus haut).

Contrairement aux tableaux, il n'est pas nécessaire de prévoir le nombre de variables pointées à l'avance. C'est l'**allocation dynamique de mémoire** : on réserve la place pour chaque variable en cours d'exécution du programme, par la commande NEW(nom\_variable). On récupère la place, si la variable est devenue inutile, par DISPOSE(nom\_variable).

p1 contient donc une adresse d'une variable de type tval. Cette variable sera p1^ (c'est-à-dire pointée par p1). On la "remplit" donc par des affectations du type :

```
p1^.valeur := 15;
p1^.suivant := p2;
```

Examinons un programme qui lit puis affiche une liste chaînée d'entiers :

```
PROGRAM liste (input,output);
```

```
TYPE
  tpoint = ^tval;
  tval = record
    valeur : integer;
    suivant : tpoint
  end;

VAR
  prem, precedent, point : tpoint;
  i, n : integer;
BEGIN
  write('Combien d\'éléments comporte votre liste ?');
  readln(n);
  new(prem);
  { le 1er est particulier : si on le perd, on perd la liste }
  write('1ère valeur ? ');
  readln(prem^.valeur);
  { lecture de l'enregistrement VALEUR de la variable d'adresse (pointée par) PREM }
  precedent := prem;
  for i := 2 to n do begin
    new(point);
    { création d'une nouvelle variable }
    write(i, 'ième valeur ? ');
    readln(point^.valeur);
    precedent^.suivant := point;
    { mise à jour du chainage }
    precedent := point
    { se préparer pour la prochaine boucle }
  end;
  precedent^.suivant := NIL;
  { NIL signifie "rien" car 0 est un entier, non une adresse }
  point := prem;
  { heureusement, on se souvient du premier }
  for i := 1 to n do begin
    writeln(point^.valeur);
    point := point^.suivant
    { pour la prochaine boucle }
  end
END.
```

**Exercice pointeurs :**

Modifier le programme ci-dessus pour qu'il permette de rajouter ou supprimer des éléments. Le décomposer en routines.

**Voir la correction**

## Correction des exercices

### Ex ex\_puiss

```
PROGRAM puissances (input, output);
VAR
  n, max : integer;
BEGIN
  writeln('Nombre maxi ? ');
  readln(max);
  n := 2;
  while n <= max do begin
    writeln(n);
    n := n * 2
  end;
  writeln('C'est fini')
END.
```

### Ex ex\_jeu

```
PROGRAM jeu (input, output);
VAR
  choix, rep, nb : integer;
BEGIN
  nb := 0;
  choix := random(11);
  repeat
    nb := nb + 1;
    writeln('Choix ndeg. ', nb, ' ? ');
    readln(rep)
  until rep = choix;
  writeln('Trouvé en ', nb, ' coups')
END.
```

### Ex ex\_moy

```
PROGRAM moyenne (input, output);
VAR
  n, i : integer;
  note, total, moyenne : real;
BEGIN
  writeln('Nombre notes à entrer ?');
  readln(n);
  total := 0;
  for i := 1 to n do begin
    writeln(i, 'ième note ? ');
    readln(note);
    total := total + note
  end;
  moyenne := total / n;
  writeln('La moyenne est : ', moyenne)
END.
```

## Ex ex\_jeu\_bis

```
PROGRAM jeu_ameliore (input, output);
VAR
  choix, rep, nb : integer;
BEGIN
  nb := 0;
  choix := random(11);
  repeat
    nb := nb + 1;
    writeln('Choix ndeg. ',nb,' ? ');
    readln(rep);
    if rep < choix then
      writeln('c'est plus')
    else if rep > choix then
      writeln('c'est moins')
    { le 2ème if empêche d'écrire si juste }
  until rep = choix;
  writeln('Juste en ',nb,' coups')
END.
```

## Ex ex\_calc

```
PROGRAM calculatrice (input, output);
VAR
  val1, val2, resultat : real;
  operation : char;
BEGIN
  writeln('Première valeur ?');
  readln(val1);
  writeln('Opération (+ - * /) ? ');
  readln(operation);
  writeln('Deuxième valeur ? ');
  readln(val2);
  case operation of
    '+' : resultat := val1 + val2;
    '-' : resultat := val1 - val2;
    '*' : resultat := val1 * val2;
    '/' : resultat := val1 / val2;
  end;
  writeln('Résultat : ',resultat)
END.
```

## Ex moy\_a

```
PROGRAM moyenne (input,output);
VAR
  n, compteur : integer;
  somme, moyenne, ecart : real;
  note : array [1..100] of real;
BEGIN
  repeat
    writeln('nb notes (100 maxi)?');
    readln(n)
  until (n > 0) and (n <= 100);
  {entrée notes et calcul de la somme}
  somme := 0;
  for compteur := 1 to n do
    begin
      writeln(compteur,'è note ?');
```

```
    readln(note[compteur]);
    somme := somme + note[compteur]
  end;
  {calcul et affichage de la moyenne}
  moyenne := somme / n;
  writeln('Moyenne : ',moyenne);
  {calcul et affichage des écarts}
  writeln('Ecart :');
  for compteur := 1 to n do
    begin
      ecart := note[compteur] - moyenne;
      writeln(compteur, 'ième note (' ,note[compteur],') : écart : ',ecart)
    end
  end
END.
```

## Ex rot\_b

```
PROGRAM rotation (input,output);
VAR
  index, n : integer;
  prem : real;
  tableau : array [1..100] of real;
BEGIN
  repeat
    writeln('Nb valeurs (100 maxi) ?');
    readln(n)
  until (n > 0) and (n <= 100);
  { entrée des valeurs }
  for index := 1 to n do
    begin
      writeln(index, 'ième valeur ?');
      readln(tableau[index]);
    end;
  writeln('On décale vers le haut');
  prem := tableau[1]; { ne pas écraser ! }
  for index := 2 to n do
    tableau[index - 1] := tableau[index];
  tableau[n] := prem;
  for index := 1 to n do
    writeln(tableau[index]);
  writeln('on re-décale vers le bas');
  prem := tableau[n];
  for index := n downto 2 do
    tableau[index] := tableau[index - 1];
  tableau[1] := prem;
  for index := 1 to n do
    writeln(tableau[index])
  end
END.
```

## Ex clas\_c

```
PROGRAM classer (input,output);
VAR
  n, i, index, petit, indexpetit : integer;
  avant, apres : array [1..100] of integer;
  pris : array [1..100] of boolean;
  { pour noter ceux déjà pris }
BEGIN
  repeat
    writeln('Nb valeurs (100 maxi) ?');
    readln(n)
  until (n > 0) and (n <= 100);
  { entrée valeurs - initialisation de pris }
  for index := 1 to n do
    begin
```

```

    writeln(index, 'ième valeur ? ');
    readln(avant[index]);
    pris[index] := false
  end;
  { ordre croissant, on cherche N valeurs }
  for i := 1 to n do
    begin
      petit := maxint; { plus grand possible }
      { recherche du plus petit non pris }
      for index := 1 to n do
        if (not pris[index]) and (avant[index] <= petit) then
          begin
            petit := avant[index];
            indexpetit := index
          end;
      { sauvegarde dans le tableau APRES et mise à jour de PRIS }
      apres[i] := petit;
      pris[indexpetit] := true
    end; { passage au prochain i }
  { affichage du tableau APRES }
  writeln('Par ordre croissant : ');
  for i := 1 to N do writeln(apres[i]);
  { classement par ordre décroissant }
  writeln('Par ordre décroissant : ');
  for i := n downto 1 do writeln(apres[i])
  { n'auriez-vous pas tout refait ? }
END.

```

## Ex ex\_str

```

PROGRAM position (input,output);
VAR
  ch, sch : string [255];
  i, j, n, l, ls : integer;
BEGIN
  writeln('Chaîne à tester ? ');
  readln(ch);
  writeln('Sous-chaîne à trouver ?');
  readln(sch);
  l := length(ch);
  ls := length(sch);
  n := 0;
  for i := 1 to l - ls do
    begin
      j := 1;
      while (j <= l) and (ch[i + j - 1] = sch[j]) do
        j := j + 1;
      if j > ls then
        begin
          writeln('Trouvé position ', i);
          n := n + 1
        end
      end;
    end;
  writeln(n, ' fois ', sch, ' dans ', ch)
END.

```

## Ex mat

```

PROGRAM produit_mat (input,output);
VAR
  m1, m2, m3 : array [1..10,1..10] of real;
  l, m, n, j1, jm, jn : integer;
BEGIN
  writeln('Nb lignes 1ère matrice ?');
  readln(m);

```



```
writeln('Nb colonnes 1ère matrice ?');
readln(l);
writeln('Nb colonnes 2ème matrice ?');
readln(n);
{ entrée de m1 }
writeln('Première matrice');
for jm := 1 to m do
  for jl := 1 to l do
    begin
      writeln('lig',jm,', col',jl,'?');
      readln(m1[jm,jl])
    end;
{ entrée de m2 }
writeln('2ème matrice');
for jl := 1 to l do
  for jn := 1 to n do
    begin
      writeln('lig',jl,', col',jn,'?');
      readln(m2[jl,jn])
    end;
{ calcul du produit }
for jm := 1 to m do
  for jn := 1 to n do
    begin {calcul composante m,n de m2}
      m3[jm,jn] := 0;
      for jl := 1 to l do
        m3[jm,jn] := m3[jm,jn] + (m1[jm,jl] * m2[jl,jn]);
      end;
{ affichage du résultat }
writeln('Résultat');
for jm := 1 to m do
  for jn := 1 to n do
    writeln('m[' ,jm, ', ',jn, ']=',m3[jm,jn])
END.
```

## Ex tel

```
PROGRAM annuaire (input,output);
{ version simplifiée }
TYPE
  ligne = string [40];
  typepersonne = record
    nom : ligne;
    num_tel : ligne
    { integer malheureusement < 32635 }
  end;
VAR
  pers : array [1..100] of typepersonne;
  nb, i : 1..100;
  rep : char;
  imprimer : boolean;
  texte : ligne;
BEGIN
  { on suppose avoir ici les instructions permettant de lire sur fichier disque NB et le tableau PERS }
  repeat
    writeln('Recherche suivant : ');
    writeln(' N : nom');
    writeln(' T : numéro téléphone');
    writeln(' Q : quitter le prog');
    writeln('Quel est votre choix ?');
    readln(rep);
    if rep <> 'Q' then begin
      writeln('Texte à chercher ? ');
      readln(texte)
      for i := 1 to nb do with pers[i] do
        begin
          case rep of
            'N' : imprimer := nom = texte;
```

```

    'T' : imprimer := num_tel = texte;
  end;
  if imprimer then begin
    writeln('Nom : ', nom);
    writeln('Tel : ', num_tel)
  end
end
end
until rep = 'Q'
END.

```

## Ex rec

```

PROGRAM determ (input,output);
{ on se limite à 10x10, ce qui fait 7h de calcul et 6.235.314 appels à DETN }

TYPE
  tmat = array [1..10,1..10] of real;

VAR
  dim : integer; { dimension matrice à calculer }
  det : real;    { résultat désiré }
  mat : tmat;    { matrice à calculer }
  appel : real;  { nb d'appels }

procedure entree;
var lig, col : integer;
begin
  writeln('Dimension de la matrice ?');
  readln(dim); { DIM variable globale }
  writeln('Entrez les composantes :');
  for lig := 1 to dim do begin
    writeln('pour la ligne ndeg. ', lig);
    for col := 1 to dim do begin
      writeln('colonne ', col, ' ?');
      readln(mat[lig,col])
    end
  end
end;

procedure sous_mat (mdeb : tmat; var mfin : tmat; ind, dim : integer);
{ on supprime la colonne 1 et la ligne ind pour avoir la s/mat de dim-1 }
var col, lig, l : integer;
begin
  l := 0;
  for lig := 1 to dim do begin
    if lig <> ind then begin
      l := l + 1;
      for col := 2 to dim do
        mfin[l,col - 1] := mdeb[lig,col]
      end
    end
  end
end;

function detn (m : tmat; d : integer) : real;
{ dét ordre d en fonction ordre d-1 }
var result : real;
    mprim : tmat; { matrice intermédiaire }
    lig, signe : integer;
begin
  appel := appel + 1;
  if d = 1 then detn := m[1,1]
  { fin de récursivité }
  else begin
    result := 0;
    signe := -1;
    for lig := 1 to d do begin
      sous_mat(m, mprim, lig, d);
    end
  end
end;

```

```

    signe := -signe;
    { changer de signe à chaque ligne }
    result := result + (signe * m[lig,1] * detn(mprim,d - 1))
  end;
  detn := result
end
end;

BEGIN { programme principal }
  entree;
  appel := 0;
  det := detn(mat,dim);
  writeln('résultat : ',det);
  writeln('nb appels DETN : ',appel)
END.

```

## Ex fichier

```

procedure lirefic;
var i : 1..100;
    f : file of typepersonne;
{ variables globales : NB et le tableau PERS }
begin
  assign(f,'annuaire'); {non standard}
  reset(f);
  nb := 0;
  while not EOF(f) do begin
    nb := nb+1;
    read(f,pers[nb])
  end;
  close(f)
end;
{à vous de faire la suite}

```

## Ex pointeurs

```

PROGRAM liste(input,output);

TYPE
  tpoint = ^tval;
  tval = record
    valeur : integer;
    suivant : tpoint
  end;

VAR
  prem : tpoint; { variable globale }
  n:integer;
  c:char;

procedure lire;
{ modifie N et PREM }
var precedent, point : tpoint;
    i : integer;
begin
  write('Combien d''éléments?');
  readln(n);
  new(prem);
  write('1ère valeur ? ');
  readln(prem^.valeur);
  precedent := prem;
  for i := 2 to n do begin
    new(point);
    write(i,'ième valeur ? ');
    readln(point^.valeur);
    precedent^.suivant := point;
    precedent := point
  end;
end;

```

```
end;
precedent^.suivant := NIL
{ le dernier ne pointe sur rien }
end;

procedure afficher;
var point : tpoint;
    i : integer;
begin
    point := prem;
    for i := 1 to n do begin
        writeln(point^.valeur);
        point := point^.suivant
    end
end;

procedure supprimer;
var point, prec : tpoint;
    rep : char;
begin
    point := prem;
    repeat
        write(point^.valeur, ' à ôter ?');
        readln(rep);
        if rep = 'O' then begin
            n := n-1;
            if point <> prem then begin
                prec^.suivant := point^.suivant;
                dispose(point);
                point := prec^.suivant
                { se préparer pour la suite }
            end
            else begin
                prem := point^.suivant;
                dispose(point);
                { ancien premier }
                point := prem
            end
        end
        else begin
            { pointer sur le suivant }
            prec := point;
            point := point^.suivant
        end
    until point = nil
end;

procedure rajouter;
var p1, p2, prec : tpoint;
    rep : char;
begin
    p1 := prem;
    repeat
        write(p1^.valeur, 'Rajouter un élément avant (O/N) ? ');
        readln(rep);
        if rep = 'O' then begin
            n := n + 1;
            if p1 <> prem then begin
                new(p2);
                write('Valeur ? ');
                readln(p2^.valeur);
                prec^.suivant := p2;
                p2^.suivant := p1;
                prec := p2;
            end
            else begin
                new(p1);
                write('Valeur ? ');
                readln(p1^.valeur);
                p1^.suivant := prem;
                prem := p1
            end
        end
    until rep = 'N'
end;
```

```
        end
    end
else begin
    { pointer sur le suivant }
    prec := p1;
    p1 := p1^.suivant
end
until p1=nil;
p1 := prec;
repeat
    write('Ajouter un élément en fin de liste (O/N) ? ');
    readln(rep);
    if rep = 'O' then begin
        n := n + 1;
        new(p2);
        write('Valeur ? ');
        readln(p2^.valeur);
        p1^.suivant := p2;
        p2^.suivant := nil;
        p1 := p2
    end
until rep<>'O'
end;

BEGIN { programme principal }
lire;
repeat
    writeln('A:afficher, S:supprimer R:rajouter, F:fin');
    write('Votre choix ? ');
    readln(c);
    case c of
        'A' : afficher;
        'S' : supprimer;
        'R' : rajouter
    end
until c = 'F'
END.
```