Cours de Turbo Pascal 7.0

*Le cours aux 100 exemples**

Hugo ETIEVANT

http://cyberzoide.developpez.com

* : 100 programmes téléchargeables : exemples du cours, annales corrigés, programmes originaux…

Sommaire

Retrouvez aussi 5 **tests d'évaluation** en ligne ainsi que des **annales corrigées** et des **programmes** en libre téléchargement sur :
http://cyberzoide.developpez.com/info/turbo/

Préface

Le langage Pascal offre une très bonne approche de la programmation. Très utilisé dans le milieu scolaire, il permet d'acquérir des notions solides que l'on retrouve dans tous les autres langages. Le CyberZoïde est l'un des très rares site web à proposer un véritable cours de programmation en Pascal avec de très nombreux exemples et programmes annotés en libre téléchargement.

Les éléments de base de la programmation tels que : pointeurs, types, tableaux, procédures, fonctions, graphismes... et bien d'autres vous sont expliqués avec le maximum de pertinence, de simplicité et d'efficacité, puisque vous êtes déjà très nombreux (étudiants comme professeurs d'Université) à vous fier à ce cours.

De plus vous disposez également de plusieurs tests d'évaluation qui vous permettent d'évaluer vos connaissances en Pascal. Enfin, les travaux pratiques de filière 3 de l'Université Claude Bernard (Lyon 1 (69), FRANCE) sont régulièrement corrigés et mis en téléchargement sur ce site.

Introduction

Cette aide électronique sur la programmation en Turbo Pascal 7.0 est destinée en premier lieu aux non-initiés, à tous ceux qui débutent dans la programmation. Que ce soit dans le cadre de l'enseignement à l'Université ou pour votre propre intérêt personnel, vous avez décidé d'apprendre ce langage fort archaïque mais qui a néanmoins le mérite de former à la logique informatique. Le langage Pascal est très structuré et constitue en lui-même une très bonne approche de la programmation.

Vous découvrirez dans les pages qui vont suivre, les bases de la programmation en général : les structures de boucle et de contrôle, l'utilisation de la logique booléenne, la chronologie d'exécution du code... Ces notions de base vous servirons si vous décidez de changer de langage de programmation, car les principes de base (et même les instructions de base) sont les mêmes.

Dans la vie courante, nous n'avons pas pour habitude de nous limiter au strict minimum lorsqu'on communique, ici, ce principe est bafoué, puisque d'une langue vivante complexe vous allez passer à un langage strict, rigide et pauvre. Issue des mathématiques, cette langue exacte est par essence optimisée et simplifiée. Par delà, l'apprentissage d'un langage informatique forme à la systémique mathématico-informatique, vous apprendrez à dominer le comportement de la machine et à être plus clair et précis dans votre manière de construire vos idées.

Utiliser le compilateur Borland Pascal 7.0

Pour ouvrir un fichier, aller dans le menu **File/Open...** ou taper la touche fonction **F3**. Pour exécuter un programme, aller dans le menu **Run/Run** ou taper la combinaison de touches **Ctrl+F9**.

Pour compiler "correctement" un exécutable, aller dans le menu **Compile/Make** (ou **/Compile**) ou taper **F9** on obtient ainsi des exécutables de meilleurs qualité qui pourront être utilisés sur d'autres ordinateurs.

Si vous avez ommis de mettre une pause à la fin d'un programme, ou si vous désirez tout simplement avoir sous les yeux, la dernière page d'écran, il vous suffit d'alllez dans le menu : **Debug/User Screen** ou tapez **ALT+F5**.

Pour une aide, aller dans le menu **Help/Index** ou taper **Shift+F1**. Pour obtenir de l'aide sur une instruction qui apparait dans un script, placez le curseur de la souris dessus et allez dans le menu **Help/Topic Search**, une fenêtre apparaîtra alors.

Si un problème a lieu lors de l'exécution d'un programme, utilisez le débugger :
Debug/Watch. Une fenêtre apparait en bas de page. Cliquez sur **Add** et tapez le nom de la variable dont vous désirez connaître la dernière valeur.

Erreur 200 : Division par zéro

Nombreux sont ceux d'entre vous qui ont eut un grave pépin avec le compilateur Turbo Pascal. En effet, l'excécution d'un programme utilisant l'unité Crt provoque un bug chez les ordinateurs récents du type Pentium III. L'erreur observée est la suivante : `Error 200 : division by zero`.

Mais d'où vient cette erreur ?

Les nouveaux microprocesseurs sont devenus incompatibles avec les opérations de bas niveau écrites dans l'unité Crt (fichier **CRT.TPU**). En effet, les instructions de cette unité traitent l'heure système dont le codage sur le microprocesseur a changé dans les récents modèles d'ordinateurs.

Comment y remédier ?

Pour pouvoir utiliser de nouveau l'unité Crt dans vos programmes, il vous faut soit changer quelques fichiers propres au compilateur soit appliquer un patch à chacun de vos programmes compilés avant de pouvoir les exécuter normalement. Notez que la compilation du programme ne provoque aucune erreur, c'est seulement son exécution qui provoque cette erreur de division par zéro.

Où se procurer un patch ?

Sur le site web de Borland (éditeur du compilateur Pascal le plus répandu), ou sur beaucoup d'autres sites que vous trouverez en effectuant une courte recherche dans un moteur. Par exemple sur AltaVista.com, faites la recherche "Crt+patch" et télécharger les patchs proposés sur les sites trouvés par le moteur de recherche.

Télécharger les patchs :

Ces patchs étant freeware, le CyberZoïde est autorisé à vous les proposer en téléchargement. Vous avez deux types de patch à votre disposition, n'en utilisez qu'un seul.
Le premier patch : patch1.zip (http://cyberzoide.developpez.com/info/turbo/patch1.zip) contient un fichier à copier dans le répertoire **/BIN** de votre compilateur puis à compiler. Les prochains programmes que vous compilerez n'auront alors plus aucun problème et s'exécuteront normalement.
Le second patch : patch2.zip (http://cyberzoide.developpez.com/info/turbo/patch2.zip) contient toutes les explications techniques détaillées (en anglais) sur l'unité Crt ainsi qu'un programme à exécuter en lui envoyant en paramètre votre programme compilé. Ce dernier sera modifié et marchera très bien. L'inconvénient de ce patch, c'est qu'il faut l'exécuter sur chaque programme que vous fabriquez, c'est chiant mais j'ai pas réussi à faire marché l'autre !
Attention : ces patchs sont de leur auteurs respectifs, l'utilisation que vous en ferez est à vos risques et périls.

Voici quelques liens intéressants sur le sujet :

http://support.intel.com/support/processors/pentiumII/run200.htm
http://www.inprise.com/devsupport/pascal/
http://www.pro-desk.com/inside/special/error200.htm

Généralités

Architecture standard d'un listing en pascal

```
{ les instructions facultatives pour compilation doivent êtres entre accolades }
Program nom de programme ;
Uses unités utilisées ;
Const déclaration de constantes ;
Type déclaration de types ;
Function déclaration de fonction ;
Procedure déclaration de procédure paramétrée ;
Var déclaration de variables ;
Procedure déclaration de procédure simple ;
    BEGIN { programme principal }
    ...
    Commandes
    ...
    END.
```

Grammaire du Pascal

- Un nom de programme respecte les règles liées aux identificateurs (cf plus bas) et ne peut pas contenir le caractère point "."
- Un programme principal débute toujours par `BEGIN` et se termine par `END.` (avec un point). Alors qu'un sous-programme (ou fonction, procédure, bloc conditionnel...) commence lui aussi par `Begin` mais se termine par `End ;` (sans point mais avec un point-virgule).
- Chaque commande doit se terminer avec un point-virgule. Il n'y a pas d'exception à la règle hormis `Begin` et l'instruction précédent `End` ou `Else`.
- Il est toléré de mettre plusieurs instructions les unes à la suite des autres sur une même ligne du fichier mais il est recommandé de n'en écrire qu'une par ligne : c'est plus clair et en cas de bogue, on s'y retrouve plus aisément. De plus, s'il vous arrive d'écrire une ligne trop longue, le compilateur vous le signifiera en l'erreur `Error 11: Line too long`. Il vous faudra alors effectuer des retours à la ligne comme le montre l'exemple suivant :

```
WriteLn('Fichier: ', file,
    ' Date de création:', datecrea,
    ' Utilisateur courant:', nom,
    ' Numéro de code:', Round(ArcTan(x_enter)*y_old):0:10) ;
```

- Les noms de constantes, variables, procédures, fonctions, tableaux, etc. (appellés identificateurs) doivent êtres des noms simples, par exemple, n'appelez pas une variable comme ça : **x4v_t3la78yugh456b2dfgt** mais plutôt comme cela : **discriminant** (pour un programme sur les éq du 2nd degré) ou **i** (pour une variable de boucle).
- Les identificateurs doivent impérativement être différents de ceux d'unité utilisées, de mots réservés du langage Pascal et ne doivent pas exéder 127 signes (1 lettre au minimum). Ils ne doivent être composés que de lettres, de chiffres et du caractère de soulignement (Shift+8).

- Les identificateurs ne doivent pas contenir de caractères accentués, ni d'espace, ni de point et ni les caractères suivants : @, $, &, #, +, -, *, /. Mais le caractère de soulignement est autorisé. De plus, Turbo Pascal ne différencie aucunement les majuscules des minuscules. Les chiffres sont acceptés ormis en première place.
- N'hésitez pas à insérer des commentaires dans votre code, cela vous permettra de comprendre vos programme un an après les avoir écrit, et ainsi d'autres personnes n'auront aucun mal à réutiliser vos procédures, fonctions... Procédez ainsi :
  { *ici votre commentaire entre accolades* }
  (* *ici vos commentaires entre parenthèses et étoiles* *)
  Vos commentaires peuvent tenir sur une seule ligne comme sur plusieurs. Vous pouvez aussi mettre en commentaire une partie de votre programme.
- Un identificateur ne peut être égale à un mot réservé du langage pascal !

Mots réservés du langage Pascal

```
AND, ARRAY, ASM, BEGIN, CASE, CONST, CONSTRUCTOR, DESTRUCTOR,
DIV, DO, DOWNTO, ELSE, END, EXPORTS, FILE, FOR, FUNCTION,
GOTO, IF, IMPLEMENTATION, IN, INHERITED, INLINE, INTERFACE,
LABEL, LIBRARY, MOD, NIL, NOT, OBJECT, OF, OR, PACKED,
PROCEDURE, PROGRAM, RECORD, REPEAT, SET, SHL, SHR, STRING,
THEN, TO, TYPE, UNIT, UNTIL, USES, VAR, WHILE, WITH, XOR.
```

CHAPITRE I : Entrées et sorties à l'écran

La commande `write` permet d'afficher du texte et de laisser le curseur à la fin du texte affiché. Cette commande permet d'afficher des chaînes de caractères d'excédant pas 255 signes ainsi que des valeurs de variables, de constantes, de types... Le texte doit être entre apostrophe. Si le texte à afficher contient une apostrophe, il faut alors la doubler. Les différents noms de variables doivent êtres séparés par des virgules.

Note : toute commande doit être suivie d'un point virgule.

Syntaxe :

```
Write ('Texte à afficher', variable1, variable2, 'texte2') ;
Write ('L''apostrophe se double.') ;
```

La commande `WriteLn` est semblable à la précédente à la différence près que le curseur est maintenant renvoyé à la ligne suivante.

Syntaxe :

```
WriteLn ('Texte avec renvoi à la ligne') ;
```

La commande `read` permet à l'utilisateur de rentrer une valeur qui sera utilisée par le programme. Cette commande ne provoque pas de retour Chariot, c'est-à-dire que le curseur ne passe pas à la ligne.

Syntaxe :

```
Read (variable) ;
```

La commande `ReadLn` permet à l'utilisateur de rentrer une valeur qui sera utilisée par le programme. Cette commande provoque le retour Chariot, c'est-à-dire que le curseur passe à la ligne suivante. Lorsqu'aucune variable n'est affectée à la commande, il suffit de presser sur <ENTREE>.

Syntaxe :

```
ReadLn (variable1, variable2) ;

ReadLn ;
```

```
Program exemple1;
Var nom : String ;
BEGIN
   Write('Entrez votre nom : ') ;
   ReadLn(nom) ;
   WriteLn('Votre nom est ', nom) ;
   ReadLn ;
END.
```

Ce programme *exemple1* déclare tout d'abord la variable nommée *nom* comme étant une chaîne de caractère (`String`). Ensuite, dans le bloc programme principal, il est demandé à l'utilisateur d'affecter une valeur à la variable *nom* qui est initialisée automatiquement (valeur nulle) à chaque démarrage du programme. Ensuite, il y a affichage de la valeur de la variable et attente que la touche entrée soit validée (`ReadLn`).

L'équivalent de la commande `ReadLn` est `ReadKey` qui donne une valeur à une variable de type `Char` (caractère ASCII).

Syntaxe :

```
x := ReadKey ;
```

Il existe une équivalence à cette commande très utile pour sortir d'une boucle : `KeyPressed`.

Syntaxe :

```
Repeat
...
commandes
...
Until KeyPressed ;
```

```
Program exemple2 ;
Uses crt ;
Var i : integer ;
Const bornesup=10000 ;
BEGIN
   Repeat
   WriteLn(sqrt(i)) ;
   Inc(i) ;
   Until (i=bornesup) or KeyPressed ;
END.
```

Ce programme *exemple2* répète une boucle jusqu'à qu'une valeur soit atteinte (*bornesup*) mais s'arrête si on appuie sur une touche. L'instruction `Inc(a,n);` incrémente la valeur n à la variable a (par défault n vaut 1), cette dernière étant de type `integer`.

CHAPITRE II : Opérateurs

Opérateurs mathématiques

Addition (et union[1]) +
Soustraction (et complément[1]) -
Division /
Multiplication (et intersection[1]) *
Egalité =
MOD : renvoie le reste de la division x MOD y
DIV : renvoie le quotient de la division x DIV y
*Opérateurs prioritaires : *, /, DIV et MOD. Opérateurs secondaires : + et -. Vous pouvez utiliser des parentèses.*

Opérateurs relationnels

Inférieur strict <
Inférieur ou égale (et inclu[1]) <=
Supérieur strict >
Supérieur ou égale (et contenant[1]) >=
Différent <>
Opérateur ultra-prioritaire : NOT. Opérateur semi-prioritaire : AND. Opérateur non prioritaires : OR et XOR.

Opérateurs logiques :

AND : le "et" logique des maths (voir chapitre 15 sur les booléens et tables de vérité)
OR : le "ou"
XOR : le "ou" exclusif
NOT : le "non"

Priorité des opérateurs

Niveau 1 : NOT.
Niveau 2 : *, /, MOD, DIV, AND.
Niveau 3 : +, -, OR, XOR.
Niveau 4 : =, <, >, <=, >=, <>.

[1]: les opérateurs union, complément, intersection, inclu et contenant s'appliquent aux ensembles (voir Chap XXV).

CHAPITRE III : Variables, formats et maths

1. Déclaration

Toutes les variables doivent êtres préalablement déclarées avant d'être utilisées dans le programme, c'est-à-dire qu'on leur affecte un type (voir types de variables). On peut les déclarer de divers manières :

- Au tout début du programme avec la syntaxe **VAR** ***nom de la variable*** **:** ***type*** **;** elles seront alors valables pour le programme dans son intégralité (sous-programmes, fonctions, procédures...).
- Au début d'une procédure avec la syntaxe précédente. Elles ne seront valables que dans la procédure.
- Après la déclaration des procédures, toujours avec la même syntaxe, elles ne pourront alors pas êtres utilisée par les procédures qui devront donc être paramétrées (voir procédures paramétrées).

2. Prise de valeurs

Les variables sont faites pour varier, il faut donc pouvoir leur donner différentes valeurs au moyen du commutateur suivant **:=** (deux points et signe égale) ou de certaines fonction. Il faut bien sûr que la valeur donnée soit compatible avec le type utilisé. Ainsi, on ne peut donner la valeur `'bonjour'` à un nombre entier (`integer`).

Syntaxes :

```
Y := 1998 ;
```

On donne ainsi la valeur 1998 à la variable Y (déclarée préalablement en INTEGER).

```
LETTRE := 'a' ;
```

On affecte la valeur **a** à la variable LETTRE (déclarée préalablement en CHAR).

```
TEST := true ;
```

On donne la valeur **true** (vrai) à la variable TEST (déclarée préalablement en BOOLEAN).

```
NOMBRE := Y + 103 ;
```

Il est ainsi possible d'utiliser les valeurs d'autres variables, du moment qu'elles sont de même type, sinon, il faut faire des conversions au préalable.

```
DELTA := sqr(b) - 4*(a*c) ;
```

On peut donc également utiliser une expression littérale mathématique dans l'affectation de variables. Mais attention à la priorité des opérateurs (voir opérateurs).

```
PHRASE := 'Bonjour' + chr(32) + NOM ;
```

On peut aussi ajouter des variables `String` (voir Chapitre 13 pour les chaînes de caractères).

---

3. Fonctions

Fonction mathématiques Pascal de base

Syntaxe	Fonction
`Sin(a)`	sinus
`Cos(a)`	cosinus
`ArcTan(a)`	arctangeante
`Abs(a)`	valeur absolue
`Sqr(a)`	carré
`Sqrt(a)`	racine carré
`Exp(a)`	exponentielle
`Ln(a)`	logarithme népérien

L'argument des fonctions trigonométriques doit être exprimé en radian (`Real`), à vous donc de faire une convertion si nécessaire. De plus, on peut voir que les fonctions tangente, factorielle n'existent pas, il faudra donc créer de toute pièce les fonctions désirées (voir fonctions).

---

4. Emplois

Les variables peuvent êtres utilisées dans de nombreux emplois :

- Pour des comparaisons dans une structure conditionnelle (voir chapitre 4 sur les conditions).
- Pour l'affichage de résultats (voir chapitre 1 sur l'affichage).
- Pour le dialogue avec l'utilisateur du programme (voir chapitre 1 sur les entrées au clavier).
- Pour excécuter des boucles (voir chapitre 6)...

---

5. Opérations

Syntaxe	Utilisation	Type des variables	Description
`Inc(a);`	Procédure	intervalle ou énuméré	Le nombre a est incrémenté de 1
`Inc(a,n);`	Procédure	intervalle ou énuméré	Le nombre a est incrémenté de n
`Dec(a);`	Procédure	intervalle ou énuméré	Le nombre a est décrémenté de 1
`Dec(a,n);`	Procédure	intervalle ou énuméré	Le nombre a est décrémenté de n
`Trunc(a)`	Fonction	tout scalaire	Prise de la partie entière du nombre a sans arrondis
`Int(a)`	Fonction	`a:Real` `Int(a):Longint`	Prise de la partie entière du nombre a sans arrondis
`Frac(a)`	Fonction	`Real`	Prise de la partie fractionnaire du nombre a

`Round(a)`	Fonction	`a:Real` `Round(a):Longint`	Prise de la partie entière du nombre a avec arrondi à l'unité la plus proche
`Pred(x)`	Fonction	intervalle ou énuméré	Renvoit le prédécesseur de la variable x dans un ensemble ordonnée
`Succ(x)`	Fonction	intervalle ou énuméré	Renvoit le successeur de la variable x dans un ensemble ordonnée
`Hight(x)`	Fonction	tous	Renvoit la plus grande valeur possible que peut prendre de la variable x
`Low(x)`	Fonction	tous	Renvoit la plus petite valeur possible que peut prendre de la variable x
`Odd(a)`	Fonction	`a:Longint` `Odd(a):Boolean`	Renvoit `true` si le nombre a est impair et `false` si a est pair
`SizeOf(x)`	Fonction	`x:`tous `SizeOf(x):Integer`	Renvoit ~~renvoit~~ le nombre d'octets occupés par la variable x

6. Format

Sachez encore que le format (le nombre de signes) d'une variable de type `real` peut être modifié :

- Lors de son affichage : `WriteLn ( nombre : 5 ) ;` pour mettre 5 espaces devant le nombre.
- Lors de son affichage (bis) : `WriteLn ( nombre : 0 : 5 ) ;` pour ne mettre aucun espace avant mais pour n'afficher que 5 signes (un réel en possède bien plus).

Pour pouvez appliquer ce format pour tous les autres types de variable de manière générale si vous ne stipuler que le nombre d'espace(s) à afficher devant votre texte ou valeur.

Exemple : `WriteLn ( 'Coucou' : 20 ) ;`
Ici, la chaîne de caractères sera affichée après 20 espaces.

CHAPITRE IV : Différents types de variables

On peut donner n'importe quel nom aux variables à condition qu'il ne fasse pas plus de 127 caractères et qu'il ne soit pas utilisé par une fonction, procédure, unité ou commande déjà existante.
Les identificateurs ne doivent pas contenir de caractères accentués, ni d'espace. Ils doivent exclusivement être composés des 26 lettres de l'alphabet, des 10 chiffres et du caractère de soulignement. De plus, Turbo Pascal ne différencie aucunement les majuscules des minuscules et un chiffre ne peut pas être placé en début de nom de variable.

Petite liste-exemple très loin d'être exhaustive :

Désignation	Description	Bornes	Place en mémoire
REAL	nombres réels	2.9E-039 et 1.7E+038	6 octets
SINGLE(*)	réel	1.5E-045 et 3.4E+038	4 octets
DOUBLE(*)	réel	5.0E-324 et 1.7E+308	8 octets
EXTENDED(*)	réel	1.9E-4951 et 1.1E+4932	10 octets
COMP(*)	réel	-2E+063 +1 et 2E+063 +1	8 octets
INTEGER	nombres entier (sans virgule)	-32768 et 32767	2 octets
LONGINT	entier	-2147483648 et 2147483647	4 octets
SHORTINT	entier	-128 et 127	1 octet
WORD	entier	0 et 65535	2 octets
BYTE	entier	0 et 255	1 octet
LONG	entier	$(-2)^{31}$ et $(2^{31})-1$	4 octets
BOOLEAN	variable booléenne	TRUE ou FALSE	1 octet
ARRAY [1..10] OF xxx	tableau de 10 colones fait d'éléments de l'ensemble défini xxx (CHAR, INTEGER...)		
ARRAY [1..10, 1..50, 1..13] OF xxx	tableau en 3 dimensions fait d'éléments de l'ensemble défini xxx (CHAR, INTEGER...)		
STRING	chaîne de caractères		256 octets
STRING [y]	chaîne de caractère ne devant pas excéder y caractères		y+1 octets
TEXT	fichier texte		
FILE	fichier		
FILE OF xxx	fichier contenant des données de type xxx (REAL, BYTE...)		
CHAR	nombre correspondant à un caractère ASCII codé	0 et 255	1 octet
POINTEUR	adresse mémoire		4 octet
DATETIME	format de date		
PATHSTR	chaîne de caractère (nom		

	complet de fichier)		
DIRSTR	chaîne de caractère (chemin de fichier)		
NAMESTR	chaîne de caractère (nom de fichier)		
EXTSTR	chaîne de caractère (extention de fichier)		

(*) : nécessitent un co-processeur mathématique.

CHAPITRE V : Structures alternatives

1. If ... Then ... Else

Cette commande est similaire au basic, elle se traduit par : SI … ALORS … SINON …

```
Program exemple3a ;
Var chiffre:integer ;
BEGIN
   Write('Entrez un entier pas trop grand : ') ;
   Readln(chiffre) ;
   If chiffre < 100 then writeln(chiffre, ' est inférieur à cent.') else
   writeln(chiffre, ' est supérieur ou égale à cent.') ;
END.
```

Ce programme *exemple3a* compare un chiffre entré par l'utilisateur au scalaire 100. Si le chiffre est inférieur à 100, alors il affiche cette information à l'écran, sinon il affiche que le chiffre entré est supérieur ou égale à 100.

```
Program exemple3b ;
Var chiffre:integer ;
BEGIN
   Write('Entrez un entier pas trop grand : ') ;
   Readln(chiffre) ;
   If chiffre < 100 then
      begin
      writeln(chiffre, ' est inférieur à cent.') ;
      end
   else
      begin
      writeln(chiffre, ' est supérieur ou égale à cent.') ;
      end ;
END.
```

Ce programme *exemple3b* fait strictement la même chose que le *3a* mais sa structure permet d'insérer plusieurs autres commandes dans les sous-blocs THEN et ELSE. Notez que le END terminant le THEN ne possède pas de point virgule car s'il en possédait un, alors le ELSE n'aurait rien à faire ici et le bloc condition se stopperait avant le ELSE.
Il est également possible d'insérer d'autres bloc IF dans un ELSE, comme l'illustre l'exemple3c qui suit :

```
Program exemple3c ;
Var i : integer ;
BEGIN
```

```
   Randomize ;
   i := random(100) ;
   if i < 50 then writeln ( i, ' est inférieur à 50.')
   else if i < 73 then writeln ( i, ' est inférieur à 73.')
   else writeln ( i, ' est supérieur ou égale à 73.')
END.
```

2. Case ... Of ... End

Cette instruction compare la valeur d'une variable de type entié ou caractère (et de manière générale de type intervalle, voir Chap Type) à tout un tas d'autres valeurs constantes.

Note : attention car `Case Of` ne permet de comparer une variable qu'avec des constantes.

```
Program exemple4 ;
Var age:integer ;
BEGIN
   Write('Entrez votre âge : ') ;
   Readln(age) ;
   Case age of
      18 : writeln('La majorité, pile-poil !') ;
      0..17 : writeln('Venez à moi, les petits enfants…') ;
      60..99 : writeln('Les infirmières vous laisse jouer sur l''ordinateur à
      votre âge ?!!!')
      Else writeln('Vous êtes d''un autre âge...') ;
   End ;
END.
```

Ce programme *exemple4a* vérifie certaines conditions quant à la valeur de la variable *age* dont l'a affecté l'utilisateur. Et là, attention : le point-virgule avant le `Else` est facultatif. Mais pour plus sécurité afin de ne pas faire d'erreur avec le bloc `If`, choisissez systématiquement d'ommettre le point-virgule avant un `Else`.

Note : On peut effectuer un test de plusieurs valeurs en une seule ligne par séparartion avec une virgule si on souhaite un même traitement pour plusieurs valeurs différentes. Ainsi la ligne :

```
   0..17 : writeln('Venez à moi, les petits enfants…') ;
```

peut devenir :

```
   0..10, 11..17 : writeln('Venez à moi, les petits enfants…') ;
```

ou encore :

```
   0..9, 10, 11..17 : writeln('Venez à moi, les petits enfants…') ;
```

ou même :

```
   0..17, 5..10 : writeln('Venez à moi, les petits enfants…') ;
```

{ cette dernière ligne est stupide mais correcte ! }

CHAPITRE VI : Structures répétitives

1. For ... : = ... To ... Do ...

Cette instruction permet d'incrémenter une variable à partir d'une valeur inférieur jusqu'à une valeur supérieur et d'exécuter une ou des instructions entre chaque incrémentation. Les valeurs extrémum doivent être des entiers (`integer`) ou des caractères de la table ASCII (`char`). De manière plus générale, les bornes doivent être de type intervalle (voir chap Type) c'est-à-dire qu'ils doivent êtres de type entier ou compatibles avec un type entier. La boucle n'exécute les instructions de son bloc interne que si la valeur inférieur est effectivement inférieur ou égale à celle de la borne supérieur. Le pas de variation est l'unité et ne peut pas être changé.

Syntaxe :

```
For variable := borne inférieur To borne supérieur Do instruction ;
```

Autre **Syntaxe :**

```
For variable := borne inférieur To borne supérieur Do
     Begin
     ...
     commandes
     ...
     End ;
```

```
Program exemple5 ;
Var i : integer ;
BEGIN
   For i := 10 To 53 Do writeln ('Valeur de i : ', i ) ;
END.
```

---

2. For ... : = ... DownTo ... Do ...

Cette instruction permet de décrémenter une variable à partir d'une valeur supérieur jusqu'à une valeur inférieur et d'exécuter une ou des instructions entre chaque décrémentation. S'appliquent ici les mêmes remarques que précédement.

Syntaxe :

```
For variable := borne supérieur DownTo borne inférieur Do instruction ;
```

Autre **Syntaxe :**

```
For variable := borne supérieur DownTo borne inférieur Do
     Begin
     ...
     commandes
```

```
   ...
   End ;

Program exemple6 ;
Var i : integer ;
BEGIN
   For i := 100 DownTo 0 Do
        Begin
        WriteLn ('Valeur de i : ', i ) ;
        End ;
END.
```

---

3. Repeat ... Until ...

Cette boucle effectue les instructions placées entre deux bornes (`repeat` et `until`) et évalue à chaque répétition une condition de type bouléenne avant de continuer la boucle pour décider l'arrêt ou la continuité de la répétition. Il y a donc au moins une fois exécution des instructions. Il est nécessaire qu'au moins une variable intervenant lors de l'évaluation de fin de boucle soit sujette à modification à l'intérieur de la structure exécutive interne à la boucle.

Syntaxe :

```
Repeat
...
commandes
...
Until variable condition valeur ;
```

```
Program exemple7 ;
Uses crt ;
Var i : integer ;
BEGIN
     Repeat
          Inc ( i , 1 ) ;
          Writeln ('Boucle itérée ', i, ' fois.') ;
       Until i > 20 ;
END.
```

Ce programme *exemple7* permet de répéter l'incrémentation de la variable *i* jusqu'à que *i* soit supérieure à 20.

Note : la commande `Inc` permet d'incrémenter une variable d'une certaine valeur. La commande `Dec` permet au contraire de décrémenter une variable d'une certaine valeur. Ces commandes permettent d'éviter la syntaxe : *variable* `:=` *variable* `+ 1` et *variable* `:=` *variable* `- 1`.

Syntaxe :

```
Inc ( variable , nombre ) ;
Dec ( variable , nombre ) ;
```

---

4. While ... Do ...

Ce type de boucle, contrairement à la précédente, évalue une condition avant d'exécuter des instructions (et nom pas l'inverse), c'est-à-dire qu'on peut ne pas entrer dans la structure de répétition si les conditions ne sont pas favorables. De plus, au moins une variable de l'expression d'évaluation doit être sujette à modification au sein de la structure de répétition pour qu'on puisse en sortir.

Syntaxe :

```
While variable condition valeur Do instruction ;
```

Autre **Syntaxe :**

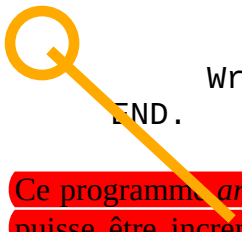
```
While variable condition valeur Do
      Begin
      ...
      commandes
      ...
      End ;
```

```
Program exemple8 ;
Var code : boolean ;
essai : string ;
Const levraicode = 'password' ;
BEGIN
      code:=false ; { facultatif, la valeur false est donnée par défaut }
      While code = false Do
            Begin
                  Write ('Entrez le code secret : ') ;
                  Readln (essai) ;
                  If essai = levraicode then code:=true ;
               End ;
   END.
```

5. Arrêts de boucle.

Il est possible de terminer une boucle `For`, `While` ou `Repeat` en cours grâce à la commande `Break` lorsque celle-ci est placée au sein de la boucle en question.
Pour reprendre une boucle stoppée par `Break`, il faut utiliser la commande `Continue`.

```
Program arrets1 ;
Var i, x : Integer ;
BEGIN
      x := 0 ;
            Repeat
                  Inc(i) ;
                  Break ;
                  x := 50 ;
                  Continue ;
```



```
            Until i>10 ;
         WriteLn(x) ;
END.
```

Ce programme *arrets1* stoppe systématiquement une boucle `Repeat` avant que la variable x puisse être incrémenté de 50 et la reprend après la ligne d'incrémentation. Ce qui a pour résultats que la variable x soit nulle à la fin du programme.

```
Program arrets2 ;
Var i, x : Integer ;
BEGIN
     x := 0 ;
     For i := 1 to 10 Do
          Begin
               Break ;
               x := 50 ;
               Continue ;
            End ;
         WriteLn(x) ;
   END.
```

Ce programme *arrets2* fait la même chose que le programme précédent mais dans une boucle `For`.

```
Program arrets3 ;
Var i, x : Integer ;
BEGIN
     x := 0 ;
     While i<10 Do
          Begin
               Break ;
               x := 50 ;
               Continue ;
            End ;
         WriteLn(x) ;
   END.
```

Ce programme *arrets3* fait la même chose que les programmes précédents mais dans une boucle `While`.

Et pour quitter un bloc sous-programme (structure `Begin ... End ;`) ou même le programme principal (structure `Begin ... End.`), utilisez la commande `Exit`.

```
Program arrets4 ;
Var i : Integer ;
BEGIN
     While i <> 13 Do
          Begin
               Write ('Entrez un nombre : ') ;
               Readln (i) ;
```

```
                Writeln (i) ;
                If i = 0 Then Exit ;
              End ;
        Writeln ('Boucle terminée.') ;
END.
```

CHAPITRE VII : Procédures

Les procédures et fonctions sont des sortes de sous-programmes écrits avant le programme principal mais appelés depuis ce programme principal, d'une autre procédure ou même d'une autre fonction. Le nom d'une procédure ou d'une fonction (ou comme celui d'un tableau, d'une variable ou d'une constante) de doit pas excéder 127 caractères et ne pas contenir d'accent. Ce nom doit, en outre, être différent de celui d'une instruction en Pascal. L'appel d'une procédure peut dépendre d'une structure de boucle, de condition, etc.

1. Procédure simple

Une procédure peut voir ses variables définies par le programme principal, c'est-à-dire que ces variables sont valables pour tout le programme et accessible partout dans le programme principal mais aussi dans les procédures et fonctions qui auront été déclarées après. La déclaration des variables se fait alors avant la déclaration de la procédure pour qu'elle puisse les utiliser. Car une procédure déclarée avant les variables ne peut pas connaître leur existence et ne peut donc pas les utiliser.

Syntaxe :

```
Program nom de programme ;
Var variable : type ;
Procedure nom de procédure ;
   Begin
   ...
   commandes
   ...
   End ;
BEGIN
nom de procédure ;
END.
```

```
Program exemple9a ;
Uses crt ;
Var a, b, c : real ;
Procedure maths ;
   Begin
      a := a + 10 ;
      b := sqrt(a) ;
      c := sin(b) ;
   End ;
BEGIN
   Clrscr ;
   Write('Entrez un nombre :') ;
```

```
    Readln(a) ;
    Repeat
       maths ;
       Writeln (c) ;
    Until keypressed ;
END.
```

Ce programme *exemple9a* appelle une procédure appelée *maths* qui effectue des calculs successifs. Cette procédure est appelée depuis une boucle qui ne se stoppe que lorsqu'une touche du clavier est pressée (instruction `keypressed`). Durant cette procédure, on additionne 10 à la valeur de **a** entrée par l'utilisateur, puis on effectue le carré (`sqrt`) du nombre ainsi obtenu, et enfin, on cherche le sinus (`sin`) de ce dernier nombre.

---

2. Variables locales et sous-procédures

Une procédure peut avoir ses propres variables locales qui seront réinitialisées à chaque appel. Ces variables n'existent alors que dans la procédure. Ainsi, une procédure peut utiliser les variables globales du programme (déclarées en tout début) mais aussi ses propres variables locales qui lui sont réservées. Une procédure ne peut pas appeler une variable locale appartenant à une autre procédure. Les variables locales doivent porter des noms différents de celles globales si ces dernières ont été déclarée avant la procédure. Enfin, on peut utiliser dans une procédure, un nom pour une variable locale déjà utilisé pour une autre variable locale dans une autre procédure.

Une procédure, étant un sous-programme complet, peut contenir ses propres procédures et fonctions qui n'existent alors que lorsque la procédure principale est en cours. Un sous-procédure ne peut appeler d'autres procédures ou fonctions que si ces dernières font parti du programme principal ou de la procédure qui contient la sous-procédure.

Syntaxe :

```
Procedure nom de procédure ;
Var variable : type ;
   Procedure nom de sous-procédure ;
   Var variable : type ;
      Begin
      …
      End ;
   Begin
   …
   commandes
   …
   End ;
```

---

3. Procédure paramétrée

On peut aussi créer des procédures paramétrées (dont les variables n'existent que dans la procédure). Ces procédures là ont l'intérêt de pouvoir, contrairement aux procédures simples, être déclarée avant les variables globales du programme principal ; elles n'utiliseront que les

variables passées en paramètres ! Le programme principal (ou une autre procédure qui aurait été déclarée après) affecte alors des valeurs de variables à la procédure en passant des variables en paramètres. Et ces valeurs s'incorporent dans les variables propres à la procédure (dont les identificateurs peuvent ou non êtres identiques, ça n'a aucune espèce d'importance). La déclaration des variables se fait alors en même temps que la déclaration de la procédure, ces variables sont des paramètres formels car existant uniquement dans la procédure. Lorsque que le programme appelle la procédure et lui envoie des valeurs de type simple (car celles de type complexe ne sont pas acceptées, voir chapitre 20 sur les types), celles-ci sont appelées paramètres réels car les variables sont définies dans le programme principal et ne sont pas valables dans la procédure.

A noter qu'on peut passer en paramètre directement des valeurs (nombre, chaînes de caractères...) aussi bien que des variables.

Syntaxe :

```
Program nom de programme ;
Procedure nom de procédure( noms de variables : types ) ;
   Begin
   ...
   commandes
   ...
   End ;
BEGIN
nom de procédure ( noms d'autres variables ou leurs valeurs ) ;
END.
```

Note : on peut passer en paramètre à une procédure des types simples et structurés. Attention néanmoins à déclarer des types spécifiques de tableau à l'aide de la syntaxe `Type` (voir Chapitre 20 sur les "*Types simples et structurés*") car le passage d'un tableau en tant que type `Array` à une procédure est impossible.

```
Program exemple9b ;
Uses Crt ;
Procedure maths ( param : Real ) ;
   Begin
      WriteLn('Procédure de calcul. Veuillez patienter.') ;
      param := Sin(Sqrt(param+10)) ;
      WriteLn(param) ;
   End ;
Var nombre : Real ;
BEGIN
   ClrScr ;
   Write('Entrez un nombre :') ;
   ReadLn(nombre) ;
   maths (nombre) ;
   ReadLn ;
END.
```

Ce programme *exemple9b* appelle une procédure paramétrée appelée *maths* qui effectue les mêmes calculs que le programme *exemple9a*. Mais ici, la variable est déclarée après la procédure paramétrée. Donc, la procédure ne connaît pas l'existdnce de la variable *nombre*, ainsi, pour qu'elle puisse l'utiliser, il faut le lui passer en paramètre !

---

4. Syntaxe `Var` (procédures et fonctions)

Il est quelquefois nécessaire d'appeler une procédure paramétrée sans pour autant avoir de valeur à lui affecter mais on souhaiterait que ce soit elle qui nous renvoie des valeurs (exemple typique d'une procédure de saisie de valeurs par l'utilisateur) ou alors on désire que la procédure puisse modifier la valeur de la variable passée en paramètre. Les syntaxes précédentes ne conviennent pas à ce cas spécial. Lors de la déclaration de variable au sein de la procédure paramétrée, la syntaxe `Var` (placée devant l'identificateur de la variable) permet de déclarer des paramètres formels dont la valeur à l'intérieur de la procédure ira remplacer la valeur, dans le programme principal, de la variable passée en paramètre. Et lorsque `Var` n'est pas là, les paramètres formels doivent impérativement avoir une valeur lors de l'appel de la procédure.

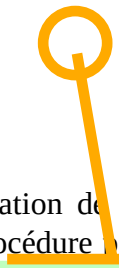
Pour expliquer autrement, si `Var` n'est pas là, la variable qu'on envoie en paramètre à la procédure doit absolument déjà avoir une valeur (valeur nulle acceptée). De plus, sans `Var`, la variable (à l'intérieur de la procédure) qui contient la valeur de la variable passée en paramètre, même si elle change de valeur n'aura aucun effet sur la valeur de la variable (du programme principal) passée en paramètre. C'est à dire que la variable de la procédure n'existe qu'à l'intérieur de cette dernière et ne peut absolument pas affecter en quoi que ce soit la valeur initiale qui fut envoyée à la procédure : cette valeur initiale reste la même avant et après l'appel de la procédure. Car en effet, la variable de la procédure est dynamique : elle est créée lorsque la procédure est appelée et elle est détruite lorsque la procédure est finie, et ce, sans retour d'information vers le programme principal. La procédure paramétrée sans `Var` évolue sans aucune interaction avec le programme principal (même si elle est capable d'appeler elle-même d'autres procédures et fonctions).

Par contre, si `Var` est là, la valeur de la variable globale passée en paramètre à la procédure va pouvoir changer (elle pourra donc ne rien contenir à l'origine). Si au cours de la procédure la valeur est changée (lors d'un calcul, d'une saisie de l'utilisateur...), alors la nouvelle valeur de la variable dans la procédure, une fois la procédure terminée, ira se placer dans la variable globale (du programme principal) qui avait été passée en paramètre à la procédure. Donc, si on veut passer une variable en paramètre dont la valeur dans le programme principal ne doit pas être modifiée (même si elle change dans la procédure), on n'utilise pas le `Var`. Et dans le cas contraire, si on veut de la valeur de la variable globale placée en paramètre change grâce à la procédure (saisie, calcul...), on colle un `Var`.

```
Program Exemple9c ;
Uses Crt ;
Procedure Saisie ( var nom : String ) ;
   Begin
      Write('Entrez votre nom : ') ;
      ReadLn(nom) ;
   End ;
```

```
Procedure Affichage ( info : String ) ;
   Begin
      WriteLn('Voici votre nom : ', info) ;
   End ;
Var chaine : String ;
BEGIN
   ClrScr ;
   Saisie(chaine) ;
   Affichage(chaine) ;

END.
```



Ce programme *exemple9c* illustre l'utilisation de la syntaxe `Var`. En effet, le programme principal appelle pour commencer une procédure paramétrée *Saisie* et lui affecte la valeur de la variable *chaine* (c'est-à-dire rien du tout puisque qu'avant on n'a rien mis dedans, même pas une chaîne vide). Au sein de la procédure paramétrée, cette variable porte un autre nom : *nom*, et comme au début du programme cette variable n'a aucune valeur, on offre à la procédure la possibilité de modifier le contenu de la variable qu'on lui envoie, c'est-à-dire ici d'y mettre le nom de l'utilisateur. Pour cela, on utilise la syntaxe `Var`. Lorsque cette procédure *Saisie* est terminée, la variable *chaine* du programme principal prend la valeur de la variable *nom* de la procédure. Ensuite, on envoie à la procédure *Affichage* la valeur de la variable *chaine* (c'est-à-dire ce que contenait la variable *nom*, variable qui fut détruite lorsque la procédure *Saisie* se termina). Comme cette dernière procédure n'a pas pour objet de modifier la valeur de cette variable, on n'utilise pas le mot clé `Var`, ainsi, la valeur de la variable *chaine* ne pourra pas être modifiée par la procédure. Par contre, même sans `Var`, la valeur de la variable *info* pourrait varier au sein de la procédure si on le voulait mais cela n'aurait aucune influence sur la variable globale *chaine*. Comme cette variable *info* n'est définie que dans la procédure, elle n'existera plus quand la procédure sera terminée.

Il faut savoir qu'une procédure paramétrée peut accepter, si on le désire, plusieurs variables d'un même type et aussi plusieurs variables de types différents. Ainsi, certaines pourront être associées au `Var`, et d'autres pas. Si l'on déclare, dans une procédure paramétrée, plusieurs variables de même type dont les valeurs de certaines devront remplacer celles initiales, mais d'autres non ; il faudra déclarer séparément (séparation par une virgule `;`) les variables déclarée avec `Var` et les autres sans `Var`. Si on déclare plusieurs variables de types différents et qu'on veut que leurs changements de valeur affecte les variables globales, alors on devra placer devant chaque déclaration de types différents un `Var`.

Syntaxes :

```
Procedure identifiant(Var var1, var2 : type1 ; var3 : type1) ;
Begin
…
End ;
Procedure identifiant(Var var1 : type1 ; Var var2 : type2) ;
Begin
…
End ;
```

CHAPITRE VIII : Fonctions

Quant aux fonctions, elle sont appelées à partir du programme principal, d'une procédure ou d'une autre fonction. Le programme affecte des valeurs à leur variables (comme pour les procédures paramétrées, il faudra faire attention à l'ordre d'affectation des variables). Ces fonctions, après lancement, sont affectées elles-mêmes d'une valeur intrinsèque issue de leur fonctionnement interne. Il faut déclarer une fonction en lui donnant tout d'abord un identifiant (c'est-à-dire un nom d'appel), en déclarant les variables locales dont elle aura besoin et enfin, il faudra préciser le type correspondant à la valeur que prendra en elle-même la fonction (`string`, `real`, etc.). Attention, on ne peut pas affecter un type complexe (`array`, `record`) à une fonction : seuls les types simples sont acceptés (voir chapitre 20 sur les types simples et complexes). De plus, comme le montre les syntaxes suivantes, on peut fabriquer une fonction sans paramètre (ex: `random`). Il ne faut surtout pas oublier, en fin (ou cours) de fonction, de donner une valeur à la fonction c'est-à-dire d'affecter le contenu d'une variable ou le résultat d'une opération (ou autre...) à l'identifiant de la fonction (son nom) comme le montrent les syntaxes suivantes.

Syntaxes :

```
Function nom de fonction (variable : type ) : type ;
Var déclaration de variables locales ;
   Begin
   ...
   commandes
   ...
   nom de fonction := une valeur ;
   End ;

Function nom de fonction : type ;
Var déclaration de variables locales ;
   Begin
   ...
   commandes
   ...
   nom de fonction := une valeur ;
   End ; >
```

```
Program exemple10 ;
Uses crt ;
Function exposant ( i , j : integer ) : integer ;
Var i2 , a : integer ;
   Begin
      i2 := 1 ;
      For a := 1 To j Do i2 := i2 * i ;
      exposant := i2 ;
   End ;
Var resultat, x, n : integer ;
BEGIN
```

```
    Write ('Entrez un nombre : ') ;
    Readln (x) ;
    Write('Entrez un exposant : ') ;
    Readln (n) ;
    resultat := exposant ( x , n ) ;
    Writeln ( resultat ) ;
    Readln ;
END.
```

CHAPITRE IX : Audio

1. Sound ... Delay ... Nosound

Pour faire du son, il faut indiquer la fréquence (f) en Hz et le délay (t) en ms.

Syntaxe :

```
Sound ( f ) ;
Delay ( t ) ;
Nosound ;
```

```
Program exemple11 ;
Uses crt ;
Var i, f : integer ;
BEGIN
   For i := 1 to 20 do
      Begin
         For f := 500 to 1000 do sound ( f ) ;
         Delay (10) ;
      End ;
   Nosound ;
END.
```

---

2. Chr (7)

La fonction `Chr` permet d'obtenir le caractère de la table ASCII correspondant au numéro. Il se trouve que les 31 premiers caractères correspondent à des fonctions : beep, delete, insert, return, esc... Le caractère 7 correspond au beep.

Syntaxes :

```
Write ( chr ( 7 ) ) ;
Write ( #7 ) ;
```

CHAPITRE X : Manipulation de fichiers

1. Déclaration

Pour utiliser un ou des fichiers tout au long d'un programme, il faudra l'identifier par une variable dont le type est fonction de l'utilisation que l'on veut faire du fichier. Il existe trois types de fichiers :

- Les **fichiers textes** (`Text`), qui sont écrits au format texte (chaînes de caractères, nombres) dans lesquels ont peut écrire et lire ligne par ligne ou à la file avec les procédures `Write(Ln)` et `Read(Ln)`. Chaque fin de ligne du fichier se termine par les caractères 10 et 13 de la table ASCII qui signifient respectivement *retour chariot* et *passage à la ligne*. Ces deux derniers caractères sont transparent au programmeur. On pourra donc y écrire ou y lire indifféremment des chaînes ou des nombres, cela dépend du type que l'on affecte à la variable passée en paramètre aux procédures d'entrée/sorties (voir plus bas).

  Note : S'il y a lieu de faire une conversion nombre/chaîne, le compilateur le fait tout seul, par contre si le type de la variable ne correspond pas avec la donnée lue dans le fichier et qu'aucune conversion n'est possible (exemple : `WriteLn(f, x:Real);` alors que le fichier ne contient que des lettres), alors cela produit une erreur.

  Syntaxe :
  ```
  Var f : Text ;
  ```

- Les **fichiers typés** (`File Of`), qui sont des fichiers écrits sur disque telles que les données se présentent en mémoire. C'est-à-dire que la taille du fichier résultera directement et exactement de la taille en mémoire qu'occupe telle ou telle variable. Cela accrue la vitesse d'accès aux données du fichiers. Mais le plus grand avantage c'est que l'on obtient ainsi des fichiers parfaitement formatés, c'est-à-dire qu'on peut y lire et écrire directement des variables de type structuré qui contiennent plusieurs champs de données (voir chap Type) sans avoir à se soucier des divers champs qu'elles contiennent. Il va sans dire que ce type de fichier est préférable à tous les autres.

  Syntaxe :
  ```
  Var f : File Of type ;
  ```

  Exemple :
  ```
  Var f : File Of Integer ;
  ```

- Les **fichiers** tout court ! (`File`), qui sont des fichiers dont on ne connaît pas le contenu. N'ayant aucune information sur la structure des données, n'ayant aucune conversion à faire, la lecture et son écriture en sont plus rapide. Mais sont utilité est bien maigre : à part faire une simple copie d'un fichier dans un autre...

Syntaxe :
```
Var f : File ;
```

2. Lecture, écriture

Avant de travailler sur un fichier, il faut le déclarer en lui affectant une variable qui servira à désigner le fichier tout au long du programme. `Assign` s'applique à tous les types de fichiers (`Text`, `File Of` et `File`).

Syntaxe :
```
Assign ( variable d'appel , nom du fichier ) ;
```

Ensuite, il faut renvoyer le pointeur au début du fichier pour pouvoir lire (`Text`, `File Of` et `File`) ou écrire (`File Of` et `File`) à partir du début du fichier. Attention, on ne peut pas écrire sur un `Text` avec `Reset` !

Syntaxe :
```
Reset ( variable d'appel ) ;
```

Il est possible pour le type `File` uniquement, de spécifier la taille de chaque bloc de donnée lu ou écrit sur le fichier, en rajoutant en argument à `Reset` une variable (ou un nombre directement) de type `Word` (entier) spécifiant cette taille en octet. Cela nécessite de connaître la taille mémoire de chaque type de variables (voir chap IV "*Différents types de variables*"). Par exemple cette *taille* vaudra 6 si on veut lire des nombres réels (`Real`) ou bien 256 pour des chaînes de caractères (`String`). Le fait que la variable *taille* soit de type `Word` implique que sa valeur doit être comprise entre 0 et 65535. Par défaut, *taille*=128 octets.

Syntaxe :
```
Reset ( variable d'appel, taille ) ;
```

Pour créer un fichier qui n'existe pas ou bien pour en effacer sont contenu, on emploi `ReWrite` qui pemret d'effectuer des lectures (`File Of` et `File`) et écritures (`Text`, `File Of` et `File`). Attention, on ne peut pas lire sur un `Text` avec `ReWrite` !

Syntaxe :
```
Rewrite ( variable d'appel ) ;
```

Tout comme `Reset`, `ReWrite` permet de spécifier une *taille* aux échanges de données sur un `File` seulement (aussi bien en écriture qu'en lecture). Avec `ReWrite` c'est le cas ou le fichier n'existe pas encore alors qu'avec `Reset` c'est le cas ou il existe déjà.

Syntaxe :
```
Rewrite ( variable d'appel, taille) ;
```

Tableau des correspondances entre procédures et types de fichiers

Syntaxe	Types de fichiers associés	
	Lecture	Ecriture
`Reset(`*f*`)`	- `Text` - `File Of` - `File`	- `File Of` - `File`
`ReWrite(`*f*`)`	- `File Of` - `File`	- `Text` - `File Of` - `File`
`Reset(`*f*`, `*taille*`)`	`File`	`File`
`ReWrite(`*f*`, `*taille*`)`	`File`	`File`

Pour lire le contenu d'une ligne d'un fichier `Text` ouvert, on utilise la même instruction qui permet de lire la valeur d'une variable au clavier à savoir `ReadLn`. Sera alors lue, la ou les variable(s) correspondant au contenu de la ligne courante (celle pointée par le pointeur). Si la ou les variable(s) n'étai(en)t pas de taille suffisamment grande pour contenir toutes les données de la ligne, alors l'excédent serait perdu.

Syntaxes :

```
ReadLn ( variable d'appel, variable ) ;
ReadLn ( variable d'appel, var1, var2, ... varN) ;
```

Pour écrire sur un fichier `Text`, il suffit d'employer la commande `WriteLn`.

Syntaxes :

```
WriteLn ( variable d'appel, variable ) ;
WriteLn ( variable d'appel, var1, var2, ... varN) ;
```

Les procédures `Read` et `Write` s'utilisent respectivement de la même manière que `ReadLn` et `WriteLn` mais s'appliquent aux `File Of` aussi bien qu'aux `Text`.
Pour lire et écrire sur un `File`, il faut utiliser les procédures `BlockRead` et `BlockWrite`.

Syntaxes :

```
BlockRead ( f, variable, nbr ) ;
BlockRead ( f, variable, nbr, result ) ;
BlockWrite ( f, variable, nbr ) ;
BlockWrite ( f, variable, nbr, result ) ;
```

`BlockRead` lit sur le fichier *f* de type `File` une *variable* qui contiendra un nombre de bloc mémoire (dont la taille est définie par `Reset` `ReWrite`) égale à *nbr*. La variable facultative *result* prend pour valeur le nombre de bloc effectivement lu (car il peut y en avoir moins que prévu initialement).
`BlockWrite` écrit sur le fichier *f* de type `File` une *variable* sur un nombre de bloc mémoire égale à *nbr*. La variable facultative *result* prend pour valeur le nombre de bloc effectivement écrit (car il peut y en avoir plus à écrire que ne le permet l'initialisation par `Reset` ou `ReWrite`). Dans le cas ou cette variable *result* est ommise et qu'il est impossible d'écrire autant de blocs que voulu, il est généré une erreur !

Note : Les variables *nbr* et *result* doivent être de type `Word`.

Tableau des correspondances entre procédures et types de fichiers

Syntaxe	Types de fichiers associés
`WriteLn`	`Text`
`ReadLn`	`Text`
`Write`	- `Text` - `File Of`
`Read`	- `Text` - `File Of`
`BlockWrite`	`File`
`BlockRead`	`File`

Il est impératif de fermer les fichiers ouverts pendant un programme Turbo Pascal avant de terminer le programme sous peine de voir les données inscrites en son sein perdues.

Syntaxe :

```
Close ( variable d'appel ) ;
```

Il est possible de rappeler un fichier `Text` en cours de programme même s'il a déjà été refermé grâce à sa variable d'appel. Et alors la position courante du curseur sera à la fin du fichier. Ne fonctionne qu'en écriture et qu'avec un `Text`.

Syntaxe :

```
Append ( variable d'appel ) ;
```

La syntaxe `Append` est équivalente au bloc suivant :

```
Begin
Reset(variable d'appel) ;
Seek(variable d'appel, FileSize(variable d'appel)) ;
End ;
```

```
Program exemple12 ;
Uses crt, dos ;
Var f : text ;
nom : string ;
choix : char ;
Procedure lecture ;
   Begin
      Append (f) ;
      Reset (f) ;
      Readln (f, nom) ;
      Writeln (nom) ;
   End ;
BEGIN
   Clrscr ;
   Assign (f, 'init.txt') ;
```

```
   Rewrite (f) ;
   Write ('Entrez un nom d''utilisateur : ') ;
   Readln (nom) ;
   nom := 'Dernier utilisateur : ' + nom ;
   Writeln (f, nom) ;
   Close (f) ;
   Write ('Voulez-vous lire le fichier init.txt ? [O/N] ') ;
   Readln (choix) ;
   If (choix='O') or (choix='o') then lecture ;
END.
```

Ce programme *exemple12* illustre les principales commandes qui permettent de travailler sur des fichiers de type texte. Ici, le programme réinitialise à chaque lancement le fichier *init.txt* et y inscrit une valeur entrée par l'utilisateur (son nom, en l'occurrence). Il permet également à l'utilisateur de lire le contenu du fichier (qui ne contient qu'une seule ligne de texte).

---

3. Fonctions supplémentaires

Il est possible de déplacer à volonté le curseur en spécifiant à la procédure suivante le fichier de type `File Of` ou `File` ainsi que le numéro de l'octet (le premier à pour numéro :"0", le second : "1", le troisième : "2", etc...) où l'on veut mettre le curseur. Cela s'appelle l'accès direct à un fichier contrairement à l'accès séquentiel qui consiste à parcourir toutes les informations précédant celle qu'on cherche. Cette dernière méthode séquentielle est toutefois la plus utilisée. De plus, `Seek` n'est pas utilisable avec des `Text`.

Syntaxe :

```
Seek ( variable d'appel, position ) ;
```

```
Program exemple13 ;
Uses crt, dos ;
Var f : text ;
s : string ;
BEGIN
   Assign (f, 'c:\autoexec.bat') ;
   Reset (f) ;
   Writeln ('Affichage du contenu du fichier AUTOEXEC.BAT : ') ;
   Repeat
      Readln (f, s) ;
      Writeln (s) ;
   Until eof (f) ;
END.
```

Ce programme *exemple13* permet de lire un fichier texte en son entier et d'afficher son contenu à l'écran. La fonction `eof` permet de vérifier si le pointeur arrive en fin de fichier (elle aura alors la valeur `true`).
Il est possible de connaître la taille d'un fichier en octets lorsque celui-ci est déclaré en `file` et non plus en `text`.

Syntaxe :

```
FileSize ( variable d'appel ) ;
```

Il est possible de connaître la position du pointeur dans fichier en octets lorsque celui-ci est déclaré en `file of byte`. La fonction suivante prend pour valeur un type `longint`.

Syntaxe :

```
FilePos ( variable d'appel ) ;
```

```
Program exemple14 ;
Var f : file of byte ;
taille : longint;
BEGIN
   Assign (f, 'c:\autoexec.bat') ;
   Reset (f) ;
   taille := filesize (f) ;
   Writeln ('Taille du fichier en octets :', taille ) ;
   Writeln ('Déplacement du curseur...') ;
   Seek (f, taille div 2);
   Writeln ('Le pointeur se trouve à l''octet : ', filepos (f) ) ;
   Close (f) ;
END.
```

Le programme *exemple14* déclare le fichier *autoexec.bat* comme `file of byte` et nom plus comme `text`, puisqu'on ne désire plus écrire du texte dedans mais seulement connaître sa taille et accessoirement faire mumuse avec le pointeur.
Il est possible de savoir si lors de la lecture d'un fichier, on se trouve ou non en find de ligne ou de fichier grâce aux fonctions suivantes qui renvoient une valeur de type `boolean`.

Syntaxe :

```
Var f : Text ;
Eof ( f ) ; { Pointeur en fin de fichier. }
SeekEoLn ( f ) ; { Pointeur en fin de ligne. }
```

Autre **syntaxe :**

```
Var f : File ;
EoLn ( f ) ; { Pointeur en fin de ligne. }
SeekEof ( f ) ; { Pointeur en fin de fichier. }
Program exemple15 ;
Var f : text ;
i, j : string ;
BEGIN
   Assign (f, 'c:\autoexec.bat') ;
   Reset (f) ;
   While not seekeof (f) do
      Begin
         If seekeoln (f) then readln ;
         Read (f, j) ;
         Writeln (j) ;
      End ;
END.
```

On peut également effacer un fichier préalablement fermé.

Syntaxe:

```
Erase (f) ;
```

On peut aussi renomer un fichier.

Syntaxe :

```
Rename (f, nouveau nom ) ;
```

Il est possible de tronquer un fichier, c'est-à-dire de supprimer tout ce qui se trouve après la position courante du pointeur.

Syntaxe :

```
Truncate (f) ;
```

Il est possible d'appeler un fichier exécutable externe à partir d'un programme écrit en Pascal, et de lui assigner des paramètres grâce à la commande `Exec`. Cette commande nécessite un commentaire de compilation: `{ $M $4000,0,0 }`.

Syntaxe :

```
SwapVectors ;
Exec ( nom+chemin, paramètres ) ;
SwapVectors ;
```

Pour télécharger un programme utilisant la commande `Exec` pour utiliser le compacteur ARJ grâce à une interface ulta-simplifiée : **A.PAS** (http://cyberzoide.developpez.com/info/prog/a.pas).

CHAPITRE XI : Graphismes

Les instructions qui vont suivre nécessitent l'unité `graph`. Pour créer des graphismes, il faut tout d'abord initialiser le mode graphique de la manière suivante pour un écran VGA 640x480x16:

```
Uses Graph ;
Var VGA, VGAHi : integer ;
BEGIN
   InitGraph (VGA, VGAHi, 'c:\bp\bgi') ;
END.
```

ATTENTION : le chemin du répertoire BGI peut changer d'une machine à l'autre. Par exemple, à l'Université Claude Bernard – Lyon1, c'est : c:\turbo7\bgi.

Ici, la valeur VGA correspond au pilote graphique, VGAHi au mode graphique, on obtient en générale une résolution de 640x480 pour 16 couleurs. Il est théoriquement nécessaire d'écrire la commande suivante : `closegraph ;` en fin de programme afin de retourner en mode texte, mais on peut s'en passer.

Pour une autodétection du mode graphique maximal admisible par le système si celui-ci est inférieur à : 640x480x16 qui est la résolution maximale de l'unité Graph de Turbo Pascal 7.0 (autodétection nécessaire pour les très vieux ordinateurs et aux examens...), initialisez de la manière suivante :

```
Uses Graph ;
Var Pilote, Mode : integer ;
BEGIN
   Pilote := Detect ;
   InitGraph(Pilote, Mode, 'c:\turbo7\bgi') ;
   ...
   CloseGraph ;
END.
```

Pilote graphique

Nom	Valeur
CGA	1
EGA	9
VGA	3

Mode graphique

Résolution	Valeur
640x200	0
640x320	1
640x480	2

Ayez toujours en tête que la résolution de l'écran graphique, en Turbo Pascal, est au maximum de 640x480 pixels et de 16 couleurs (sans trifouillage).

A noter que l'origine de l'écran graphique se trouve en haut à gauche de l'écran, c'est-à-dire que le point de coordonnées (0,0) est le premier pixel de l'écran, ainsi le point à l'opposé qui est de coordonées (629, 479) se trouve en bas à droite.

Pour un affichage de meilleur résolution, fabriquez vous-même une unité spécifique (voir chapitre 14 sur les unités). Généralement, dans ce genre d'unité sensée permettre de faire plus que permis avec les unités de base, le code doit être en assembleur...

```
SetColor ( couleur ) ;
```

Instruction qui permet de sélectionner une couleur (voir tableau ci-après) qui affectera toutes les commandes graphiques. Il vous suffit d'entrer en paramètre le code ou alors le nom correspondant à la couleur de votre choix.

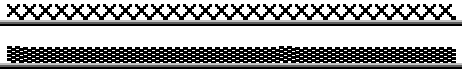
Code des couleurs
(valable aussi pour MS-DOS).

Code	Nom	Couleur	Description
0	Black		noir
1	Blue		bleu
2	Green		vert foncé
3	Cyan		cyan foncé
4	Red		rouge
5	Magenta		mauve foncé
6	Brown		marron
7	LightGray		gris clair
8	DarkGray		gris foncé
9	LightBlue		bleu flou
10	LightGreen		vert clair
11	LightCyan		cyan clair
12	LightRed		rouge clair
13	LightMagenta		mauve clair
14	Yellow		jaune
15	White		blanc

```
SetFillStyle ( style, couleur ) ;
```

Sélectionne un motif de remplissage spécifique (voir tableau ci-après) ainsi qu'une couleur parmi 16. Ces paramètres ne seront utilisés que par quelques instructions dont celle qui suit (bar). Il vous suffit d'entrer en paramètre le code ou alors le nom correspondant au motif de votre choix.

Motifs de remplissage

Code	Nom	Rendu
0	EmptyFill	couleur du fond
1	SolidFill	couleur du tracé
2	LineFill	
3	LtSlashFill	
4	SlashFill	
5	BkSlashFill	
6	LtBkSlashFill	
7	HatchFill	
8	XHatchFill	
9	InterLeaveFill	
10	WideDotFill	
11	CloseDotFill	
12	UsesFill	motif défini par le programmeur

`FloodFill (x, y,` *border* `) ;`

Rempli une surface fermée identifiée par sa couleur de bordure : *border* à partir du point de coordonnées `(x, y)`. La couleur de remplissage sera celle choisie par un `SetColor` ou un `SetFillStyle`.

`Bar (x1, y1, x2, y2 ) ;`

Construit un rectangle plein aux coordonnées indiquées. L'axe des x étant croissant de gauche à droite et celui des y croissant de haut en bas.

`Bar3d (x1, y1, x2, y2, z,` *TopOn* `) ;`

Construit un parallépipède aux coordonnées indiquées et de profondeur z. L'axe des x étant croissant de gauche à droite et celui des y croissant de haut en bas. *TopOn* est une constante signifiant que les lignes de perspectives supérieures sont activées (pour les cacher : *TopOff*).

`SetLineStyle (` *style*`, $c3,` *épaisseur* `) ;`

Sélectionne un style et une épaisseur de line utilisés par les instructions graphiques de base : line, rectangle, circle (seulement l'épaisseur). Voir les tableaux suivant pour les valeurs possibles. Il vous suffit d'entrer en paramètre le code ou alors le nom correspondant aux style et épaisseur de votre choix.

Styles de ligne

Code	Nom	Description, rendu
0	SolidLn	ligne pleine

1	DottedLn	ligne en pointillée .....
2	CenterLn	ligne mixte – – – –
3	DashedLn	ligne tiretée – – – –
4	UseBitLn	motif défini par le programmeur -- -- --

Épaisseurs de line

Code	Nom	Description
1	NormWidth	trait normal ▁▁▁
3	ThickWidth	trait épais ▁▁▁

```
Line (x1, y1, x2, y2) ;
```

Construit une ligne débutant au point de coordonnées (x1, y1) et se terminant au point de coordonnées (x2, y2).

```
LineTo (x, y) ;
```

Trace une ligne à partir de la position courante du curseur graphique, jusqu'aux coordonnées x et y spécifiées.

```
LineRel (deltaX, deltaY) ;
```

Trace une ligne à partir de la position courante du curseur graphique, jusqu'aux coordonnées calculées à partir des pas de variation delatX et deltaY spécifiés.
A noter qu'il peut y avoir équivalence entre diverses combinaisons des troix dernières instructions.

```
MoveTo (x, y) ;
```

Positionne le pointeur graphique aux coordonnées X et Y spécifiées.

```
MoveRel (deltaX, deltaY) ;
```

Déplace relativement le pointeur graphique depuis sa position courante jusqu'aux coordonnées calculées à partir des pas de variation deltaX et deltaY.

```
Rectangle (x1, y1, x2, y2) ;
```

Construit un rectangle de coin haut-gauche de coordonnées (x1, y1) et de coin bas-droite de coordonnées (x2, y2).

```
Circle (x, y, rayon ) ;
```

Construit un cercle de coordonnées et de rayon spécifiés.

```
Ellipse (x, y, angle1, angle2, axe1, axe2) ;
```

Construit une ellipse de centre (x,y) de largeur *axe1* et de hauteur *axe2*. On peut ne tracer qu'une partie de l'ellipse en spécifiant l'angle de départ *angle1* et l'angle d'arrivé *angle2* exprimés en degrés et dans le sens trigonométrique. Pour tracer une ellipse complète : *angle1*=0 et *angle2=360*.

```
Arc (x, y, angle1, angle2, rayon) ;
```

Construit un arc de cercle de centre (x,y) et de rayon *rayon*. On peut tracer un arc en spécifiant l'angle de départ *angle1* et l'angle d'arrivé *angle2* exprimés en degrés et dans le sens trigonométrique. Pour tracer un arc maximum, c'est-à-dire un cercle : *angle1*=0 et *angle2=360*.

```
Program coquille ;
Uses Graph ;
Var r, a, mode, pilote : Integer ;
BEGIN
mode := Detect ;
InitGraph(mode, pilote, 'c:\bp\bgi') ;
SetColor(14) ;
r := 0 ;
   Repeat
   Inc(r, 8) ;
   Arc(GetMaxX Div 3, GetMaxY Div 2, 0, r, 180-(r Div 2)) ;
   Until r>360 ;

ReadLn ;
CloseGraph ;
END.
```

Ce programme *coquille* dessine à l'écran une série d'arcs de cercles incomplets afin de former la coquille d'un mollusque. Ce programme montre également comment il est simple d'introduire des fonctions et des opérateurs au sein d'une ligne de commande de procédure paramétrée (`Arc();`).

```
GetPixel (a, b) ;
```

Fonction qui prend la valeur de la couleur du pixel aux coordonnées considérées.

`PutPixel (a, b,` *couleur* `) ;`

Instruction qui affiche un pixel de couleur choisie aux coordonnées spécifiées.

`GetMaxY`
Fonction qui retourne la valeur de la résolution sur l'axe des x.

`GetMaxY`
Fonction qui retourne la valeur de la résolution sur l'axe des y.

`GetMaxColor`
Fonction qui retourne la valeur de la capacité de couleurs (le nombre total de couleurs).

`GetMaxMode`
Fonction qui retourne la valeur du mode graphique.

`SetTextStyle (` *nom de la fonte ou code correspondant*`,` *orientation ou code correspondant*`,` *taille* `) ;`

Définit les paramètres d'affichage du texte qui suivra (avec la commande suivante). Il vous suffit d'entrer en paramètre le code ou alors le nom correspondant à la fonte et à l'orientation de votre choix.

Note :
Il est possible d'utiliser un très grand nombre de fontes supplémentaires sous la forme de fichiers au format **CHR** qui doivent être placés dans le répertoire **/BIN** de Turbo Pascal 7.0.

Différentes fontes de Turbo Pascal 7.0

Code	Rendu	Nom
0	DefaultFont	DefaultFont
1	TriplexFont	TriplexFont
2	SmallFont	SmallFont
3	SansSerifFont	SansSerifFont
4	GothicFont	GothicFont

Orientation de la fonte

Code	Nom	Desciption
0	HorizDir	orientation horizontale
1	VertDir	orientation verticale

`OutText (` *chaîne de caractères* `) ;`

Instruction qui permet d'afficher du texte en mode graphique.

`OutTextXY (x, y,` *chaîne de caractères* `) ;`

Instruction qui permet d'afficher du texte aux coordonnées voulues.

```
OutTextXY (succ (getmaxx) div 2, succ (getmaxy) div 4, chaîne de caractères ) ;
```

Instruction qui permet de centrer du texte au milieu de l'axe des x et au trois quart haut de l'axe des y.

```
ClearDevice ;
```

```
ClearViewPort ;
```

Effacent le contenu de l'écran graphique.

```
CloseGraph ;
```

```
RestorCrtMode ;
```

Ferment l'écran graphique pour retourner à l'affichage MS-DOS habituel.

```
SetGraphMode ( GetGraphMode ) ;
```

Retourne au mode graphique de la procédure *initialisation*.

CHAPITRE XII : Affichage à l'écran

En règle générale, les programmes dialoguent avec l'utilisateur : entrées et sorties de données respectivement avec les commandes `read` et `write`. La nécessité pratique ou la volonté de présenter une interface plus conviviale imposent l'utilisation d'instructions spécifiques : effacer une ligne seulement d'écran, changer la couleur des lettres… Ce chapitre énumère la quasi-totalité des instructions en Pascal vous permettant de faire des opérations graphiques à l'écran tout en restant en mode texte sous MS-DOS.

```
ClrScr ;
```

Pour effacer tout l'écran et placer le curseur en haut à gauche de l'écran, très utilisé au démarrage de chaque programme.

```
DelLine ;
```

Efface la ligne courante c'est-à-dire celle qui contient le curseur.

```
InsLine ;
```

Insére une ligne vide à la position courante du curseur.

```
ClrEol ;
```

Pour effacer une ligne à l'écran à partir de la position courante du curseur. Note : la position du curseur n'est pas modifiée.

Ayez toujours en tête que la résolution de l'écran texte, en Turbo Pascal, est de 80 colonnes par 25 lignes et de 16 couleurs.

```
TextBackground ( x ) ;
```

Choix d'une couleur de fond pour le texte qui sera tapé par la suite. x est le numéro (entre 0 et 15) de la couleur, il est tout à fait possible d'y mettre une variable de type `integer` a la place de x. Pour la liste des couleur, voir le chapitre Graphismes.

```
TextColor ( x ) ;
```

Choix d'une couleur pour le texte qui sera affiché par la suite.

```
TextColor ( x + blink ) ;
```

Choix d'une couleur pour le texte qui sera affiché en mode clignotant.

```
Window ( x1, y1, x2, y2 ) ;
```

Pour créer une fenêtre à l'écran. x1, y1 sont les coordonnées du caractère en haut à gauche et x2, y2 sont les positions du caractère en bas à droite. La résolution de l'écran en mode texte est de 80 colonnes par 25 lignes.

```
GotoXY ( x, y ) ;
```

Pour positionner le curseur à la position voulue dans l'écran ou dans une fenêtre Window. x et y sont respectivement le numéro de colonne et le numéro de ligne (axes des abscisses et des ordonnées).

```
WhereX ;
```

```
WhereY ;
```

Pour connaître la position courante du curseur. Ce sont des fonctions et donc renvoient de manière intrinsèque la valeur. C'est-à-dire que `WhereX` prend la valeur du numéro de colonne.

```
HightVideo ;
```

Pour sélectionner le mode haute densité des caractères. C'est-à-dire que la couleur sélectionnée pour l'affichage du texte est modifiée en son homologue plus vive dans la liste des couleurs (liste de 15 couleurs).

```
LowVideo ;
```

Au contraire, pour sélectionner le mode faible densité de la couleur des caractères. C'est-à-dire que la couleur sélectionnée pour l'affichage du texte est modifiée en son homologue moins vive dans la liste des couleurs.

```
NormVideo ;
```

Pour revenir au mode normal de couleur de texte, c'est-à-dire pour pouvoir utiliser indifféremment les couleurs vives et ternes.

```
TextMode ( x ) ;
```

Pour sélectionner un mode spécifique d'affichage du texte. x est la valeur-code du mode désiré.

CHAPITRE XIII : Caractères et chaînes de caractères

1. Chaînes de caractères

Le type `String` défini des variables "chaînes de caractères" ayant au maximum 255 signes, ces derniers appartenant à la table ASCII. Une chaîne peut en contenir moins si cela est spécifié lors de la déclaration où le nombre de signes (compris entre 1 et 255) sera mis en crochet à la suite du type `String`. Le premier cractère de la chaîne a pour indice 1, le dernier a pour indice 255 (ou moins si spécifié lors de la déclaration).

Syntaxe :

```
Var chaine : String ;
telephone : String[10] ;
```

Lorsqu'une valeur est affectée à une variable chaîne de caratères, on procède comme pour un nombre mais cette valeur doit être entre quotes. Si cette valeur contient une apostrophe, celle-ci doit être doublée dans votre code.

Syntaxe :

```
variable := valeur ;
animal := 'l''abeille' ;
```

Note très importante : le type `String` est en fait un tableau de caractères à une dimension dont l'élément d'indice zéro contient une variable de type `Char` et dont le rang dans la table ASCII correspond à la longueur de la chaîne. Il est donc possible, une chaîne de caractère étant un tableau de modifier un seul caractère de la chaîne grâce à la **syntaxe** suivante :

```
chaine[index]:=lettre;
Program Chaine;
Var nom:String;
BEGIN
   nom:='Etiévant';
   nom[2]:='Z';
   nom[0]:=Chr(4);
   WriteLn(nom);
   nom[0]:=Chr(28);
   Write(nom,'-tagada');
END.
```

L'exemple *Chaine* remplace la deuxième lettre de la variable *nom* en un "Z" majuscule, puis spécifie que la variable ne contient plus que 4 caractères. Ainsi la valeur de la variable *nom*

est devenue : *EZié*. Mais après, on dit que la variable *nom* a une longueur de 28 caractères et on s'apperçoit à l'écran que les caractères de rang supérieur à 4 ont été conservés et des espaces ont été insérés pour aller jusqu'à 28 ! Ce qui veut dire que la chaîne affichée n'est pas toujours la valeur totale de la chaîne réelle en mémoire.
Attention cependant aux chaînes déclarées de longueur spécifiée (voir Chapitre 20 sur *Types simples et structurés* exemple: `Type nom:String[20];`) dont la longueur ne doit pas dépasser celle déclarée en début de programme.

```
Concat ( s1, s2, s3, ..., sn) ;
```

Cette fonction concatène les chaînes de caractères spécifiées `s1`, `s2`, etc. en une seule et même chaîne. On peut se passer de cette fonction grâce à l'opérateur `+ : s1 + s2 + s3 + ... + sn`. Rappelons que les chaînes de caractères sont généralement définies en `string`.

Syntaxes :

```
   s := Concat ( s1, s2 ) ;
   s := s1 + s2 ;
```

```
Copy ( s, i, j ) ;
```

Fonction qui retourne de la chaîne de caractère `s`, un nombre `j` de caractères à partir de la position `i` (dans le sens de la lecture). Rappelons que `i` et `j` sont des entiers (`integer`).

```
Delete ( s, i, j ) ;
```

Procédure qui supprime dans la chaîne nommée `s`, un nombre `j` de caractères à partir de la position `i`.

```
Insert ( s1, s2, i ) ;
```

Procédure qui insert la chaîne `s1` dans la chaîne `s2` à la position `i`.

```
Pos ( s1, s2 ) ;
```

Fonction qui renvoit sous forme de variable `byte` la position de la chaîne `s1` dans la chaîne-mère `s2`. Si la chaîne `s1` en est absente, alors cette fonction renvoie 0 comme valeur.

```
Str ( x, s ) ;
```

Procédure qui convertit le nombre (`Integer` ou `Real`) `x` en chaîne de caractère de nom `s`.

```
Val ( x, s, error ) ;
```

Procédure qui convertit la chaîne de caractère de nom `s` en un nombre (`Integer` ou `Real`) `x` et renvoie un code erreur `error` (de type `integer`) qui est égale à 0 si la conversion est possible.

```
FillChar ( s, n, i ) ;
```

Procédure qui introduit `n` fois dans la chaîne `s` la valeur `i` (de type `Byte` ou `Char`).

2. Caractères seuls

Un caractère est une variable de type `Char` qui prend 1 octet (= 8 bits) en mémoire. La table ASCII est un tableau de 256 caractères numérotés de 0 à 255 où les 23 premiers sont associés à des fonction de base de MS-DOS (Suppr, End, Inser, Enter, Esc, Tab, Shift...) et tous les autres sont directement affichables (lettres, ponctuations, symboles, caractères graphiques). Dans un programme en Pascal, on peut travailler sur un caractère à partir de son numéro dans la table ASCII (avec la fonction `Chr(n°)` ou #n°) ou directement avec sa valeur affichage entre quote `''`.

Exemples :

```
espace :=' ';
lettre :=#80;
carac :=Chr(102);
```

Table ASCII :

0		32		64	@	96	`	128	Ç	160	á	192	└	224	ó
1	☺	33	!	65	A	97	a	129	ü	161	í	193	┴	225	ß
2	☻	34	"	66	B	98	b	130	é	162	ó	194	┬	226	Ô
3	♥	35	#	67	C	99	c	131	â	163	ú	195	├	227	Ò
4	♦	36	$	68	D	100	d	132	ä	164	ñ	196	─	228	õ
5	♣	37	%	69	E	101	e	133	à	165	Ñ	197	┼	229	Õ
6	♠	38	&	70	F	102	f	134	å	166	ª	198	ã	230	µ
		39	'	71	G	103	g	135	ç	167	º	199	Ã	231	Þ
		40	(	72	H	104	h	136	ê	168	¿	200	╚	232	Þ
		41	)	73	I	105	i	137	ë	169	®	201	╔	233	Ú
		42	*	74	J	106	j	138	è	170	¬	202	╩	234	Û
11	♂	43	+	75	K	107	k	139	ï	171	½	203	╦	235	Ù
12	♀	44	,	76	L	108	l	140	î	172	¼	204	╠	236	ý
13		45	-	77	M	109	m	141	ì	173	¡	205	═	237	Ý
14	♫	46	.	78	N	110	n	142	Ä	174	«	206	╬	238	¯
15	☼	47	/	79	O	111	o	143	Å	175	»	207	¤	239	´
16	►	48	0	80	P	112	p	144	É	176	░	208	ð	240	-
17	◄	49	1	81	Q	113	q	145	æ	177	▒	209	Ð	241	±
18	↕	50	2	82	R	114	r	146	Æ	178	▓	210	Ê	242	=
19	‼	51	3	83	S	115	s	147	ô	179	│	211	Ë	243	¾
20	¶	52	4	84	T	116	t	148	ö	180	┤	212	È	244	¶
21	§	53	5	85	U	117	u	149	ò	181	Á	213	ı	245	§
22	▬	54	6	86	V	118	v	150	û	182	Â	214	í	246	÷
23		55	7	87	W	119	w	151	ù	183	À	215	Î	247	¸
24	↑	56	8	88	X	120	x	152	ÿ	184	©	216	ï	248	°
25	↓	57	9	89	Y	121	y	153	Ö	185	╣	217	┘	249	¨
26	→	58	:	90	Z	122	z	154	Ü	186	║	218	┌	250	·
27	←	59	;	91	[	123	{	155	ø	187	╗	219	█	251	¹
28	∟	60	<	92	\	124	¦	156	£	188	╝	220	▄	252	³
29	↔	61	=	93	]	125	}	157	Ø	189	¢	221	¦	253	²
30	▲	62	>	94	^	126	~	158	×	190	¥	222	Ì	254	■
31	▼	63	?	95	_	127	⌂	159	ƒ	191	┐	223	▀	255	

Le type `Char` défini des variables "caractère seul" ou "lettre" ayant 1 seul signe, ce dernier appartenant à la table ASCII.

Syntaxe :

```
Var lettre : Char ;
```

Lorsqu'on donne une valeur à une variable `Char`, celle-ci doit être entre quotes. On peut aussi utiliser les fonctions `Chr` et `Ord` ou même une variable `String` dont on prend un caractère à une position déterminée.

Syntaxe :
lettre `:=` *chaine*`[`*position*`]` `;`

```
StrUpper ( s ) ;
```

Convertit une chaîne de caractères minuscules en MAJUSCULES. S'applique aux tableaux de caractères.

Syntaxe de déclaration :
```
Var s : Array[1..x] of Char
```
avec `x` *la longueur maximale de la chaîne.*

```
StrLower ( s ) ;
```

Convertit une chaîne de caractères MAJUSCULES en minuscules. S'applique aux tableaux de caractères.

Syntaxe de déclaration :
```
Var s : Array[1..x] of Char
```
avec `x` *la longueur maximale de la chaîne.*

```
UpCase ( k ) ;
```

Cette fonction ne s'applique qu'aux caractères seuls (de type `Char`) pris dans une chaîne `s`. Convertit un caractère minuscule en MAJUSCULE.

Syntaxe :
```
For i := 1 To Lenght(s) Do s[i] := UpCase(s[i]) ;
```

```
Chr ( x ) ;
```

Cette fonction renvoie un caractère `Char` correspondant au caractère d'indice `x` dans la table ASCII.

Exemple :
```
k := Chr(64) ;
```

```
Ord ( y ) ;
```

Cette fonction renvoie l'indice (en `byte`) correspondant au caractère `y` dans la table ASCII. C'est la fonction réciproque de `Chr`.

Exemple :
```
i := Ord('M') ;
```

CHAPITRE XIV : Créer ses propres unités

Lorsque vous tapez un programme en Turbo Pascal, vous utilisez nécessairement un certain nombre d'instructions (procédures ou fonctions) qui sont définies dans des unités extérieures au programme et qui stockent (souvent en langage assembleur) la marche à suivre pour l'exécution d'une certaine instruction. Même lorsque vous ne spécifiez aucune unité par la commande `Uses`, l'unité `System` est automatiquement associée au programme (inscrite dans l'exécutable compilé). Quant aux autres unités fournies avec Turbo Pascal : `Crt`, `Dos`, `Graph`, `Printer`, elles contiennent des instructions spécifiques qui ne pourront êtres appelées depuis le programme que si les unités correspondantes sont déclarées par la commande `Uses`. Le but de ce chapitre est d'apprendre à fabriquer de ses propres mains une unité qui pourra être appelée depuis un programme écrit en Turbo Pascal. Précisons qu'une unité s'écrit dans un fichier au format **PAS** depuis le logiciel de programmation Turbo Pascal. Mais une fois écrite, l'unité doit impérativement être compilées (au format **TPU**) pour être utilisable plus tard par un programme.

Un programme en Pascal débute par la déclaration (certes facultative) du nom de programme comme suit :

```
Program nom_du_programme ;
```

De manière analogue, une unité doit être (impérativement) déclarée comme telle :

```
Unit nom_de_l'unité ;
```

Ensuite, vient une partie déclarative très spéciale qui catalogue le contenu de l'unité, cette partie est très similaire à celle d'un programme puisqu'on y déclare les constantes, les variables, les procédures, fonctions... et même d'autres unités. Tout comme pour un sous-programme (procédure, fonction), les identificateurs d'une unité n'existent que dans cette dernière (les variables sont dynamiques), ils peuvent donc être les mêmes que ceux utilisés dans d'autres unités, sous-programmes ou même dans le programme principal. Cette partie déclarative obligatoire s'écrit selon la **syntaxe** suivante :

```
INTERFACE
Uses ...;
Const ...;
Var ...;
Procedure ...;
Function ...;
```

Désormais, passons aux choses sérieuses, il faut passer à la partie la plus technique, c'est-à-dire écrire le code fonctionnel : les procédures et/ou fonctions qui seront appelées par le programme principal.

Syntaxe :

```
IMPLEMENTATION
Function Tan ( a : Real ) : Real ;
   Begin
      Tan := Sin(a)/Cos(a) ;
   End ;
```

Et la touche finale : un bloc `Begin ... End.` Qui peut très bien ne rien contenir.

Syntaxe :

```
BEGIN
END.
```

A noter que ses quatre parties doivent toutes impérativement apparaître dans le code.

Vous pouvez télécharger l'unité **TANGT.PAS**

```
Unit Tangt ;

INTERFACE
Var a : Real ;
Function Tan ( a : Real ) : Real ;

IMPLEMENTATION
Function Tan ( a : Real ) : Real ;
    Begin
        Tan := Sin(a)/Cos(a) ;
    End ;

BEGIN
END.
```

(http://cyberzoide.developpez.com/info/prog/tangt.pas)

et le programme **TAN.PAS**

```
Uses Tang ;

Var x : Real ;

BEGIN
    Write('Entrez un angle en radian : ') ;
    ReadLn(x) ;
    WriteLn('Voici sa tangeante : ',tan(2)) ;
END.
```

(http://cyberzoide.developpez.com/info/prog/tan.pas)

CHAPITRE XV : Booléens et tables de vérité

Les booléens ont été inventés par Monsieur Boole dans le but de créer des fonctions de base qui, associées les unes aux autres, pourraient donner d'autres fonctions beaucoup plus complexes.

Les booléens (`boolean` en anglais et en Pascal) ne peuvent prendre que deux valeurs possibles : vrai (`true`) ou faux (`false`) et sont codés en 0 ou 1. Ces valeurs sont analogues aux états possibles d'un interrupteur : ouvert ou fermé, d'une lampe : allumée ou éteinte, d'un composant électronique du type transistor : conducteur ou isolant.

En bref, les booléens ne sont utiles que pour connaître un état : vrai ou faux et en général pour caractériser si une condition est vraie ou fausse. Vous les utilisez déjà sans toujours le savoir dans les blocs IF : si telle condition est vraie, alors...

Boole inventa une algèbre qui porte son nom : l'*algèbre de Boole*. C'est cette dernière qui nous permet de faire des opérations sur les booléens grâce à des opérateurs (voir chapitre 2 sur les opérateurs) : NOT (non), OR (ou), AND (et), XOR (ou exclusif), NAND (inverse de et), NOR (inverse de ou). En Turbo Pascal 7.0 n'existent que les opérateurs `NOT`, `OR`, `AND` et `XOR`. Qui suffisent (en les combinant) à retrouver les autres. Ainsi, NAND=NOT(AND) et NOR=NOT(OR). Les tables de vérité de opérateurs logiques disponibles sur Turbo Pascal 7.0 sont en bas de page.

Comme toute variable, il faut déclarer une variable booléenne dans la partie déclarative en en-tête du programme.

Syntaxe :

```
Var nomdevariable : boolean ;
```

Pour donner une valeur à une variable booléenne, on procède comme pour tout autre type de variable, à l'aide du commutateur `:=` .

Syntaxes :

```
nomdevariable := true ;
nomdevariable := false ;
```

Note : par défaut, une variable booléenne est FALSE (tout comme une variable INTEGER est égale à zéro lors du lancement du programme).

On peut bien évidemment utiliser une structure IF.

Syntaxe :

```
If condition then nomdevariablebooléenne := true ;
```

Il existe une autre façon de donner la valeur true (ou false) à une telle variable, bien plus simple qu'une structure IF.

Syntaxe :

```
nomdevariable := condition ;
```

Exemple :

```
test := (x>100) and (u='coucou') ;
```

Dans une structure conditionnelle (`If`, `Until`, `While`), on peut avantageusement utiliser des booléens sans spécifier sa valeur qui sera alors prise par défaut égale à true. C'est-à-dire que si on ne précise pas la valeur d'une variable booléenne dans une structure IF, par exemple, le compilateur Turbo Pascal se dira systématiquement : si variable est true, alors faire… Il devient donc inutile de spécifier la valeur de la variable dans ce cas là.

Syntaxes :

```
If nomdevariablebooléenne then instruction ;

Repeat
    instructions
Until nomdevariable ;

While nomdevariable do instruction end ;
```

Nous avons vu plus haut que les opérateurs spécifiques aux booléens (NOT, OR, AND, XOR) pouvaient se composer pour donner des expressions plus complexes. Il est bien entendu possible d'introduire dans ces expressions le opérateurs relationnels (=, <, >, <=, >=, <>) et plus généralement tous les autres opérateurs disponibles en Turbo Pascal. Pour pouvez même utiliser directement des expressions qu'elles soient mathématiques ou non.

Exemples :

```
test := (lenght(u)<= 20) or ((sin(a)*pi) < x) ;
```

Ici, la variable booléenne *test* devient vraie si la longueur de la chaîne **u** n'excède pas 20 caractères ou si le sinus de l'angle **a** multiplié par la valeur de pi est strictement inférieur à la valeur de **x**.

```
If (a=0) or ((b=0) and (c=0)) then writeln('La lampe est allumée')
;
```

Ici, écriture à l'écran d'un message si **a** est nul ou si **b** et **c** sont simultanément nuls.
Note : si vous faites afficher à l'écran la valeur d'une variable booléenne, il s'affichera FALSE ou TRUE (selon sa valeur effective) en caractères majuscules.

Tables de vérité des opérateurs logiques :

NOT

X	NOT X
false	true
true	false

AND

X	Y	X AND Y
false	false	false
false	true	false
true	false	false
true	true	true

OR

X	Y	X OR Y
false	false	false
false	true	true
true	false	true
true	true	true

XOR

X	Y	X XOR Y
false	false	false
false	true	true
true	false	true
true	true	false

CHAPITRE XVI : Gestion des dates et heures

Le système de l'ordinateur travaille avec l'horloge à quartz qui donne le *tempo* de calcul. Cette horloge possède sa propre date et heure qu'il est possible d'afficher ou de modifier. A partir du prompt MS-DOS, il suffit d'utiliser les commandes **date** ou **time**, mais en Turbo Pascal, c'est un peut plus délicat. En effet, en pascal, il est nécessaire de déclarer un grand nombre de variables qu'il faut formater avant l'affichage.

Note : toutes les instructions qui suivent nécessites l'unité `dos`.

`GetDate (`*an, mois, jour, joursem*`) ;`

Pour obtenir la date courante du système, avec **an** qui est le numéro de l'année (compris entre 1980 et 2099), **mois** : le numéro du mois (1 à 12), **jour** : le numéro du jour dans le mois (1 à 31) et **joursem** : le numéro du jour dans la semaine (0 à 6, le zéro correspondant au samedi). Ces variables sont déclarées en `word` qui est un ensemble d'entier supérieur au type `integer` puisqu'il contient les entiers positifs de 0 à 65535.

`SetDate (`*an, mois, jour*`);`

Pour changer la date du système. Ici, les variables obéissent aux mêmes conditions décrites précédemment. Si une date entrée est invalide, alors elle ne sera pas prise en compte et la date restera inchangée.

`GetTime (`*heure, minute, seconde, cent*`);`

Pour obtenir l'heure courante avec **heure** qui est le numéro de l'heure (comprise entre 0 et 23), **minute** : numéro de la minute (0 à 59), **seconde** : numéro de la seconde (0 à 59) et **cent** : les centièmes de seconde (0 à 99).

`SetTime (`*heure, minute, seconde, cent*`);`

Pour changer l'heure système. Les variables obéissant aux mêmes conditions décrites plus haut. Si une heure entrée est invalide, alors elle ne sera pas prise en compte et l'heure courante n'en sera aucunement affectée.

Vous pouvez télécharger le programme annoté et explicatif **DATE.PAS**

```
{ * DECLARATION DE L'UNITE UTILISEE : }

uses dos;  {unité utilisée}


{ * DECLARATION DES CONSTANTES UTILISEES : }

const jours : array [0..6] of string[8] =

('dimanche','lundi','mardi','mercredi','jeudi','vendredi','samedi');
    {construction d'un tableau contenant les jours de la semaine}
      mois : array [0..11] of string[9] =
```

```
('décembre','janvier','février','mars','avril','mai','juin','juillet'
,
      'août','septembre','octobre','novembre');
    {construction d'un tableau contenant les mois de l'année}


{ * DECLARATION DES VARIABLES UTILISEES : }

var a,m,j,jour,heure,min,sec,cent:word;
{déclaration des variables en WORD (nombres entiers compris dans
l'ensemble [0..65535])
 a     : numéro de l'année [1980..2099]
 m     : numéro du mois [1..12]
 j     : numéro du jour dans le mois [1..31]
 jour  : nom du jour dans la semaine [0..6] le zéro correspond au
samedi
 heure : numéro de l'heure [0..23]
 min   : numéro de la minute [0..59]
 sec   : numéro de la seconde [0..59]
 cent  : numéro du centième de seconde [0..99]}


{ * DECLARATION DE LA FONCTION UTILISEE : }

function format(w:word):string; {fonction, renvoie en valeur
intrinsèque}
var s:string;                   {variable chaîne locale}
begin                           {d,but de la fonction}
  str(w:0,s);                     {convertie une chaîne STRING nombre
  en WORD}
  if length(s)=1 then s:='0'+s;
  {si la chaîne possède un seul caractère, alors on lui rajoute un
  zéro devant}
  format:=s;                      {la fonction FORMAT prend la valeur
  de S}
end;                            {fin de la fonction}


BEGIN                           {d,but du programme principal}
  getdate(a,m,j,jour);            {lecture de la date système}
  write('Nous sommes le ',jours[jour],' ',j,' ',mois[m],' ',a);
                                  {affichage de la date correctement
  format,e}
  gettime(heure,min,sec,cent);    {lecture de l'heure système}
  writeln(' et il est
  ',format(heure),':',format(min),':',format(sec),'.',
  format(cent));                  {affichage de l'heure correctement
  format,e}
END.                            {fin du programme}
```

(http://cyberzoide.developpez.com/info/prog/date.pas)

`GetFTime (f, heure);`

Pour obtenir la date de dernière modification de fichier (qui est affichée dans l'Explorateur de Windows 95). Ici, **f** est une variable d'appel de fichier (`file`, `file of ...` ou `text`) et **heure** est une variable de type `longint`(qui est un ensemble de nombres entiers relatifs compris entre -2147483648 et 2147483647).

`SetFTime (f, heure);`

Vous l'aurez deviné, c'est la réciproque de l'instruction `GetFTime`.

`UnpackTime (heure, dt);`

Une information obtenue avec l'instruction `GetFTime` est illisible sans avoir été décodée avec cette l'instruction où **heure** est la même variable que précédemment et **dt** est une variable de type `datetime`. Ensuite pour pouvoir utiliser les informations contenues dans **dt**, il faut les sortir une par une : **dt.hour** (séparés par un point, comme pour les types) représente l'heure, **dt.min** : les minutes, et **dt.sec** : les secondes.

`PackTime (dt, heure);`

Cette instruction est l'inverse de la précédente.

Vous pouvez télécharger le programme annoté **FDATE.PAS**

```
uses dos;              {unité utilisée}

var f:file;            {F est déclarée en FILE : "fichier"}
    heure:longint;     {HEURE est un nombre de type LONGINT}
    dt:datetime;       {DT est du type DATETIME}

function format(w:word):string; {fonction, renvoie en valeur
intrinsŠque}
var s:string;                   {variable chaîne locale}
begin                           {d,but de la fonction}
  str(w:0,s);                     {convertit une chaîne STRING nombre
  en WORD}
  if length(s)=1 then s:='0'+s;
  {si la chaîne possède un seul caractère, alors on lui rajoute un
  zéro devant}
  format:=s;                      {la fonction FORMAT prend la valeur
  de S}
end;                            {fin de la fonction}


BEGIN                  {d,but du programme}
  assign(f,'c:\autoexec.bat'); {la variable F est assignée au fichier
  indiqu,}
  getftime(f,heure);
  {lecture de l'heure de dernière modification du fichier,
  l'information
  correspondante est consignée dans la variable HEURE}
  writeln(heure);       {écriture … l'écran de la valeur de cette
  variable}
  unpacktime(heure,dt); {conversion de l'information vers DT}
  writeln(format(dt.hour),':',format(dt.min),':',format(dt.sec));
  {affichage correct format de l'heure, des minutes et des secondes}
END.                   {fin du programme}
```

(http://cyberzoide.developpez.com/info/prog/fdate.pas)

CHAPITRE XVII : Commandes systèmes

Tous comme sous Dos ou Windows, il est quelquefois nécessaire pour certains types de programmes (cryptage, installation, setup, etc.) de créer des répertoires, de dépacer des fichiers... Turbo Pascal 7.0 propose un certains nombre de commandes permettant ces manipulations. Certaines d'entre elles seront discutées au cours de ce chapitre. Les instructions suivantes nécessitent l'unité `Dos`

1. Répertoires et lecteurs

```
MkDir ( s ) ;
```

Procédure qui crée le sous-répertoire `s` qui est une variable de type `string` dans le lecteur et répertoire courant.

```
RmDir ( s ) ;
```

Procédure qui supprime le sous-répertoire `s` qui est une variable de type `string` dans le lecteur et répertoire courant.

```
ChDir ( s ) ;
```

Procédure qui change de répertoire courant pour aller dans le répertoire `s` avec `s` une variable `string`.

```
GetDir ( b, s ) ;
```

Procédure qui renvoit le répertoire courant dans la variable `s` de type `string` du lecteur lui-même spécifié en `byte`.

Code des lecteurs

Valeur	Lecteur
0	courant
1	A:
2	B:
3	C:

```
Program exemple16 ;
Uses dos ;
Var s,r,t:String ;
i:integer ;
BEGIN
```

```
      GetDir(0,s) ;
      Writeln('Lecteur et répertoire courant: ',s) ;
      {$I-}
      Write('Aller dans quel répertoire ? -> ') ;
      ReadLn(r) ;
      For i := 1 To Length(r) Do r[i] := UpCase(r[i]) ;
      ChDir(r) ;
      If IOResult <> 0 Then
         Begin
            Write(r,' n''existe pas, voulez-vous le créer [o/n] ? -> ') ;
            Readln(s) ;
            If (s='o') Or (s='O') Then
               Begin
                  MkDir(r) ;
                  WriteLn('Création de ',r) ;
               End ;
         End
         Else Writeln('Ok : ',r,' existe !') ;
      ReadLn ;
      ChDir(s) ;
   END.
```

Ce programme *exemple16* affiche le répertoire courant du disque courant et propose à l'utlisateur de changer de répertoire. Si le répertoire choisi n'existe pas, il le crée.

```
DiskFree ( b ) ;
```

Fonction qui retourne dans une variable de type `longint` la taille libre en octets du disque se trouvant dans le lecteur `b` spécifié avec `b` une variable de type `byte`.

```
DiskSize ( b ) ;
```

Fonction qui retourne dans une variable de type `longint` la capacité totale exprimée en octets du disque spécifié `b`, avec `b` de type `byte`.

```
   Program exemple17 ;
   Uses dos ;
   BEGIN
      Writeln(DiskSize(0), ' octets') ;
      Writeln(DiskSize(0) div 1024, ' kilo octets') ;
      Writeln(DiskSize(0) div 1048576, ' méga octets') ;
      Writeln(DiskSize(0) div 1073741824, ' giga octets') ;
   END.
```

Ce programme *exemple17* affiche à l'écran la capacité totale du disque dur sous différents formats, en utilisant la propriété binaire du mode de stockage.

2. Environnement MS-DOS

`DosVersion ;`

Fonction qui retourne le numéro de version du système d'exploitation MS-DOS présent dans le système sous la forme d'une variable de type `word`.

```
Program exemple18 ;
Uses dos ;
Var version : word ;
BEGIN
   version := DosVersion ;
   WriteLn('MS-DOS version : ', Lo(version), '.', Hi(version)) ;
END.
```

Ce programme *exemple18* affiche le numéro de la version résidante de MS-DOS correctement formatée avec les fonctions `Lo` et `Hi` qui renvoient respectivement le byte inférieur et suppérieur de l'information contenue dans la variable `version`.

`DosExitCode ;`

Fonction qui renvoit le code sortie d'un sous-processus sous la forme d'une variable de type `word`.

Valeurs de DosExitCode

Valeur	Description
0	Normal
1	Ctrl+C
2	Device error
3	Procédure Keep (TSR)

`EnvCount ;`

Fonction qui renvoit sous la forme d'une variable de type `integer` le nombre de chaînes de caractères contenues dans l'environnement MS-DOS.

`EnvStr ( i ) ;`

Fonction qui renvoit sous la forme d'une variable de type `string` la chaînes de caractères contenue dans l'environnement MS-DOS à la position dans l'index spécifiée par la variable `i` de type `integer`.

```
Program exemple19 ;
Uses dos ;
Var i : integer ;
BEGIN
   For i := 1 To EnvCount Do WriteLn(EnvStr(i)) ;
END.
```

Ce programme *exemple19* affiche l'intégralité des chaînes d'environnement MS-DOS à l'aide d'une boucle.

```
GetCBreak ( break ) ;
```

Procédure qui permet de connaître la valeur (vrai ou fausse) de la variable `break` de MS-DOS. Avec `break` de type `boolean`.

```
SetCBreak ( break ) ;
```

Procédure qui permet de donner la valeur vrai ou fausse à la variable `break` de MS-DOS. Avec `break` de type `boolean`.

---

3. Fichiers

```
SetFAttr ( f, attr ) ;
```

Procédure qui attribut au fichier `f` de type `file` la variable `attr` de type `word`.

```
GetFAttr ( f, attr ) ;
```

Procédure qui renvoit dans la variable `attr` de type `word`, la valeur de l'attribut du fichier `f` déclaré en `file`.

Code des attributs de fichiers

Valeur	Nom	Description
$01	ReadOnly	Lecture seule
$02	Hidden	Caché
$04	SysFile	Système
$08	VolumeID	VolumeID
$10	Directory	Répertoire
$20	Archive	Archive
$3F	AnyFile	tous

Code des DosError

Valeur	Description
2	Fichier non trouvé
3	Répertoire non trouvé
5	Accès refusé
6	Procédure non valable
8	Pas assez de mémoire
10	Environnement non valable
11	Format non valable
18	Plus de fichiers

```
FExpand ( fichier ) ;
```

Fonction qui rajoute le chemin d'accès du fichier spécifié dans le nom de ce fichier. La variable `fichier` doit être de type `PathStr` mais vous pouvez entrer directement une chaîne de caractère.

```
FSplit ( fichier, dir, name, ext ) ;
```

Procédure qui sépare un nom de fichier (`fichier`) de type `PathStr`en ses trois composantes : chemin (`dir`) de type `DirStr`, nom (`name`) de type `NameStr`, son extension (`ext`) de type `ExtStr`.

```
Program exemple20 ;
Uses dos ;
Var P : PathStr ;
D : DirStr ;
N : NameStr ;
E : ExtStr ;
BEGIN
   Write('Entrez un nom de fichier : ') ;
   Readln(P) ;
   FSplit(P, D, N, E) ;
   Writeln('Son chemin : ',D,'', son nom : ',N, '' et son extension :
   ',E,'.') ;
END.
```

Ce programme *exemple20* demande à l'utilisateur d'entrer un nom de ficher avec son chemin, et il affiche séparément toutes les informations : le chemin, le nom et l'extension.

```
FileSize ( f ) ;
```

Fonction qui renvoie sous la forme d'une variable `longint` la taille du fichier `f` déclaré en `file`.

Il est possible de rechercher des fichiers selon certains critères de nom, d'attribut, d'extension avec les commandes `FindFirst` et `FindNext`. Regrouper ces commandes permet de simuler aisément la commande **DIR** de MS-DOS ou l'option *RECHERCHER* de Windows.

Syntaxe :

```
Program exemple21 ;
Uses dos ;
Var fichier : SearchRec ;
BEGIN
   FindFirst('*.PAS', Archive, fichier) ;
   While DosError = 0 Do
      Begin
         WriteLn(fichier.Name) ;
         FindNext(fichier) ;
      End ;
END.
```

Ce programme *exemple21* permet de rechercher et d'afficher le nom de tous les fichiers correspondants aux critères de recherche, c'est-à-dire les fichiers d'extension **PAS** et d'attribut **archive**.

Voir chapitre 10 pour l'utilisation des fichiers externes, voir aussi chapitre 16 pour la gestion des dates et heures.

4. Mémoire vive

`MemAvail ;`

Cette fonction retourne la mémoire totale libre en octets.

`MaxAvail ;`

Cette fonction retourne la longueur en octets du bloc contigu le plus grand de la mémoire vive. Très utile pour connaître la taille allouable à un pointeur en cours d'exécution.

CHAPITRE XVIII : Pseudo-hasard

Il est quelquefois nécessaire d'avoir recours à l'utilisation de valeurs de variables (scalaire ou chaîne) qui soient le fruit du hasard. Mais l'ordinateur n'est pas capable de créer du vrai hasard. Il peut cependant fournir des données dites *aléatoires*, c'est-à-dire issues de calculs qui utilisent un grand nombre de paramètres eux-mêmes issus de l'horloge interne. On appelle cela un pseudo-hasard car il est très difficile de déceler de l'ordre et des cycles dans la génération de valeurs aléatoires. Ainsi, on admettra que Turbo Pascal 7.0 offre la possibilité de générer des nombres aléatoires.

Avant l'utilisation des fonctions qui vont suivre, il faut initialiser le générateur aléatoire (tout comme il faut initialiser la carte graphique pour faire des dessins) avec la procédure `Randomize`. Cette initialisation est indispensable pour être sûr d'obtenir un relativement "bon hasard" bien que ce ne soit pas obligatoire.

Syntaxe :

```
Randomize ;
```

On peut générer un nombre réel aléatoire compris entre 0 et 1 grâce à la fonction `Random`.

Syntaxe :

```
X := Random ;
```

On peut générer un nombre entier aléatoire compris entre 0 et Y-1 grâce à la fonction `Random(Y)`.

Syntaxe :

```
X := Random(Y) ;
```

```
Program exemple22;
Uses crt ;
Const max = 100 ;
Var test : Boolean ;
x, y : Integer ;
BEGIN
   ClrScr ;
   Randomize ;
   y := Random(max) ;
   Repeat
      Write('Entrez un nombre : ') ;
      ReadLn(x) ;
      test := (x=y) ;
      If test Then WriteLn('Ok, en plein dans le mille.')
      Else If x>y Then WriteLn('Trop grand.')
      Else writeln('Trop petit.') ;
   Until test ;
   ReadLn ;
END.
```

Dans ce programme *exemple22* (programme *Chance* typique des calculettes), on a génération d'un nombre aléatoire compris entre 0 et une borne `max` définie comme constante dans la

partie déclarative du programme, ici, on prendra la valeur 100. Le programme saisie un nombre entré par l'utilisateur, effectue un test et donne la valeur `true` à une variable `boolean` nommée `test` si le nombre du joueur est correct, sinon, affiche les messages d'erreurs correspondants. Le programme fonctionne à l'aide d'une boucle conditionnelle `Repeat...Until`.

CHAPITRE XIX : Paramètres et TSR

Lorsqu'on distribue un programme créé sur Turbo Pascal 7.0, il est de bon augure de pouvoir passer des paramètres en ligne de commande lors du lancement sous MS-DOS, de pouvoir rendre résident en mémoire le dit programme (Terminate Stay Resident).

`Keep ( error ) ;`

Procédure qui rend résident en mémoir le programme (TSR). Et renvoit sous la forme d'une variable `error` de type `word` le code erreur de sortie. Nécessite l'unité `Dos`.

Syntaxe :

```
Keep(0) ;
```

`ParamCount`

Fonction qui renvoit le nombre de paramètres passés en ligne de comande lors du lancement du programme sous une valeur de type `word`. Présent dans l'unité `System` (inutile à spécifier).

Syntaxe :

```
i := ParamCount ;
```

`ParamStr(i)`

Fonction qui renvoit la chaîne passée en commande selon sa place `i` (`word`) dans l'index. Présent dans l'unité `System`.

Syntaxe :

```
s := ParamStr(i) ;
```

```
Program exemple23 ;
Uses dos ;
Var i : word ;
f : text ;
s : string ;
Procedure acces ;
Begin
   WriteLn('Ok') ;
   ...
End ;
BEGIN
   Assign(f,'password.dat') ;
   Reset(f) ;
   ReadLn(f,s) ;
   If ParamStr(1) = s Then acces
   Else WriteLn('Acces denied.') ;
END.
```

Ce programme *exemple23* est protégé d'accès. C'est-à-dire que seul un code passé en ligne de commande (et contenu dans un fichier) permet à l'utilisateur de faire tourner le programme. On aurait put créer des procédures paramétrées dont les paramètres soient ceux passés en ligne de commande, comme pour un compacteur en mode MS-DOS, par exemple.

CHAPITRE XX : Types

Il est possible de créer de nouveaux types de variables sur Turbo Pascal 7.0. Il y a encore quelques décennies, un "bon" programmeur était celui qui savait optimiser la place en mémoire que prenait son programme, et donc la "lourdeur" des types de variables qu'il utilisait. Par conséquent, il cherchait toujours à n'utiliser que les types les moins gourmands en mémoire. Par exemple, au lieu d'utiliser un `integer` pour un champs de base de donnée destiné à l'âge, il utilisait un `byte` (1 octet contre 2). Voir chapitre 4 sur les types de variables. Il est donc intéressant de pouvoir manipuler, par exemple, des chaînes de caractères de seulement 20 signes : `string[20]` (au lieu de 255 pour `string`, ça tient moins de place). Les variables de types simples comme celles de type complexe peuvent être passées en paramètre à une procédure ou fonction que ce soit par l'identificateur principal ou par ses champs.

1. Type simple

On déclare les nouveaux types simple de variable dans la partie déclarative du programme et avant la déclaration des variables utilisant ce nouveau type.

Syntaxe :

```
Type nom_du_type = nouveau_type ;
```

Exemples :

```
Type nom = string[20] ;
Type entier = integer ;
Type tableau = array [1..100] of byte ;
Program exemple24 ;
Type chaine = string[20] ;
Var nom : chaine ;
age : byte ;
BEGIN
   Write('Entrez votre nom : ') ;
   ReadLn(nom) ;
   Write('Entrez votre âge : ') ;
   ReadLn(age) ;
   WriteLn('Votre nom est : ',nom,' et votre âge : ',age) ;
END.
```

Ce programme *exemple 24* utilise un nouveau type appelé *chaîne* qui sert à déclarer la variable *nom*.

2. Type structuré (encore appelé enregistrement)

On peut être amené à utiliser des types structurés car dans une seule variable on peut réussir à caser des sous-variables nommées **champs**. Comme nous l'avons vu dans le chapitre 16 avec le type `datetime`.

Syntaxe :

```
Type nom_du_type = Record
   sous_type1 : nouveau_type1 ;
   sous_type2 : nouveau_type2 ;
   sous_type3 : nouveau_type3 ;
   End ;
```

Note : les champs sont placés dans un bloc `Record ... End ;` et un sous-type peut lui-même être de type `Record`.

Syntaxe :

```
Type nom_du_type = Record
   sous_type1 : nouveau_type1 ;
   sous_type2 = Record ;
      sous_type2_1 : nouveau_type2 ;
      sous_type2_2 : nouveau_type3 ;
      sous_type2_3 : nouveau_type4 ;
      End ;
   End ;
```

Note : une constante ne peut pas être de type complexe (`Array`, `Record`...) mais seulement de type simple.

Note : on ne peut pas afficher le contenu d'une variable structurées sans passer par une syntaxe spécifiant le champs dont on veut connaître la valeur.

Note : les champs d'une variable de type structuré peuvent être de tout type (même tableau) sauf de type fichier (`Text`, `File`, `File OF x`).

```
Program exemple25a ;
Type formulaire = Record
   nom : string[20] ;
   age : byte ;
   sexe : char ;
   nb_enfants : 0..15 ;
   End ;
Var personne : formulaire ;
BEGIN
With personne Do
   Begin
      nom := 'Etiévant' ;
      age := 18 ;
```

```
      sexe := 'M' ;
      nb_enfants := 3 ;
   End ;
END.

Program exemple25b ;
Type formulaire = Record
   nom : string[20] ;
   age : byte ;
   sexe : char ;
   nb_enfants : 0..15 ;
End ;
Var personne : formulaire ;
BEGIN
   personne.nom := 'Etiévant' ;
   personne.age := 18 ;
   personne.sexe := 'M' ;
   personne.nb_enfants := 3 ;
END.
```

Ces programmes *exemple25* (a et b) sont absolument identiques. Ils utilisent tout deux une variable *personne* de type *formulaire* qui comprend trois champs : *nom*, *age* et *sexe*. L'utilisation de ces champs se fait ainsi : **variable[point]champ** (*exemple25b*). Lorsqu'on les utilise à la chaîne (*exemple25a*), on peut faire appel à `With`.

```
Program exemple25c ;
Type date = Record
   jour : 1..31 ;
   mois : 1..12 ;
   an : 1900..2000 ;
   End ;
Type formulaire = Record
   nom : string[20] ;
   date_naissance : date ;
   End ;
Var personne : formulaire ;
BEGIN
With personne Do
   Begin
      nom := 'Etiévant' ;
      With date_naissance Do
         Begin
            jour := 21 ;
            mois := 10 ;
            an := 1980 ;
         End ;
   End ;
END.
```

```
Program exemple25d ;
Type formulaire = Record
   nom : string[20] ;
   date_naissance : Record
      jour : 1..31 ;
      mois : 1..12 ;
      an : 1900..2000 ;
      End ;
   End ;
Var personne : formulaire ;
BEGIN
With personne Do
   Begin
      nom := 'Etiévant' ;
      With date_naissance Do
         Begin
            jour := 21 ;
            mois := 10 ;
            an := 1980 ;
         End ;
   End ;
END.
```

La aussi, les programmes *exemple25* (c et d) sont absolument identiques. Ils utilisent tout deux une variable *personne* de type *formulaire* qui comprend deux champs : *nom*, et *date_naissance* qui elle-même est de type structuré et comprenant les variables *jour*, *mois* et *an*.

3. Type intervalle

Les types intervalles très utilisés ici ont rigoureusement les mêmes propriétés que ceux dont ils sont tirés. Ils peuvent être de type nombre entier (`Byte`, `Integer`, `ShortInt`, `LongInt`, `Long`, `Word`) ou caractères (`Char`). Un type intervalle est forcément de type entier ou est compatible avec un type entier. Certaines fonctions sont réservées aux types intervalle, comme par exemple renvoyer le successeur dans l'intervalle considéré. Sachant qu'un intervalle est forcément ordonné et continu.

Syntaxe :

```
Type mon_type = borneinf..bornesup ;
```

On doit obligatoirement avoir :

- *borneinf* et *bornesup* de type entier ou caractère
- *borneinf* <= *bornesup*
-

Exemples :

```
Type bit = 0..1 ;
Type alpha = 'A'..'Z' ;
Type cent = 1..100 ;
```

Toutes ces instructions : `Inc()` (incrémentation de la variable passée en paramètre), `Dec()` (décrémentation de la variable passée en paramètre), `Succ()` (renvoie le successeur de la variable passée en paramètre), `Pred()` (renvoie le prédécesseur de la variable passée en paramètre), `Ord()` (renvoie l'index de la variable dans l'intervalle auquel elle appartient) s'appliquent aux seuls types intervalles qu'ils soient de type nombre entier ou caractère et énumérés. Par exemple, la boucle `For` et la condition `Case Of` n'acceptent que des variables de type intervalles (dont ont peut tiré un successeur pour l'itération...).

```
Program exemple31a ;
Const Max=100 ;
Type intervalle=1..Max ;
Var x : intervalle ;
BEGIN
x:=1 ;
{...}
If Not(Succ(x)=Max) Then Inc(x) ;
{...}
END.
```

Cet *exemple31a* utilise quelques fonctions spécifiques aux types intervalles. L'exemple suivant montre qu'on aurait pu se passer de déclarer un nouveau type en le spécifiant directement dans la syntaxe `Var`.

```
Program exemple31b ;
Const Max=100 ;
Var x : 1..Max ;
BEGIN
x:=1 ;
{...}
If Not(Succ(x)=Max) Then Inc(x) ;
{...}
END.
```

4. Type énuméré

Un type énuméré est un type dont les variables associées n'auront qu'un nombre très limité de valeur (au maximum 256 différentes possibles). Un type énuméré sert de définition à un ensemble mathématique par l'intermédiaire de la syntaxe `Set Of` dont ce n'est pas le sujet ici, voir chapitre Ensemble. La définition d'un type énuméré consiste à déclarer une liste de valeurs possibles (256 au maximum) associées à un type, c'est-à-dire qu'une variable de type énuméré aura l'une et une seule de ces valeurs et pas une autre.

```
Program exemple32 ;
Type jours=(dim, lun, mar, mer, jeu, ven, sam) ;
Var today : jours ;
BEGIN
today := mar ;
today:=Succ(today) ;
```

```
Inc(today,2) ;
Case today Of
   dim : WriteLn('Dimanche') ;
   lun : WriteLn('Lundi') ;
   mar : WriteLn('Mardi') ;
   mer : WriteLn('Mercredi') ;
   jeu : WriteLn('Jeudi') ;
   ven : WriteLn('Vendredi') ;
   sam : WriteLn('Samedi') ;
   Else WriteLn('autre, ',Ord(today)) ;
End;
END.
```

Les instructions propres au type intervalle sont valables également pour le type énuméré. Dans cet *exemple32*, il est déclaré un type *jours* de type énuméré composé de 7 éléments représentant les jours de la semaine. Remarquez que les éléments sont uniquement des identifiants qui n'on aucune valeur intrinsèque, on peut tout juste les repérer par leur index (l'ordre dans lequel ils apparaissent dans la déclaration, où le premier élément à le numéro 0 et le dernier : n-1). Tout d'abord une affectation à l'aide de l' opérateur habituel `:=` vers la variable *today*. Puis on lui affecte sont successeur dans la déclaration. Ensuite, on l'incrémente de 2 c'est-à-dire qu'on le remplace par son sur-suivant. Et selon, sa valeur, on affiche à l'écran le jour de la semaine correspondant si cela est possible.

Remarque : La fonction `Chr()` réciproque de `Ord()` dans le cas de la table ASCII ne s'applique pas aux types intervalles et énumérés.
La partie déclarative de cet *exemple32* :

```
Type jours=(dim, lun, mar, mer, jeu, ven, sam) ;
Var today : jours ;
```

aurait très bien pu être racourcie en :

```
Var today : (dim, lun, mar, mer, jeu, ven, sam) ;
```

Note : Il est impossible d'utiliser les procédures `Write(Ln)` et `Read(Ln)` avec les variables de type énuméré.

```
Program exemple33 ;
Var color:(red, yellow, green, black, blue) ;
BEGIN
For color:=red To blue Do WriteLn('*') ;
END.
```

Cet *exemple33* montre que l'instruction de boucle `For` marche aussi bien pour les types intervalles qu'énumérés.

```
Program exemple34 ;
Var color:(red, yellow, green, black, blue) ;
BEGIN
color:=green ;
Case color Of
```

```
    red : WriteLn('Rouge') ;
    yellow : WriteLn('Jaune') ;
    green : WriteLn('Vert') ;
    black : WriteLn('Noir') ;
    blue : WriteLn('Bleu') ;
End ;
END.
```

Cet *exemple34* montre que l'instruction de contrôle conditionnel `Case Of` fonctionne aussi avec le type énuméré, conformément à ce qui a été dit dans le chapitre sur les types intervalles.

```
Program exemple35 ;
Var color:(red, yellow, green, black, blue) ;
BEGIN
color:=red ;
Repeat
Inc(color) ;
Until color>green ;
If color=black Then WriteLn('Noir') ;
END.
```

Cet *exemple35* montre que comme toute variable, *color* - qui est de type énuméré - peut être sujette à des tests booléens. Ici, sa valeur est incrémentée dans une boucle `Repeat` qui ne s'arrête que lorsque *color* atteint une valeur qui dans le type énuméré est supérieure à la valeur *green*. Ensuite un test `If` vient confirmer que la dernière valeur prise par *color* (à laquelle on s'attendait au vu de la définition du type énuméré appliqué à *color*) est *black*.

---

5. Enregistrement conditionnel

Lors de la création d'un enregistrement (type structuré), il est quelquefois nécessaire de pouvoir en fonction d'un champs, décider de la création d'autres champs de tel ou tel type. Une telle programmation s'effectue grâce à la syntaxe `Case Of` que l'on connaissait déjà pour tester des variables de type intervalle. Cette fois-ci on va tester des champs dont les valeurs doivent être de type énuméré !

Syntaxe :

```
Type type_enumere=(élément1, élément2, ... élémentN) ;
   mon_type=Record
      champ1:type1;
      Case champ2:type_enumere Of
         élément1:(champ3:type3);
         élément2:(champ4:type4; champ5:type5; ... champM:typeM);
         ...
         élémentN:( );
      End;
```

Le principe c'est de créer un type énuméré dont les valeurs seront les valeurs-test de l'instruction `Case Of`. Ensuite, on créer le type enregistrement et on commence à créer les champs fixes, en ne mettant la structure conditionnelle qu'en dernier car son `End;` est confondu avec celui du `Record`. On écrit `Case` + un autre champ fixe dont la valeur conditionne la suite + `:` (deux points) + le type de ce champ qui est le type énuméré créé plus haut + `Of`. Ensuite à partir de la ligne suivante on procède comme pour un `Case Of` normal : on écrit les différentes valeurs possibles c'est-à-dire les éléments (par forcément tous...) du type énuméré + `:` (deux points) + entre parenthèses `(  )` on met les champs que l'on veut suivant la valeur de l'élément sans oublier de spécifier leur type.
Donc suivant la valeur d'un champs fixe (de type énuméré), on va procéder à la création d'un champ, de plusieurs champs ou même d'aucun champ (pour cette dernière option, il suffit de ne rien mettre entre les parenthèses).

```
Program exemple30a ;
Const Nmax=1 ;

Type materiaux=(metal, beton, verre) ;
   produit=Record
      nom : String[20] ;
      Case matiere : materiaux Of
         metal : (conductivite : Real) ;
         beton : (rugosite : Byte) ;
         verre : (opacite : Byte; incassable : Boolean) ;
      End ;
   tab=Array[1..Nmax] Of produit ;

Procedure affichage(prod : produit) ;
Begin
With prod Do
   Begin
   WriteLn('Produit ', nom) ;
   Case matiere Of
      metal : WriteLn('Conductivité : ', conductivite) ;
      beton : WriteLn('Rugosité : ', rugosite) ;
      verre : Begin
         WriteLn('Opacité : ', opacite) ;
         If incassable Then WriteLn('Incassable') ;
         End ;
      End ;
   End ;
End ;

Var x : tab ;
i : Integer ;
BEGIN
With x[1] Do
   Begin
   nom:='Lampouille' ;
```

```
      matiere:=verre ;
      opacite:=98 ;
      incassable:=true ;
      End ;
   For i:=1 To Nmax Do affichage(x[i]) ;
   END.
```

Note : Il est absolument nécessaire de remplir le champs qui conditionne le choix des autres champs avant de remplir les champs qui sont soumis à condition. Sinon, il est renvoyé des résultats absurdes.

```
   Program exemple30b ;
   Type toto=Record
      Case i:Integer Of
         1:( ) ;
         2:(a:Real) ;
         3:(x, y:String) ;
      End ;
   Var x:toto ;
   BEGIN
   x.i:=2 ;
   x.a:=2.23 ;
   WriteLn(x.a) ;
   x.i:=3 ;
   x.x:='Castor' ;
   WriteLn(x.x) ;
   END.
```

Cet *exmple30b* montre que l'on peut utiliser des variables d'autres types que celui énuméré pour créer des enregistrements conditionnels. Ici c'est un `Integer` qui est utilisé et dont la valeur dans le programme conditionne l'existence d'autres champs.

CHAPITRE XXI : Tableaux

Il est courant d'utiliser des tableaux afin d'y stocker temporairement des données. Ainsi, une donnée peut être en relation avec 1, 2 ou 3 (ou plus) entrées. L'intérêt du tableau est de pouvoir stocker en mémoire des données que l'on pourra retrouver grâce à d'autres valeurs à l'aide de boucles, de formules, d'algorithmes. On peut utiliser un tableau afin de représenter l'état d'un échiquier, le résultat d'une fonction mathématique... Il est possible d'introduire des variables de presque tous les types au sein d'un tableau : `Char`, `Integer`, `Real`, `String`, `Byte`, `Record`, etc.

Un tableau, tout comme une variable quelconque doit être déclaré dans la partie déclarative du programme. On doit toujours spécifier le type des variables qui seront introduites dans le tableau.

Syntaxe :

`Var` *nom_du_tableau* `: Array[`*MinDim*`..`*MaxDim*`] Of` *type* `;`

Note : les valeurs *MinDim* et *MaxDim* doivent être des `Integer` ou des `Char` (c'est-à-dire de *type énuméré*). Avec *MinDim* inférieur à *MaxDim*. L'un ou les deux peuvent êtres négatifs. Le type des variables qui seront mises dans le tableau devra être celui là : *type*.

Remarque : il ne peut y avoir qu'un seul type de variable au sein d'un tableau.

Exemples :

```
Var tab1 : Array[0..10] Of Byte ;

Var tab2 : Array[1..100] Of Integer ;

Var tab3 : Array[-10..10] Of Real ;

Var tab4 : Array[50..60] Of String ;

Var tab5 : Array['A'..'S'] Of Char ;

Var tab6 : Array['a'..'z'] Of Extended ;
```

Remarque : que les bornes d'un tableau soient déclarées par des valeurs de type caractère (`Char`) n'interdit pas pour autant de remplir le tableau de nombres à virgules (voir le *tab6* ci-dessus). Car en effet, le type des bornes d'un tableau n'influe aucunement sur le type des variables contenues dans le tableau. Et réciproquement, le type des variables d'un tableau de renseigne en rien sur le type des bornes ayant servi à sa déclaration.

Un tableau peut avoir plusieurs dimensions. Si toutefois, vous imposez trop de dimensions ou des *index* trop importants, une erreur lors de la compilation vous dira : `Error 22: Structure too large.`

Syntaxes :

```
Var nom_du_tableau : Array[MinDim1..MaxDim1, MinDim2..MaxDim2] Of type ;
Var nom_du_tableau : Array[MinDim1..MaxDim1, MinDim2..MaxDim2, MinDim3..MaxDim3] Of type ;
```

Exemples :

```
Var tab1 : Array[0..10, 0..10] Of Byte ;

Var tab2 : Array[0..10, 0..100] Of Integer ;

Var tab3 : Array[-10..10, -10..10] Of Real ;

Var tab4 : Array[5..7, 20..22] Of String ;

Var tab5 : Array[1..10, 1..10, 1..10, 0..2] Of Char ;
```

La technique pour introduire des valeurs dans un tableau est relativement simple. Il suffit de procéder comme pour une variable normale, sauf qu'il ne faut pas oublier de spécifier la position *index*qui indique la place de la valeur dans la dimension du tableau.

Syntaxes :

```
nom_du_tableau[index] := valeur ;
nom_du_tableau[index1, index2] := valeur ;
```

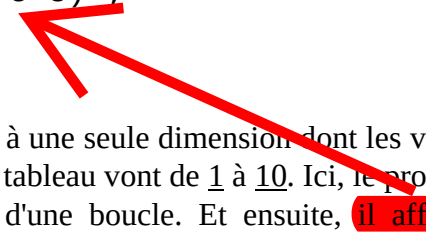
Note : la variable *index* doit nécessairement être du même type énuméré que celui utilisé pour la déclaration du tableau.

On peut introduire dans un tableau les valeurs d'un autre tableau. Mais pour cela, il faut que les deux tableaux en question soient du même type (ou de types compatibles), qu'ils aient le même nombre de dimension(s) et le même nombre d'éléments.

Syntaxe :

```
nom_du_premier_tableau:= nom_du_deuxième_tableau ;
```

```
Program exemple26 ;
Var tab : Array[1..10] Of Integer ;
i : Integer ;
BEGIN
   For i:=1 To 10 Do
      Begin
         tab[i]:=i ;
         WriteLn(sqrt(tab[i])*5+3) ;
      End;
END.
```



Ce programme *exemple26* utilise un tableau à une seule dimension dont les valeurs seront des variables de type `integer`. Les "cases" du tableau vont de 1 à 10. Ici, le programme donne à chaque case la valeur de l'*index* à l'aide d'une boucle. Et ensuite, il affiche les valeurs contenues dans le tableau.

Il existe une autre manière de déclarer un tableau de dimensions multiples en créeant un tableau de tableau. Mais cette technique n'est pas la plus jolie, pas la pratique, pas la plus appréciée aux examens... Donc à ne pas utiliser !

Syntaxe :

```
Var nom_du_tableau : Array[MinDim1..MaxDim1] Of
Array[MinDim2..MaxDim2] Of type ; { syntaxe désuette }
```

Mais une autre manière d'introduire des valeurs accompagne ce type de déclaration.

Syntaxe :

```
nom_du_tableau[index1][index2] := valeur ;
```

Le passage d'un tableau (type complexe) en paramètre à une procédure pose problème si on ne prend pas des précautions. C'est-à-dire qu'il faut déclarer un type précis de tableau (qui sera un type simple).

Syntaxe :

```
Type nom_du_nouveau_type_tableau = Array[DimMin..DimMax] Of Type ;
Var nom_du_tableau : nom_du_nouveau_type_tableau ;
Procedure nom_de_la_procedure(Var nom_du_tableau :
nom_du_nouveau_type_tableau) ;
Program exemple27 ;
Uses crt ;
Type tableau = Array[1..10] Of Integer ;
Procedure saisie(Var tab:Tableau) ;
Var i:Integer ;
Begin
   For i:=1 to 10 Do
      Begin
         Write('Entrez la valeur de la case n°',i,'/10 : ') ;
         ReadLn(tab[i]) ;
      End ;
End ;
Procedure tri(Var tab:Tableau) ;
Var i,j,x:Integer ;
Begin
   For i:=1 To 10 Do
      Begin
      For j:=i To 10 Do
         Begin
         if tab[i]>tab[j] Then
            Begin
               x:=tab[i] ;
               tab[i]:=tab[j] ;
               tab[j]:=x ;
            End ;
         End ;
      End ;
```

```
End ;
Procedure affiche(tab:Tableau) ;
Var i:Integer ;
Begin
   For i:=1 To 10 Do Write(tab[i],' ') ;
   WriteLn ;
End ;
Var tab1,tab2:Tableau ;
     i:Integer ;
BEGIN
   ClrScr ;
   saisie(tab1) ;
   tab2:=tab1 ;
   tri(tab2) ;
   WriteLn('Série saisie :') ;
   affiche(tab1) ;
   WriteLn('Série triée :') ;
   affiche(tab2) ;
END.
```

Ce programme *exemple27* a pour but de trier les éléments d'un tableau d'entiers dans l'ordre croissant. Pour cela, il déclare un nouveau type de tableau grâce à la syntaxe `Type`. Ce nouveau type est un tableau à une dimension, de 10 éléments de type `integer`. Une première procédure *saisie* charge l'utilisateur d'initialiser le tableau *tab1*. Le programme principal copie le contenu de ce tableau vers un autre appelé *tab2*. Une procédure *tri* range les éléments de *tab2*dans l'ordre. Et une procédure *affiche* affiche à l'écran le tableau *tab1* qui contient les éléments dans introduits par l'utilisateur et le tableau *tab2* qui contient les mêmes éléments mais rangés dans l'ordre croissant.

Il est également possible d'introduire dans un tableau des données complexes, c'est-à-dire de déclarer un tableau en type complexe (`Record`).

Syntaxe :

```
Var tab : Array[1..10] Of Record
     nom : String[20] ;
     age : Byte ;
          End ;
```

Introduire des valeurs dans un tel tableau nécessite d'utiliser en même temps la syntaxe des tableaux et des types complexes.

Syntaxe :

```
tab[5].nom := 'CyberZoïde' ;
```

La déclaration de tableau en tant que constante nécessite de forcer le type (le type tableau bien sûr) à la déclaration. (voir Chap XXVI Constantes)

Syntaxe :

```
Const a : Array[0..3] Of Byte = (103, 8, 20, 14) ;
```

```
b : Array[-3..3] Of Char = ('e', '5', '&', 'Z', 'z', ' ', #80) ;
c : Array[1..3, 1..3] Of Integer = ((200, 23, 107), (1234, 0, 5), (1, 2, 3)) ;
d : Array[1..26] Of Char = 'abcdefghijklmnopqrstuvwxyz';
```

Attention : cette forme parenthèsée est réservée aux constantes car elle ne permet pas de passer un tableau en paramètre à une procédure ou d'initialiser un tableau.

CHAPITRE XXII : Une bonne interface DOS

Lors de la création d'un programme informatique, l'élaboration de l'interface utilisateur est très critique, demande du code et de la patience. Outre les procédures systématiques de gestion des erreurs et de contrôle des entrées, la présentation des données aussi bien en mode MS-DOS qu'en mode graphique conditionne bien souvent la qualité de la diffusion d'un programme. Il est vrai qu'en règle générale, un programme écrit en Pascal est surtout destiné à une utilisation personnelle mais dans le cas d'une distribution plus large (internet, amis...) il est capital de présenter à l'utilisateur les données de façon ordonnée, aérée et claire. Car quand dans un programme des informations apparaissent à l'écran et qu'on ne sait pas d'où elles viennent, ni comment intervenir sur son déroulement, l'utilisateur est frustré et l'abandonne. Il est donc nécessaire, ne serait-ce que pour être à l'aise devant son écran, de construire une interface simple et claire.

Doter son programme d'une interface efficace commence déjà par lui donner nom ou du moins un descriptif de l'objectif ou du travail qu'il doit produire ainsi que des diverses opérations qu'il est capable de réaliser. En générale, cela commence par l'affichage d'un menu du type suivant :

```
    - GESTION D'UN TABLEAU -

[A] - Abandon
[B] - Création du tableau
[C] - Affichage du tableau
[D] - Modification d'un élément du tableau
[E] - Ajout d'un élément à la fin du tableau
[F] - Suppression d'un élément du tableau
[G] - Insertion d'un élément dans le tableau
[H] - Recherche du plus petit élément dans le tableau
[I] - Recherche du plus grand élément dans le tableau

Entrez votre choix au clavier : _
```

Il faut être capable de gérer correctement l'entrée de l'utilisateur pour être sûr qu'il entre un paramètre correct :

Syntaxe :

```
Procedure menu(Var reponse : Char) ;
Begin
Repeat
ClrScr;
WriteLn('[A] - Abandon') ;
WriteLn('[B] - Création du tableau') ;
WriteLn('[C] - Affichage du tableau') ;
WriteLn('[D] - Modification d''un élément du tableau') ;
WriteLn('[E] - Ajout d''un élément à la fin du tableau') ;
WriteLn('[F] - Suppression d''un élément du tableau') ;
WriteLn('[G] - Insertion d''un élément dans le tableau');
WriteLn('[H] - Recherche du plus petit élément dans le tableau') ;
WriteLn('[I] - Recherche du plus grand élément dans le tableau') ;
WriteLn;
```

```
   Write('Entrez votre choix au clavier : ');
   ReadLn(reponse) ;
   Until UpCase(reponse) In ['A'..'I'] ;
   End ;
```

Ensuite, il faut pouvoir détailler les opération et leurs résultats afin d'offir le maximum d'information à l'utilisateur :

```
[H] - RECHERCHE DU PLUS PETIT ELEMENT DU TABLEAU

Résultat de la recherche :
               -   rang : 27
               - valeur : -213

Recherche terminée, tapez sur <ENTREE> pour retourner au menu.
```

La recherche d'erreur fait également parti des prorogatives du programmeur, c'est-à-dire que le programme doit être capable de repérer des entrées erronées, d'expliquer à l'utilisateur ce qui ne vas pas, pourquoi ça ne vas pas et lui permettre de recommencer sa saisie.

Syntaxe :

```
   Procedure modifier(Var tab1:tableau; n:Integer) ;
   Var rang,valeur : Integer ;
   test : Boolean ;
   Begin
   ClrScr ;
   WriteLn('[D] - MODIFICATION D'UN ELEMENT DU TABLEAU') ;
   If n>=1 Then
   Begin
   Repeat
```

```
   Write('Entrez le rang [1,',n,'] de l''élément à modifier : ') ;
   ReadLn(rang) ;
   test := (rang>0) And (rang<=n) ;
   Until test ;
   Write('Entrez la nouvelle valeur de l''élément : ') ;
   ReadLn(valeur) ;
   tab1[rang] := valeur ;
   WriteLn('Modification terminée, tapez sur ENTREE pour retourner au menu.') ;
   End
   Else WriteLn('Aucun élément dans le tableau.') ;
   ReadLn ;
   End ;
```

Très souvent le tableau est la forme la plus appropriée pour présenter des données à l'écran texte, il est donc souhaite de construire des tableau multi-cadres afin d'avoir à l'écran plusieurs informations simultanément comme le montre l'exemple suivant :

```
SuperCalculator    2.5

Module : CALCUL DE FONCTION                    Page :   3

      Valeur de X          Valeur de Y=f(X)

         1450                  2103818
         1500                  2251364
         1550                  2403909
         1600                  2561455
         1650                  2724000
         1700                  2891545
         1750                  3064091

      Appuyez sur <ENTREE> pour continuer...
```

Ici, les caractères spéciaux de la table ASCII (pour MS-DOS) ont été remplacés par des tirets ou des I car non affichables sous Windows.

```
Program exemple28 ;

Uses Crt ;

Var i,j : Integer;
x : Real ;
test : Boolean ;

Procedure menu(i : Integer; test : Boolean) ;
Begin
WriteLn('-------------------------------------------------------------') ;
WriteLn(' I SuperCalculator 2.5 I') ;
WriteLn('-------------------------------------------------------------') ;
If test Then WriteLn('Faire afficher le menu principal...')
Else
Begin
WriteLn('-------------------------------------------------------------') ;
WriteLn(' I Module : CALCUL DE FONCTION I Page : ',i,' I') ;
WriteLn('-------------------------------------------------------------') ;
WriteLn('  I Valeur de X I Valeur de Y=f(X) I') ;
WriteLn(' I I I') ;
End ;
End ;

Function f(x : Real) : Real ;
Begin
f:=Sqr(x) ;
End ;

BEGIN
ClrScr ;
i := 0 ;
j := 1 ;
test := False ;
```

```
menu(j,test) ;
x := 0 ;
Repeat
Inc(i) ;
x := x+50 ;
WriteLn('I',Round(x):12,'I',Round(f(x)):15,'I') ;
test := x>1700;
If ((i Mod 14)=0) Or test Then
Begin
WriteLn('------------------------------') ;
WriteLn(' I Appuyez sur <ENTREE> pour continuer... I') ;
Write('------------------------------') ;
ReadLn ;
ClrScr ;
Inc(j) ;
menu(j,test) ;
End ;
Until test ;
ReadLn ;
END.
```

CHAPITRE XXIII : Gestion de la mémoire par l'exécutable

1. Limite virtuelle de la mémoire.

Une fois compilé (commande **Run**, **Compile** ou **Make**), un programme gère la mémoire très rigoureusement. Le tableau ci-dessous vous montre que les variables principales, les variables locales des sous-programmes et les pointeurs ne sont pas stockés dans les mêmes parties de la mémoire. En effet, les variables principales de font la part belle, la mémoire allouée aux pointeurs (très gourmands en mémoire) est variable et celle destinée aux variables locales est assez restreinte.

Pile (*Stack*) 16 ko par défaut	Destiné aux variables locales des sous-programmes (procédures, fonctions) ainsi qu'aux valeurs passées par valeur aux sous-programmes.
Tas (*Heap*) de 0 à 640 ko	Réservé aux pointeurs.
Segment de données 64 ko	Destiné aux variables du programme principal.

Explications : les sous-programmes étant destinés à des calculs intermédiaires, ils n'ont guère besoins d'énormément de ressource mémoire. Quand aux pointeurs que l'on verra dans les chapitres suivant, ils sont destinés à la manipulation d'une grande quantité de données.

Mais il est toutefois possible de modifier manuellement une telle organisation de la mémoire afin, par exemple, de privilégier la **Pile** grâce au commentaire de compilation suivant : `{$M` *n1*`,` *n2*`,` *n3*`}`. Ce type de commentaire est destiné au compilateur Borland Pascal qui inscrira les informations spécifiées dans le programme compilé. Un commentaire de compilation se présente entre accolades comme n'importe quel autre commentaire, mais un signe dollar "`$`" signifie qu'il est destiné au compilateur. Quand au "`M`" il dit au compilateur qu'on souhaite réorganiser la disposition de la mémoire à l'aide des valeurs *n1*, *n2* et *n3* qui spécifient respectivement la taille en kilo octets de la **Pile** (doit être inférieur à 64 ko), la taille minimale et la taille maximale (inférieur à 640 ko) du **Tas**.

Mais pourquoi s'enquiquiner avec ça ? Tout simplement parce qu'il pourra vous arriver d'avoir insuffisamment de mémoire à cause d'un tableau trop long par exemple. Si vous déclarer une telle variable dans une procédure :

```
Var tableau : Array[1..50, 1..100] Of Real ;
```

vous obtiendrez le message d'erreur **n°22 : Structure too large** qui veut dire que votre

variable tient trop de place pour être stockée dans la mémoire allouée. Car en effet, ce tableau tient : 50 * 100 * 6 octets = 29 ko (1 ko = 2^10 = 1024 octets) 29 ko > 16 ko donc le compilateur renvoit une erreur. Et le seul moyen qui vous reste est de modifier les valeurs correspondantes aux grandeurs allouées à la **Pile** par un commentaire de compilation ou en allant dans le menu **Option/Memory Size**. D'où l'intérêt du Chapitre 4 ("*Différents types de variables*") qui vous indique la taille de chaque type de variable.

2. Passage d'un paramètre à un sous-programme.

Dans le Chapitre 7 ("*Procédures et sous-programmes*") vous avez appris qu'on pouvait passer un paramètre par valeur ou bien par adresse à une procédure paramétrée. Vous avez également compris l'intérêt de la syntaxe `Var` dans la déclaration d'une procédure. Quand un sous-programme est appelé, le programme compilé réalise en mémoire (dans la **Pile**) une copie de chaque argument passé au sous-programme. Ces copies ne sont que temporaires puisque destinées au fonctionnement de sous-programmes qui n'interviennent que temporairement dans le programme. Ainsi un changement de valeur au sein de la procédure d'un paramètre passé par adresse sans la syntaxe `Var` n'est pas répercuté dans le programme principale. Alors que dans le cas de la présence de la syntaxe `Var`, le programme ne duplique pas la valeur ainsi passée à la procédure dans la **Pile**, mais le paramètre du sous-programme et la variable principale (passée comme argument à la procédure) sont joints vers la zone de la mémoire de la variable principale (dans la partie **Segment de données**). Ainsi toute variation interne à la procédure est répercuté directement sur l'argument (la variable du programme principal passé en paramètre).

CHAPITRE XXIV : Pointeurs

Un pointeur est une variable qui contient l'adresse mémoire d'une autre variable stockée en mémoire. Soit `P` le pointeur et `P^` la variable "pointée" par le pointeur. La déclaration d'une variable pointeur réserve 4 octets nécessaires au codage de l'adresse mémoire mais ne réserve aucune mémoire pour la variable pointée.

Jusqu'alors nous avions vu que la déclaration d'une variable provoque automatiquement la réservation d'un espace mémoire qui est fonction du type utilisé. Voir Chapitre 4 ("*Différents types de variables*") pour la taille en mémoire de chacun des types de variables utilisés ci-après.

Exemples :

```
Var somme : Integer ;
```
Réservation de 4 octets dans la mémoire.
```
Var moyenne : Real ;
```
Réservation de 6 octets dans la mémoire.
```
Var tableau : Array[1..100] Of Integer ;
```
Réservation de 400 octets (100*4) dans la mémoire.
```
Var nom : String[20] ;
```
Réservation de 21 octets dans la mémoire.
```
Var x,y,z : Integer ;
```
Réservation de 12 octets (3*4) dans la mémoire.
```
Var tab1,tab2 : Array[0..10,0..10] Of Integer ;
```
Réservation de 968 octets (2*11*11*4) dans la mémoire.
```
Type persone = Record
    nom,prenom : String[20] ;
    age : Byte ;
    tel : Integer ;
    End;
Var client,fournisseur : personne ;
```

Réservation de 94 octets (2*(2*21+1+4)) dans la mémoire.

On comprend rapidement que s'il vous prenait l'envie de faire une matrice de 100*100 réels (100*100*6 = 60 Ko) à passer en paramètre à une procédure, le compilateur vous renverait une erreur du type : **Structure too large** car il lui est impossible de réserver plus de 16 Ko en mémoire pour les variables des sous-programmes. Voir chapitre 23 ("*Gestion de la mémoire par l'exécutable*").

D'où l'intérêt des pointeurs car quelque soit la grosseur de la variable pointée, la place en mémoire du pointeur est toujours la même : **4 octets**. Ces quatres octets correspondent à la taille mémoire nécessaire pour stocker l'adresse mémoire de la variable pointé. Mais qu'est-ce qu'est une adresse mémoire ? C'est en fait deux nombres de type `Word` (2 fois 2 octets font bien 4) qui représentent respectivement l'indice du segment de donnée utilisé et l'indice du premier octet servant à coder la variable à l'intérieur de ce même segment de donné (un segment étant un bloc de 65535 octets). Cette taille de segment implique qu'une variable ne peut pas dépasser la taille de 65535 octets, et que la taille de l'ensemble des variables globales

ne peut pas dépasser 65535 octets ou encore que la taille de l'ensemble des variables d'un sous-programme ne peut dépasser cette même valeur limite.

La déclaration d'un pointeur permet donc de réserver une petite place de la mémoire qui pointe vers une autre qui peut être très volumineuse. L'intérêt des pointeurs est que la variable pointée ne se voit pas réserver de mémoire, ce qui représente une importante économie de mémoire permettant de manipuler un très grand nombre de données. Puisque la **Pile** normalement destinée aux variables des sous-programmes est trop petite (16 Ko), on utilise donc le **Tas** réservé au pointeur qui nous laisse jusqu'à 64 ko, soit quatre fois plus !

Avant d'utiliser une variable de type pointeur, il faut déclarer ce type en fonction du type de variable que l'on souhaite pointer.

Exemple 1 :

```
Type PEntier = ^Integer ;
Var P : PEntier ;
```

On déclare une variable *P* de type *PEntier* qui est en fait un pointeur pointant vers un `Integer` (à noté la présence indispensable de l'accent circonflexe!). Donc la variable *P* contient une adresse mémoire, celle d'une autre variable qui est elle, de type `Integer`. Ainsi l'adresse mémoire contenue dans *P* est l'endroit où se trouve le premier octet de la variable de type `Integer`. Il est inutile de préciser l'adresse mémoire de fin de l'emplacement de la variable de type `Integer` car une variable de type connu quelque soit sa valeur occupe toujours le même espace. Le compilateur sachant à l'avance combien de place tient tel ou tel type de variable, il lui suffit de connaître grâce au pointeur l'adresse mémoire du premier octet occupé et de faire l'addition *adresse mémoire contenue dans le pointeur + taille mémoire du type utilisé* pour définir totalement l'emplacement mémoire de la variable pointée par le pointeur.

Tout ça c'est très bien mais comment fait-on pour accéder au contenu de la variable pointée par le pointeur ? Il suffit de manipuler l'identificateur du pointeur à la fin duquel on rajoute un accent circonflexe en guise de variable pointée.

Exemple :

```
P^ := 128 ;
```

Donc comprenons-nous bien, *P* est le pointeur contenant l'adresse mémoire d'une variable et *P^* (avec l'accent circonflexe) contient la valeur de la variable pointée. On passe donc du pointeur à la variable pointée par l'ajout du symbole spécifique `^` à l'identificateur du pointeur.

```
Type Tableau = Array[1..100] Of Real ;
   PTableau = ^Tableau ;
Var P : PTableau ;
```

Ici, on déclare une type *Tableau* qui est un tableau de 100 réels. On déclare aussi un type de pointeur *PTableau* pointant vers le type *Tableau*. C'est-à-dire que dans toute variable de type *PTableau*, sera contenue l'adresse mémoire du premier octet d'une variable de type *Tableau*. Ce type *Tableau* occupe 100*6 = 600 octets en mémoire, le compilateur sait donc

parfaitement comment écrire une variable de type *Tableau* en mémoire. Quand à la variable *P* de type *PTableau*, elle contient l'adresse mémoire du premier octet d'une variable de type *Tableau*. Pour accéder à la variable de type *Tableau* pointée par *P*, il suffira d'utiliser la syntaxe *P*^.

P étant le pointeur et *P*^ étant la variable pointée. *P* contenant donc une adresse mémoire et *P*^ contenant un tableau de 100 réels. Ainsi *P*^`[10]` représente la valeur du dixième élément de *P*^ (c'est donc un nombre de type `Real`) tandis que *P[10]* (déclenche une erreur du compilateur) ne représente rien pas même l'adresse mémoire du dixième élément de *P*^.

La déclaration au début du programme des diverses variables et pointeurs a pour conséquence que les variables se voient allouer un bloc mémoire à la compilation. Et ce dernier reste réservé à la variable associée jusqu'à la fin du programme. Avec l'utilisation des pointeurs, tout cela change puisque la mémoire est alloué dynamiquement. On a vu que seul le pointeur se voit allouer (réserver) de la mémoire (4 octets, c'est très peu) pour toute la durée de l'exécution du programme mais pas la variable correspondante. Il est cependant nécessaire de réserver de la mémoire à la valeur pointée en cours de programme (et non pas pour la totalité) en passant en paramètre un pointeur *P* qui contiendra l'adresse mémoire correspondant à la variable associée *P*^. Pour pouvoir utiliser la variable pointée par le pointeur, il est absolument indispensable de lui réserver dynamiquement de la mémoire comme suit :

Syntaxe :

```
New(P) ;
```

Et pour la supprimer, c'est-à-dire libérer la place en mémoire qui lui correspondait et perdre bien sûr son contenu :

Syntaxe :

```
Dispose(P) ;
```

Ainsi lorsqu'on en a fini avec une variable volumineuse et qu'on doit purger la mémoire afin d'en utiliser d'autres tout autant volumineuses, on utilise `Dispose`. Si après,au cours du programme on veut réallouer de la mémoire à une variable pointée par un pointeur, c'est possible (autant de fois que vous voulez !).

```
Type Tab2D = Array[1..10,1..10] Of Integer ;
   PMatrice = ^Tab2D ;
Var GogoGadgetAuTableau : PMatrice ;
```

On a donc une variable *GogoGadgetAuTableau* (4 octets) qui pointe vers une autre variable (10*10*6 = 600 octets) de type *Tab2D* qui est un tableau de deux dimensions contenant 10*10 nombres entiers. Pour être précis, la variable *GogoGadgetAuTableau* est d'un type *PMatrice* pointant vers le type *Tab2D*. Donc la taille de *GogoGadgetAuTableau* sera de 4 octets puisque contenant une adresse mémoire et *GogoGadgetAuTableau*^ (avec un ^) sera la variable de type *Tab2D* contenant 100 nombres de type `Integer`.

On pourra donc affecter des valeurs à la variable comme suit :
GogoGadgetAuTableau`^[i,j]:=3 ;`.

Toutes les opérations possibles concernant les affectations de variables, ou leur utilisation dans des fonctions sont vraie pour les variables pointée par des pointeurs. Il est bien entendu impossible de travailler sur la valeur pointée par le pointeur sans avoir utilisé auparavant la procédure `New` qui alloue l'adresse mémoire au pointeur.

```
Program exemple29c ;
Const Max = 10 ;
Type Tab2D = Array[1..Max,1..Max] Of Integer ;
   PMatrice = ^Tab2D ;
Var GogoGadgetAuTableau : PMatrice ;
   i, j, x : Integer ;
BEGIN
New(GogoGadgetAuTableau);
For i:=1 to Max Do
   Begin
   For j:=1 to Max Do GogoGadgetAuTableau^[i,j] := i+j ;
   End;
x := GogoGadgetAuTableau^[Max,Max] * Sqr(Max) ;
WriteLn(Cos(GogoGadgetAuTableau^[Max,Max])) ;
Dispose(GogoGadgetAuTableau);
END.
```

Ce court *exemple29c* montre qu'on utilise une variable pointée par un pointeur comme n'importe quelle autre variable.

Note : un pointeur peut pointer vers n'importe quel type de variable sauf de type fichier (`File Of Text`).

```
Program exemple29d ;
Type Point = Record
   x, y : Integer ;
   couleur : Byte ;
   End ;
   PPoint = ^Point ;
Var Pixel1, Pixel2 : PPoint ;
BEGIN
Randomize ;
New(Pixel1);
New(Pixel2);
With Pixel1^ Do
   Begin
   x := 50 ;
   y := 100 ;
   couleur := Random(14)+1 ;
   End ;
Pixel2^ := Pixel1^ ;
Pixel2^.couleur := 0 ;
```

```
Dispose(Pixel1);
Dispose(Pixel2);
END.
```

Dans ce programme *exemple29d*, on déclare deux variables pointant chacune vers une variable de type *Point* ce dernier étant un type structuré (appelé aussi enregistrement). La ligne d'instruction : `Pixel2^ := Pixel1^ ;` signifie qu'on égalise champ à champ les variables *Pixel1* et *Pixel2*.

Si les symboles ^ avaient été omis, cela n'aurait pas provoquer d'erreur mais cela aurait eut une tout autre signification : `Pixel2 := Pixel1 ;` signifie que le pointeur *Pixel2* prend la valeur du pointeur *Pixel1*, c'est-à-dire que *Pixel2* pointera vers la même adresse mémoire que *Pixel1*. Ainsi les deux pointeurs pointent vers le même bloc mémoire et donc vers la même variable. Donc *Pixel1^* et *Pixel2^* deviennent alors une seule et même variable. Si l'on change la valeur d'un champ de l'une de ces deux variables, cela change automatiquement le même champ de l'autre variable !

Note : On ne peut égaliser deux pointeurs que s'ils ont le même type de base (comme pour les tableaux). Et dans ce cas, les deux pointeurs pointent éxactement vers la même variable. Toute modification de cette variable par l'intermédiaire de l'un des deux pointeurs se répercute sur l'autre.

Autre note : Je rappelle qu'il est impossible de travailler sur la valeur pointée par le pointeur sans avoir utilisé auparavant la procédure `New` qui alloue l'adresse mémoire au pointeur. Si vous compilez votre programme sans avoir utilisé `New`, un erreur fatale vous ramènera à l'ordre !

```
Program exemple29e ;
Const Max = 10 ;
Type Personne = Record
   nom, prenom : String ;
   matricule : Integer ;
   End ;
   Tableau = Array[1..Max] Of Personne ;
   PTableau = ^Tableau ;
Var Tab : PTableau ;
   i : Integer ;
BEGIN
New(Tab);
With Tab^[1] Do
   Begin
   nom := 'Cyber' ;
   prenom := 'Zoïde' ;
   matricule := 1256 ;
   End ;
For i:=1 To Max Do WriteLn(Tab^[i].nom) ;
Dispose(Tab);
END.
```

Il est possible de combiner les enregistrements, les tableaux et les pointeurs. Cela donne un vaste panel de combinaisons. Essayons-en quelques unes.

```
Type TabP = Array[1..100] Of ^Integer ;
Var Tab : TabP ;
```

Tableau de pointeurs pointant vers des entiers. *Tab*`[i]` est un pointeur et *Tab*`[i]^` est un entier.

```
Type Tab = Array[1..100] Of Integer ;
   PTab = ^Tab ;
Var Tab : PTab ;
```

Pointeur pointant vers un tableau d'entiers. *Tab*`^[i]` est un entier et *Tab* est un pointeur.

```
Const Max = 20 ;
Type Station = Record
   nom : String ;
   liaisons : Array[1..10] Of Station ;
   End ;
   TabStation = Array[1..Max] Of Station ;
   PTabStation = ^TabStation ;
Var France : PTabStation ;
```

France est un pointeur pointant vers un tableau d'enregistrement dont l'un des champs est un tableau et l'autre un enregistrement (récursif).

Pour alléger le code, on aurait pu faire plus court :

```
Const Max = 20 ;
Type Station = Record
   nom : String ;
   liaisons : Array[1..10] Of Station ;
   End ;
   TabStation = Array[1..Max] Of Station ;
Var France : ^TabStation ;
```

Il existe des fonctions similaires au couple `New` et `Dispose` :

Syntaxes :

```
GetMem(Pointeur,Mémoire) ;
```

Cette fonction réserve un nombre d'octets en mémoire égale à *Mémoire* au pointeur *Pointeur*. *Mémoire* correspond à la taille de la variable pointée par le pointeur *Pointeur*.

```
FreeMem(Pointeur,Mémoire) ;
```

Cette fonction supprime de la mémoire le pointeur *Pointeur* dont la variable pointée occupait *Mémoire* octets.

Note : Si vous utilisez `New` pour le pointeur *P*, il faudra lui associer `Dispose` et non pas `FreeMem`. De même, si vous utilisez `GetMem` pour le pointeur *P*, il faudra utiliser `FreeMem` et non pas `Dispose`.

```
Program exemple29f ;
Const Max = 10 ;
Type Personne = Record
   nom, prenom : String ;
   matricule : Integer ;
   End ;
   Tableau = Array[1..Max] Of Personne ;
   PTableau = ^Tableau ;
Var Tab : PTableau ;
   i : Integer ;
BEGIN
GetMem(Tab,Max*SizeOf(Personne));
For i:=1 To Max Do ReadLn(Tab^[i].nom) ;
FreeMem(Tab,Max*SizeOf(Personne));
END.
```

Vous aurez remarqué que ce programme *exemple29f* est exactement le même que le *exemple29e* mis à part qu'il utilise le couple `GetMem` et `FreeMem` au lieu des traditionnels `New` et `Dispose`. C'est un peu moins sûr à utiliser puisqu'il faut savoir exactement quelle place en mémoire occupe la variable pointée par le pointeur spécifié. Mais ça peut être très pratique si `Max`=90000 (très grand) et si décidez de faire entrer au clavier la borne supérieur du tableau. Voir le programme suivant :

```
Program exemple29g ;
Const Max = 90000 ;
Type Personne = Record
   nom, prenom : String ;
   matricule : Integer ;
   End ;
   Tableau = Array[1..Max] Of Personne ;
   PTableau = ^Tableau ;
Var Tab : PTableau ;
   i : Integer ;
   N : Longint ;
BEGIN
WriteLn('Combien de personnes ?');
ReadLn(N);
GetMem(Tab,N*SizeOf(Personne));
For i:=1 To N Do ReadLn(Tab^[i].nom) ;
FreeMem(Tab,N*SizeOf(Personne));
END.
```

CHAPITRE XXV : Ensembles

Les ensembles en pascal sont les mêmes que ceux que vous connaissez en maths. Ils sont donc régis par les mêmes lois et nécessitent les mêmes opérateurs d'inclusion...
Remarque : L'utilisation des ensembles n'a d'intérêt que pour la programmation des automatismes. Il est fortement probable que la plupart des internautes qui visitent cette page n'aient jamais en s'en servir !

---

1. Déclarations

Un ensemble est une variable qui contient un nombre fini d'éléments de même type. Ce type doit être de type ordinal c'est-à-dire qu'il ne doit être ni de type réel, ni de type chaîne de caractères, ni de type enregistrement, ni de type pointeur. Ce type doit être énuméré (voir Type énuméré) et ne doit pas excéder 256 combinaisons possibles. La déclaration d'une telle variable se fait par l'utilisation de la syntaxe `Set Of`.

Syntaxes :

```
Var identificateur : Set Of type-de-l'ensemble ;
Var id1, id2, id3 : Set Of type-de-l'ensemble ;
```

Exemples :

```
Var hasard : Set Of Byte ;
```

Ici, la variable *hasard* est un ensemble de nombres entiers et dont les valeurs possibles sont dans l'intervalle 0..255.

```
Var binaire : Set Of 0..1 ;
```

Ici, la variable *binaire* est un ensemble de 0 et de 1.

Note : Comme pour tout autre type de variable, on peut déclarer en un seul bloc plusieurs variables "ensemble" du même type.

En général, on cherchera à utiliser des ensembles dont le type sera défini par le programmeur lui-même, c'est-à-dire différent des types de base du Pascal. Ce sont des types énumérés.

Syntaxe :

```
Type type = (élément1, élément2, élément3...) ;
Var ensemble : Set Of type ;
```

Ou forme plus compacte :

```
Var ensemble : Set Of (élément1, élément2, élément3...) ;
```

Le type se résume à une liste d'éléments séparés par des virgules à l'intérieur d'une parenthèse comme le montre la syntaxe ci-haut.

Exemple :

```
Type prenom = (Boris, Hugo, Aurore) ;
Var club : Set Of prenom ;
```

Ou bien :

```
Var club : Set Of (Boris, Hugo, Aurore) ;
```

Dans cet exemple, la variable *club* est un ensemble de type *prenom*, c'est-à-dire que cet ensemble nommé *club* ne peut contenir que les éléments déclarés dans le type *prenom*. Donc la variable *club* pourra contenir une combinaison de ces trois éléments.

Ainsi déclaré, le type apparaît comme une constante dont ont peut connaître l'index des éléments avec la fonction `Ord()`. Ici, `Ord(`*Boris*`)`=0, `Ord(`*Hugo*`)`=1 et `Ord(`*Aurore*`)`=2. Le type fonctionne un peu comme la table ASCII... l'index du premier élément étant zéro.

Les ensembles de sont pas ordonnés. Donc il n'existe pas d'ordre d'apparition des éléments dans une variable ensemble. On peut tout juste être capable de comparer le contenu de deux ensembles de même type, et de déterminer si un élément est inclus ou non dans un ensemble. De plus, un même élément n'apparaît qu'une seule fois dans un ensemble. Et il n'existe pas de fonction qui renvoie le nombre d'éléments d'un ensemble.

---

2. Affectations et opérations

Après avoir vu l'aspect déclaratif des ensembles, on va apprendre à utiliser dans un programme des variables "ensemble". L'ensemble, quelque soit son type peut être un ensemble nul. Pour donner une valeur à un ensemble, c'est-à-dire, spécifier le ou les élément(s) que devra contenir l'ensemble, on utilise l'opérateur habituel d'affectation `:=`. Ce qu'il y a de nouveau, c'est que le ou les élément(s) doivent être séparés par des virgules `,` (comme dans la déclaration du type) et être entre crochets `[ ]` (contrairement à la déclaration).

Syntaxes :

ensemble `:= [ ] ;` { ensemble nul }
ensemble `:= [` *élément* `] ;` { ensemble constitué de l'élément *élément* }
ensemble `:= [` *élément5*`,` *élément1* `] ;` { ensemble constitué des éléments *élément5* et *élément1* }

Rappel : L'ordre des éléments dans une affection ou une comparaison n'a aucune espèce d'importance puisque les ensembles ne sont pas ordonnés.

Note : Une affection à un ensemble en supprime les éléments qu'il contenait avant l'affectation.

Si en cours de programme, on souhaite ajouter ou supprimer un ou des élément(s) à l'ensemble, on doit utiliser les opérateurs additif `+` et soustractif `-` traditionnels.

Syntaxes :

ensemble := *ensemble* + [ ] ; { inutile car ne joute rien ! }
ensemble := *ensemble* + [*élément4*] ;
{ rajoute l'élément *élément4* à l'ensemble }
ensemble := *ensemble* + [*élément3*, *élément2*] ;
{ rajoute les éléments *élément3* et *élément2*}
ensemble := *ensemble* + [*élément1*] - [*élément7*, *élément3*] ;
{ rajoute l'élément *élément1* et supprime les éléments *élément7* et *élément3* }

Pour être exact, les éléments entre crochets représentent des ensembles à part entière. Ces ensembles sont de même type que la variable ensemble auquel on ajoute ou supprime des sous-ensembles. Cela s'explique par le fait que l'on ne peut additionner que des variables de même type : on ne peut pas additionner éléments et ensemble, mais par contre on peut additionner entre eux des ensembles. Ainsi un élément entre crochet est un ensemble et plusieurs éléments séparés par des virgules et entre crochets est aussi un ensemble.

Pour employer le vocabulaire mathématique approprié, + est l'opérateur d'union, - est l'opérateur de complément et on peut en rajouter un troisième : * est l'opérateur d'intersection.

3. Comparaisons

Le seul moyen de connaître le contenu d'un ensemble est de le comparer à d'autres du même type. Ainsi les tests booléens par l'intermédiaire des opérateurs relationnels (voir chapitre Opérateurs), permettent de savoir si tel ou tel élément se trouve dans un ensemble, ou bien si tel ensemble est un inclus dans un autre.

Note : Les opérateurs relationnels stricts sont incompatibles avec les ensembles, ainsi seuls ceux du tableau ci-dessous sont à utiliser avec les ensembles.

Les opérateurs relationnels applicables aux ensembles

Symbole	Description
=	égale
<>	différent
<=	inclu
>=	contenant

L'opérateur In permet de connaître la présence ou non d'un élément dans un ensemble sans avoir à passer par l'utilisation des crochets.

Syntaxes :

If [*élément2*] <= *ensemble* Then ...
{ si l'ensemble constitué de l'élément *élément2* est inclu dans l'ensemble *ensemble* alors...
}
If *élément2* In *ensemble* Then ...
{ si l'élément *élément2* appartient à l'ensemble *ensemble* alors... }

`If` *ensemble* `= [ ] Then` …
{ si l'ensemble *ensemble* est nul, alors... }
test `:= Not([`*élément7*`,` *élément4*`]` `<=` *ensemble*`) ;`
{ le bouléen *test* prend la valeur `True` si l'ensemble constitué des éléments *élément7* et *élément4* n'est pas inclu dans l'ensemble *ensemble* et `False` dans l'autre cas}
`If (`*ensemble1* `*` *ensemble2*`) = [ ] Then` ...
{ si l'intersection des ensembles *ensemble1* et *ensemble2* est nulle, alors... c'est qu'ils n'ont aucun élément en commun marlgré qu'ils soient de même type }
`If (`*ensemble1* `-` *ensemble2*`) = [ ] Then` ...
{ si le complément des ensembles *ensemble1* et *ensemble2* est nul, alors... c'est que *ensemble1* est contenu dans *ensemble2*}

Note : Il est impossible d'utiliser les procédures `Write`(`Ln`) et `Read`(`Ln`) avec les variables de type ensemble.

CHAPITRE XXVI : Constantes

Dans un programme, il est souvent nécessaire sinon utile d'avoir des valeurs de référence ou des conditions initiales. Ainsi, au lieu d'affectations en début de programme qui peuvent s'avérées fastidieuses si elles doivent êtres réexécutées en cours de programme. Alors, pour alléger le code et pour facilité la lisibilité et surtout la compréhension générale du programme, il est préférable d'avoir recours à des constantes.

Une constante, pour être connue de tous les sous-programmes, doit être déclarée avant ces derniers. Si une constante est déclarée après une fonction, pour que cette fonction puisse l'utiliser, la constante doit être passée en paramètre à la fonction. Une constante déclarée avant tous les sous-programme n'a pas besoin d'être passée en paramètre à ses derniers pour y être utilisée (c'est le même principe que pour les variables, fonctions, procédures paramétrées et types).

Comme son nom l'indique, une constante ne change pas de valeur au cours du programme, une valeur de départ (qui peut être nulle) lui est affectée dès sa déclaration avant le code exécutable. C'est l'opérateur égale `=` qui affecte une valeur à une constante après que cette dernière se soit vue affectée un identificateur par l'utilisation du mot réservé `Const`.

Syntaxe :

```
Const identificateur-de-la-constante = valeur ;
```

Exemples :

```
Const nom = 'CyberZoïde' ;
Const TVA = 20.6 ;
Const Esc = #27 ;
```

Le compilateur décide automatiquement d'affecter à la constante le type de base compatible le plus économique. Ceci est totalement transparent au programmeur. Par exemple, lorsqu'il trouve un nombre à virgule affectée à une constante, il lui spécifie le type `Real`. Ou encore si vous affectez un caractère à une constante, celle-ci sera de type `Char` et non de type `String`. Mais il vous reste possible de forcer le type de la constante par les deux points `:` que vous connaissez déjà pour les variables. De plus, cette déclaration de type à une constante en plus de sa valeur est dans certains cas indispensable, si vous souhaitez leur donné une valeur qui n'est pas d'un type de base du compilateur (par exemple un ensemble ou un type que vous avez vous-même défini avant la constante).

Syntaxe :

```
Const identificateur : type-de-la-constante = valeur ;
```

Exemples :

```
Const Nmax : Longint = 60000 ;
Const Defaut : String = '*' ;
```

Note: Les constantes peuvent être de tout type sauf de type fichier.

On peut tout aussi bien affecter aux constantes des expressions dont le résultat sera évalué à la compilation.

Syntaxe :

```
Const identificateur = expression ;
```

Exemples :

```
Const Esc = Chr(27) ;
Const taux = (exp(10)-(3*Pi))/100 ;
Const nom = 'Cyber'+'Zoïde' ;
Const softi : Integer = Ord(length(nom)) ;
```

La déclaration des constantes se fait dans la partie déclarative du programme, avant le code exécutable (tout comme pour la déclarations des fonctions, procédures, variables). Pour êtres utilisables par tous les sous-programmes (procédures paramétrées...), les constantes doivent nécessairement êtres déclarée avant ces premiers.

```
Program exemple40 ;
Const nom='Bill';
Procedure saisielibre(Var toi:String);
Begin
   WriteLn('Entrez votre nom :');
   ReadLn(toi);
   If toi='' Then toi:=nom;
End;
Procedure saisiecontrainte(Var toi:String);
Begin
   Repeat
   WriteLn('Entrez votre nom :');
   ReadLn(toi);
   Until toi<>'';
End;
Var toi:String;
BEGIN
   saisielibre(toi);
   saisiecontrainte(toi);
END.
```

Cet *exemple40* montre quelle peut être l'utilisation d'une constante. Ici la procédure *saisielibre* permet à l'utilisateur de ne pas dévoiler son nom, la variable *toi* prenant une valeur par défaut qui est celle de la constante *nom*. Quant à la procédure *saisiecontrainte* elle ne laisse guère le choix à l'utilisateur : il doit absolument entrer un nom pour poursuivre le programme.