Maitrise Biologie des Populations et des Ecosystèmes

Module Bioinformatique & Modélisation

Tutorial de programmation en langage Pascal

Patrick Coquillard
Audrey Robert

2002

Sommaire

1. Avant propos : on ne s'affole pas ! *4*
2. Généralités *5*
I. Qu'est-ce qu'un programme ? **5**
II. A quoi sert un programme ? **5**
III. Comment fabrique-t-on un programme ? **5**
IV. Comment se passe l'exécution d'un programme ? **6**
3. Construction d'un programme *6*
4. L'interface Turbo Pascal 7 de Borland *10*
5. Les entrées - sorties à l'écran *11*
6. Les affichages à l'écran *12*
7. Variables, formats et maths… *13*
I. Déclaration **13**
II. Différents types de variables **14**
III. Prise de valeurs **14**
IV. Fonctions prédéfinies **15**
V. Emplois **15**
VI. Opérations **15**
VII. Format d'affichage **16**
8. Les instructions conditionnelles *16*
I. If ... Then ... Else **16**
II. Case ... Of ... End **17**
9. Les instructions de boucle (ou structures répétitives) *18*
I. For ... : = ... To ... Do **18**
II. For ... : = ... DownTo ... Do **18**
III. Repeat ... Until **19**
IV. While ... Do **19**
V. Arrêts de boucle **20**
10. Procédures et fonctions *21*
I. Procédure simple **21**
II. Variables locales et sous-procédures **22**
III. Passage de paramètres par valeur **22**
IV. fonctions **23**
V. Passage de paramètres par adresse (procédures et fonctions) **24**
11. Caractères et chaînes de caractères *26*
I. Chaînes de caractères **26**
II. Caractères seuls **27**

III. Table des caractères ASCII **28**

12. Génération de nombres aléatoires ***29***

13. Tous les types… ***30***

I. Type simple **30**

II. Type structuré (encore appelé enregistrement ou `record`) **31**

III. Type intervalle **33**

IV. Type énuméré **34**

V. Enregistrement conditionnel **35**

14. Programmation avancée (pour les mordus… mais ça se complique !) ***37***

I. En savoir plus sur les tableaux **37**

II. Les pointeurs **40**

III. Gestion de la mémoire **45**

1. Avant propos : on ne s'affole pas !

Ce Tutorial en langage Pascal est destiné à vous guider dans vos premier pas dans la programmation… Il est difficile, et pour tout dire vain et inutile, de prétendre comprendre quoi que ce soit à la modélisation des phénomènes biologiques si l'on ne possède pas un minimum de compréhension (on n'a pas dit savoir faire !) de la programmation.
Or la modélisation a envahi (et cela n'est pas fini !) l'intégralité des champs de recherche en biologie/écologie. Bon nombre de résultats publiés sont, en fait, tirés de modèles. Sans cet outil il devient très difficile d'avancer. Les quelques exemples sur lesquels vous aurez à plancher vous convaincrons facilement si vous ne l'êtes pas déjà.

Revenons à notre propos. Il n'est pas question pour vous d'apprendre à programmer comme un « Pro », mais seulement d'en apprendre et comprendre les rudiments pour pouvoir aborder quelques problèmes simples de modélisation (et surtout être en mesure de critiquer les modèles et leurs résultats).

Alors, il y a 3 situations :

1. Vous n'avez jamais programmé quoi que ce soit (sauf le four ou la machine à laver…).
   Lisez les premiers chapitres, essayez simplement d'en comprendre les exemples. Arrêtez votre lecture à la fin du chapitre « Procédures et fonctions ». Avec l'aide d'un(e) ami(e) expérimenté(e), installez le Pascal 7 sur votre ordinateur (si vous en avez un) et essayez d'exécuter quelques exemples très simples, notamment en mode « pas à pas ».

2. Vous avez déjà programmé un peu (en Basic par exemple). Le Pascal va vous ouvrir les yeux. Lancez vous jusqu'au chapitre «Programmation avancée » (page 33). Mais pas plus. Installez Pascal 7, compilez et exécutez les exemples, créez vos propres programmes.

3. Vous êtes accro. Allez-y carrément, jusqu'au bout.

Bon courage

2. Généralités

I. Qu’est-ce qu’un programme ?

Un programme est une suite d’“instructions” (ou ordres) écrites dans un langage interprétable et exécutable par une machine (ordinateur). On parle souvent d’“exécutable” en lieu et place de programme.

II. A quoi sert un programme ?

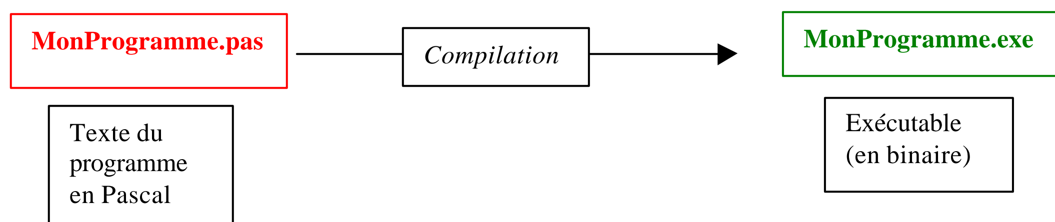
Les applications sont innombrables. L’objet des programmes va de l’exécution de taches élémentaires (opérations sur les nombres, trier des objets tels que des mots….) à des gestions très complexes de données, l’exécution de calculs très complexes (les jeux vidéo et les simulations en général comptent parmi ceux-ci), la prédiction météorologique, la direction de robots, etc… Les modèles biologiques et écologiques comptent parmi les programmes les plus complexes.

III. Comment fabrique-t-on un programme ?

Les premiers programmeurs formaient leurs instructions directement en langage "binaire" (une ligne par instruction). Il s’agit d’instructions formées par la combinaison de 4 caractères (4 bits), chaque bit ne pouvant prendre que la valeur 0 ou la valeur 1) directement « compréhensibles » par la machine. Cela ressemblait à ceci :

0000 *début du programme*
…
1001 *suite d’instructions*
1100
…
1111 *fin du programme*

Depuis, les choses ont bien évolué : les langages de programmation "ressemblent" au langage courant. L’utilisateur écrit le texte du programme (MonProgramme.pas) dans son langage préféré (le notre est le **Pascal**) au moyen d’un éditeur de texte quelconque (Word convient parfaitement). Ensuite, il lance un programme spécialisé (le "compilateur") qui traduit le code écrit dans un langage clair et compréhensible par tous en une suite d’instructions en code "binaire" que la machine pourra "comprendre". En réalité, le compilateur vérifie l’orthographe et la syntaxe du code écrit par l’utilisateur, définit les réservations mémoire nécessaires et traduit le programme en un "exécutable" en binaire (MonProgramme.exe). Un exemple de la structure d’un fichier .EXE est donné en page 45. L’utilisateur peut ensuite faire exécuter son programme par l’ordinateur en lançant l’"exécutable" (en tapant au clavier la ligne suivante à l’invite de commande : C :> MonProgramme)



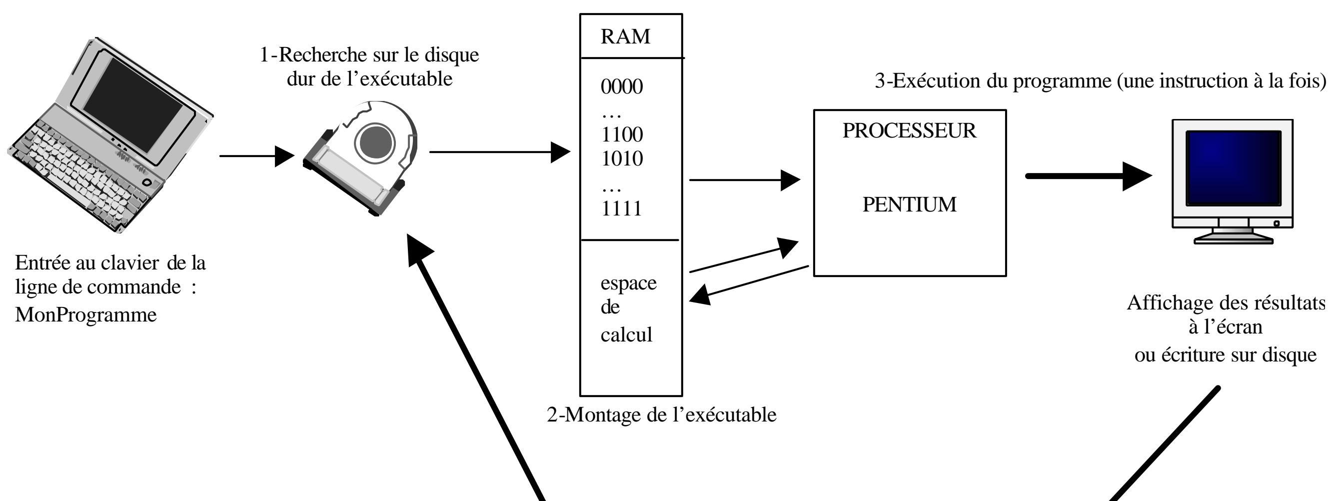
Si MonProgramme.exe ne contient aucune erreur de logique (division par zéro, actions impossibles, etc…), il s’exécutera parfaitement et donnera les résultats que vous attendiez. Sachez qu’un programme peut parfaitement fonctionner mais donner des résultats aberrants (cela ne gène nullement l’ordinateur). Ce n’est pas parce qu’un calcul est effectué par un ordinateur qu’il est exact. Exemple : la première fusée Ariane V a explosé parce que le

programme de calcul de la trajectoire de l'engin à partir des données fournies par les gyroscopes était celui élaboré pour Ariane IV avec d'autres caractéristiques (cela a coûté quelques millions d'Euros, et quelques têtes d'informaticiens sont tombées…).

IV. Comment se passe l'exécution d'un programme ?

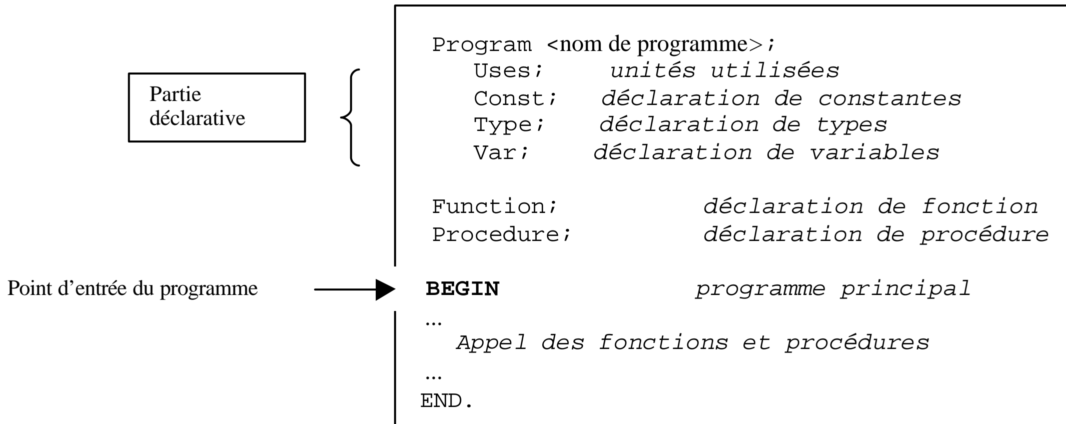
A l'état de repos, le programme (MonProgramme.exe) est stocké sur le disque dur. L'ordinateur est en attente, c'est-à-dire qu'un programme particulier appelé Operating System (OS) guette la moindre de vos actions au clavier ou à la souris. Vous connaissez au moins un O.S. : Windows ; DOS est son ancêtre mais peut toujours être utilisé sur les ordinateurs actuels. Lorsque l'ordinateur est en attente, le processeur (le fameux pentium ou autre) est en fait en train d'exécuter l'OS. Cette exécution utilise une mémoire spécialisée dans l'exécution de programmes (logiciels Word ou Excel, OS, programmes personnels …) : la RAM (Random Acces Memory, voir aussi en page 45 et suiv.).
Lorsque l'on entre la commande : <MonProgramme>, DOS va interpréter cet ordre comme :

1. chercher l'exécutable MonProgramme .exe, sur le disque dur (unité de stockage des données, appelée C : ou D :)
2. le recopier en mémoire RAM (s'il est trop gros le couper en petits morceaux… On dit aussi le "monter" en RAM).
3. le faire exécuter par le processeur.
4. reprendre la main en fin d'exécution et attendre une nouvelle commande.



3. Construction d'un programme

Après ces quelques préliminaires nous allons aborder l'écriture d'un programme proprement dit. Mais tout d'abord quelques règles élémentaires.
L'architecture standard d'un listing (on dit aussi communément « code » ou « source ») en Pascal s'écrit ainsi :



A noter :

- Un nom de programme respecte les règles liées aux identificateurs (cf plus bas) et ne peut pas contenir le caractère point "."
- Le compilateur Pascal ne différencie pas les majuscules des minuscules. Il est indifférent d'écrire begin ou Begin, voire begIn ou begiN…
- 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 excéder 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. Les chiffres sont acceptés hormis en première place.
- N'hésitez pas à insérer des commentaires dans votre code (c'est à dire du texte ignoré par le compilateur), 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* }
  ou bien ainsi :
  (* *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.
- Instaurez systématiquement une indentation. Ceci consiste en un décalage égal en début de ligne pour des instructions de niveau équivalents. Les exemples de programme qui illustrent ce cours vous guideront.

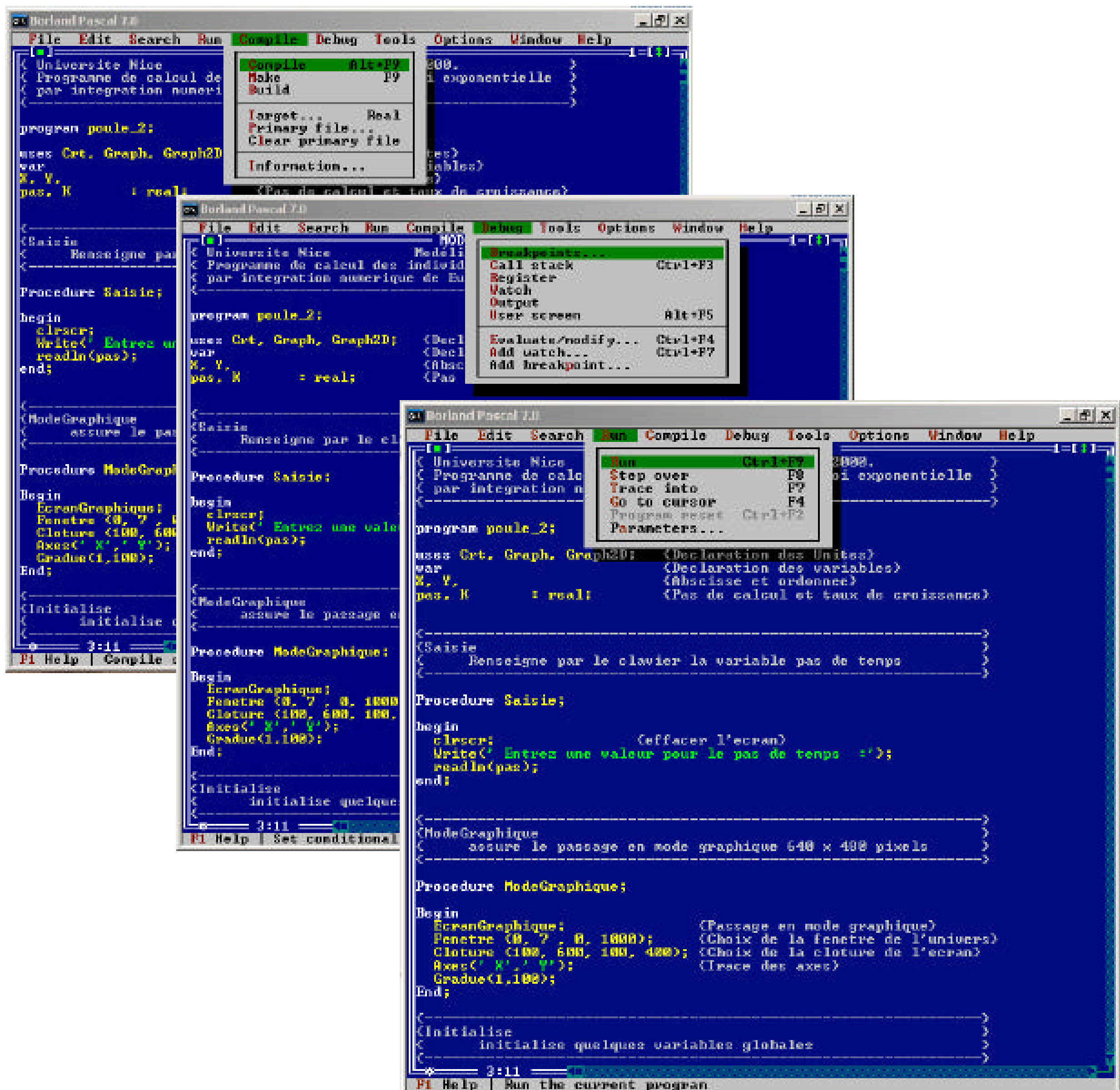
Exercice : Observez l'indentation et les commentaires du programme "poule". Retrouvez les mots réservés du Pascal standard. D'autres n'apparaissent pas dans la liste, ils sont propres à **Turbo Pascal** (élaboré par la société Borland). Essayez de comprendre ce que fait ce programme (vous le découvrirez en lisant la suite de ce guide). Notez bien que la présence de commentaires ne gène pas la compilation. Simplement, le compilateur Pascal ignorera tout texte situé entre accolades. (en gris le BEGIN point d'entrée du programme)

```
{-------------------------------------------------------------}
{ Maitrise MBPE . 2002.                                       }
{ Programme de calcul des individus selon une loi exponentielle}
{ et par intégration numérique.                               }
{-------------------------------------------------------------}
PROGRAM POULE;
uses Crt, Graph, Graph2D; {Déclaration des Unités}
var                       {Déclaration des variables}
       X, Y,              {abscisse et ordonnée}
       pas, K   : real;   {pas de calcul et taux de croissance}

{-------------------------------------------------------------}
{Saisie                                                       }
{     Renseigne par le clavier la variable pas de temps       }
{     NB : La vidéo est toujours en mode texte...             }
{-------------------------------------------------------------}
Procedure Saisie;
begin
  clrscr;                  {effacer l'écran : clear screen}
  Write(' Entrez une valeur pour le pas de temps  : ');
  readln(pas);
end;

{-------------------------------------------------------------}
{Initialise                                                   }
{      initialise quelques variables globales                 }
{-------------------------------------------------------------}
Procedure Initialise;
begin
  X:= 0;                          { Initialisation des variables}
  Y:= 10;
end;

{-------------------------------------------------------------}
{ModeGraphique                                                }
{     assure le passage en mode graphique 640 x 480 pixels}
{-------------------------------------------------------------}
Procedure ModeGraphique;
Begin
  EcranGraphique;               {Passage en mode graphique}
  Fenetre (0, 7 , 0, 1000);     {Choix de la fenetre de l'univers}
  Cloture (100, 600, 100, 400);{Choix de la cloture de l'ecran}
  Axes('X','Y');                {Tracé des axes}
End;
```

```
{-------------------------------------------------------------}
{Euler                                                        }
{     Fonction de calcul par intégration numérique            }
{        renvoie un entier en sortie                          }
{-------------------------------------------------------------}
Procedure Euler;
var
  deltay     : real;              {L'incrément par génération}
begin
  DeplaceEn(0,10);
  repeat
    X      := X + pas ;            {Calculs...}
    deltay := y * K * pas;
    Y      := Y + deltay;
    Tracevers(X,round(Y));
  until (X > 7);
end;
{-------------------------------------------------------------}
{Exponentielle                                                }
{     Calcul exact au moyen du modèle analytique              }
{-------------------------------------------------------------}
procedure Exponentielle;
begin
    Y := 10 * exp(K * X);
    DeplaceEn(X,Y);
    repeat
      X := X + 0.1;                {Calculs...}
      Y := 10 * exp (K * X);
      TraceVers (X,Y);
    until (X > 7);
end;
{-------------------------------------------------------------}
{              LE PRINCIPAL DU PROGRAMME                      }
{-------------------------------------------------------------}
BEGIN                        {debut du programme}
K := (0.44 * 3);             {le taux de croissance}
   Saisie;                   {choix du pas de temps}
   ModeGraphique;            {change de mode d'affichage}

   Setcolor(red);
   Initialise;               {Initialisation des variables}
   Exponentielle;

   Setcolor(green);
   Initialise;
   Euler;

   repeat until KeyPressed;  {Boucle pour contempler indéfiniment...}
   EcranTexte;               {Retour au mode texte avant de quitter}
END.                         {Fin du programme}
```

4. L’interface Turbo Pascal 7 de Borland

- 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 omis 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'aller 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 apparaît 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 apparaît en bas de page. Cliquez sur **Add** et tapez le nom de la variable dont vous désirez connaître la dernière valeur.



Trois menus de l’éditeur Borland Pascal

5. Les entrées - sorties à l'écran

⇨ L'intruction **Write** permet d'afficher du texte et de laisser le curseur à la fin du texte affiché. Cette instruction permet d'afficher des chaînes de caractères n'excédant pas 255 signes ainsi que des valeurs de variables, de constantes, de types... Le texte doit être entre apostrophes. 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 instruction doit être suivie d'un point-virgule.

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

⇨ L'instruction **WriteLn** est semblable à la précédente à la différence près que le curseur est maintenant renvoyé à la ligne suivante après écriture.

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

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

Syntaxe :
```
Read (variable) ;
```

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

Syntaxe :
```
Readln (variable1, variable2) ;
Readln ;
```

```
Program exemple1;
   Var patronyme : String ;      {déclaration d'une variable de type chaîne de caractères}
   BEGIN
      Write('Entrez votre nom : ') ;{demande de saisie du nom}
      Readln(patronyme) ;        { saisie de votre nom… et stockage dans la variable patronyme}
      Writeln('Votre nom est ', patronyme) ;      {affichage du contenu de patronyme}
      Readln ;              {permet d'attendre avant la fin du programme}
   END.
```

Ce programme exemple1 déclare tout d'abord la variable nommée patronyme comme étant une chaîne de caractère (String). Ensuite, dans le bloc programme principal, il est demandé à l'utilisateur d'affecter une valeur à cette variable. Ensuite, il y a affichage de la valeur de la variable et attente que la touche entrée soit validée (ReadLn).

⇨ L'équivalent de l'instruction ReadLn est **ReadKey** qui affecte à une variable de type **Char** la valeur ASCII du caractère frappé au clavier. Dans l'exemple (x :=ReadKey) si on appuie la touche <a>, la variable x prendra la valeur 97.

Syntaxe :
```
x := ReadKey ;
```

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

Syntaxe :
```
Repeat
...
```

```
instructions
...
Until KeyPressed ;
```

```
Program exemple2 ;   {écrit la racine carrée des nombres compris entre 10 et 2000}
Uses crt ;
   Var i : integer ;
   Const bornesup = 2000;
   BEGIN
        i := 10;
        Repeat
              WriteLn(sqrt(i)) ;      {écrire la racine carrée de (i)}
               Inc(i) ;               { incrémenter i de 1 ( c'est à dire i := i + 1)}
        Until (i = bornesup) or KeyPressed ;
   END.
```

Ce programme exemple2 répète une boucle jusqu'à ce que soit `i` ait atteint la bornesup, soit l'utilisateur ait appuyé sur une touche du clavier. L'instruction `Inc(a,n);` incrémente de la valeur n la variable a, et `Inc(a)` incrémente `a` de 1. `a` est nécessairement de type `integer`.

6. Les affichages 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. Ayez toujours à l'esprit que la résolution de l'écran texte, en Turbo Pascal, est de 80 colonnes par 25 lignes (ou 42, c'est une option), en 16 couleurs.

⇨ **ClrScr ;**
Efface 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 ;**
Insère une ligne vide à la position courante du curseur.

⇨ **ClrEol ;**
Efface une ligne à l'écran à partir de la position courante du curseur. Note : la position du curseur n'est pas modifiée.

⇨ **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 ;**
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 et WhereY de la ligne.

⇨ **HightVideo ;**
Sélectionne 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 ;**
Revenir au mode normal de couleur de texte, c'est-à-dire pour pouvoir utiliser indifféremment les couleurs vives et ternes.

⇨ **TextMode (x) ;**
Sélectionne un mode spécifique d'affichage du texte. x est la valeur-code du mode désiré.

7. Variables, formats et maths…

I. Déclaration

⇨ 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. (*duplicate identifier*).
⇨ 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. Rappelons que Turbo Pascal ne différencie aucunement les majuscules des minuscules et qu'un chiffre ne peut pas être placé en début de nom de variable.

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 trois façons :

⇨ Au tout début du programme avec la syntaxe <**Var** *nom de la variable* **:** *type* **;**> elles seront alors lisibles et utilisables par le programme dans son intégralité (sous-programmes, fonctions, procédures...). Ces variables sont dites *globales* parce que leur portée (« scope » en anglais) dans le programme est globale.
⇨ Au début d'une procédure avec la syntaxe précédente. Elles ne seront valables que dans la procédure et sont donc *locales*.
⇨ 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). Elles sont aussi locales.

II. Différents types de variables

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)		

III. Prise de valeurs

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

Syntaxes :

Y := 1998 ;

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

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).

IV. Fonctions prédéfinies

Fonctions mathématiques de base en Pascal

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 conversion si nécessaire. De plus, on peut voir que les fonctions tangente, factorielle, hyperboliques… n'existent pas, il faudra donc créer de toute pièce les fonctions désirées (Chapitre Fonctions).

V. Emplois

Les variables peuvent êtres utilisées dans de nombreux emplois :
- Pour le stockage temporaire de valeurs ou résultats de calculs,
- Pour des comparaisons dans une structure conditionnelle (voir chapitre 8).
- Pour l'affichage de résultats (voir chapitre 5, 6 et 7).
- Pour le dialogue avec l'utilisateur du programme (voir chapitre 5).
- Pour exécuter des boucles (voir chapitre 9)...

VI. 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 ensembre ordonnée
`Succ(x)`	Fonction	intervalle ou énuméré	Renvoit le successeur de la variable x dans un ensembre 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

VII. Format d'affichage

Le format (le nombre de signes) d'une variable de type real peut être spécifié selon les besoins. Le format prend la forme générale : `Write`(*nombre* : *TailleMini* : *NbDecimales*) ;

- ⇨ Lors de son affichage : WriteLn (nombre : 5) ; pour afficher 5 espaces devant le nombre s'il est entier.
- ⇨ WriteLn (nombre : 10) ; pour afficher un nombre réel au moyen de 10 signes maximum(9 caractères plus le point)
- ⇨ Lors de son affichage : WriteLn (nombre : 10 : 2) ; pour ne mettre aucun espace avant mais pour n'afficher que 10 caractères et 2 décimales (un réel en possède bien plus). Si l'ensemble fait moins de 10 signes, le nombre de caratcères spécifié sera atteint par ajout d'espaces en tête du nombre.

Pour pouvez appliquer ce format pour tous les autres types de variable de manière générale si vous ne stipulez 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.

8. Les instructions conditionnelles

I. If ... Then ... Else

Cette instruction se traduit simplement par : SI … ALORS … SINON …

```
Program exemple3a ;
  Var nombre : integer ;
BEGIN
   Write('Entrez un entier pas trop grand : ');
   Readln(nombre) ;
   If nombre < 100
     then writeln(nombre,'est inférieur à cent.') {pas de « ; » ici !}
     else writeln(nombre,'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 nombre : integer ;
 BEGIN
    Write('Entrez un entier pas trop grand : ') ;
    Readln(nombre);
    If (nombre < 100) then
      begin
         writeln(nombre, ' est inférieur à cent.') ;
      end          {pas de point-virgule, l'instruction n'est pas terminée !}
      else
         begin
            writeln(nombre, ' est supérieur ou égale à cent.') ;
         end ;       { l'instruction if then else est terminée}
END.
```

Ce test ne représente qu'une seule instruction !

Ce programme exemple3b fait strictement la même chose que le 3a mais sa structure permet d'insérer plusieurs autres instructions 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 terminerait 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 ;                      { initialise le générateur aléatoire }
   i := random(100) ;               { tire un nombre [0 ;100[ }
   if (i < 50)
      then  writeln ( i, ' est inférieur à 50.')
          else
            if (i < 73)
               then writeln ( i, ' est inférieur à 73 et supérieur à 50')
               else
                  writeln ( i, ' est supérieur ou égale à 73 et inférieur à 100')
               end ;
          end ;
  END.
```

A noter :

1. Les deux **end;** sont ici facultatifs. En effet, le plus interne correspond à une seule instruction un point virgule en fin de ligne aurait suffit. Quant au plus externe il correspond lui aussi à une seule instruction de type <if then else> (encadrée par le rectangle grisé), il est donc inutile. Mais cette syntaxe est correcte et a le mérite de la clarté.
2. Une excellente habitude (même si syntaxiquement inutile) consiste en l'insertion de parenthèses dans le test : `if (`*condition*`) then` …. On y gagne beaucoup en lisibilité, et on se prépare à la programmation en C et Java...

II. Case ... Of ... End

Cette instruction compare la valeur d'une variable de type entier ou caractère (de manière générale un scalaire) 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('le bel âge') ;
      70..99  : writeln('hors d'age')
      Else writeln('a voir…') ;
      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'omettre le point-virgule avant un `Else`.

Note : On peut effectuer un test de plusieurs valeurs en une seule ligne en les séparants par des virgules.On peut alors effectuer un même traitement pour plusieurs valeurs différentes. Ainsi la ligne :

0..17 : writeln('le bel age') ;

peut devenir :

0..10, 11..17 : writeln('le bel age') ;

ou encore :

0..9, 10, 11..17 : writeln('le bel age') ;

ou même :

0..17, 5..10 : writeln('le bel age') ;

9. Les instructions de boucle (ou structures répétitives)

I. For ... : = ... To ... Do ...

Cette instruction permet d'incrémenter une variable à partir d'une valeur initiale jusqu'à une valeur finale et d'exécuter une ou des instructions après chaque incrémentation. Les valeurs extrêmes doivent être des entiers (`integer`) ou des caractères de la table ASCII (`char`). De manière plus générale, les bornes doivent être des scalaires 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 inférieure est effectivement inférieure ou égale à celle de la borne supérieure. Le pas de variation est l'unité et ne peut pas être changé.

Syntaxe :
For variable **:=** *borne_inférieure* **To** *borne_supérieure* **Do** *instructions* ;

Autre Syntaxe :

For variable **:=** *born_ inférieure* **To** *borne_supérieure* **Do**
Begin
...
instructions ;
...
End ;

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

II. 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 précédemment.

Syntaxe :
For variable := *borne supérieure* **DownTo** *borne inférieure* **Do** *instruction* ;

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

III. Repeat ... Until ...

Cette boucle effectue les instructions placées entre deux bornes (`repeat` et `until`) *et évalue à chaque répétition une condition de type booléenne 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 (attention c'est dangereux…). 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 boucle. En l'absence de modification, la boucle ne se termine jamais (boucle infinie).

Syntaxe :
Repeat
...
instructions
...
Until variable condition valeur ;

```
Program exemple7 ;
Uses crt ;
Var i : integer ;
BEGIN
   i := 0 ;
   Repeat
      Inc ( i ) ;
      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 procédure `Inc` permet d'incrémenter une variable d'une certaine valeur x. `Dec` permet au contraire de décrémenter une variable d'une certaine valeur. Ces procédures permettent d'éviter ce type (lourd) de syntaxe :
`variable := variable + x` ou bien `variable := variable - x.`

Syntaxe :
Inc (variable , nombre) ;
Dec (variable , nombre) ;

Inc(variable) et Dec (variable) sont respectivement équivalentes à Inc(variable, 1) et Dec (variable, 1)

IV. 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 (cela c'est sécurisant…). De plus, au moins une variable de l'expression d'évaluation doit être sujette à modification au sein de la boucle. En l'absence de modification, la boucle ne se termine jamais (boucle infinie).

Syntaxe :
While *variable* condition **Do** *instruction* **;**

Autre Syntaxe :

While *variable* condition **Do**
Begin
...
instructions
...
End ;

```
Program exemple8 ;
Var code : boolean ;
essai : string ;
```

```
Const levraicode = 'password' ;
BEGIN
   Code := false ; {la valeur false est affectée lors de la déclaration }
   While (code = false)  Do
      Begin
         Write ('Entrez le code secret : ') ;
         Readln (essai) ;
         If (essai = levraicode) then code := true ;
      End ;
END.
```

V. Arrêts de boucle.

Il est possible de terminer en cours d'exécution une boucle `For`, `While` ou `Repeat` grâce à l'instruction `Break` lorsque celle-ci est placée au sein de la boucle en question.

```
Program exemple9 ;
Var i,  : Integer ;
       choix : char ;
BEGIN
   Writeln('programme de choix d'un nombre entre 0 et 10')
   i := 0 ;
   Repeat
      Writeln (i, ' :  ?') ;        {affiche la valeur de i}
      Readln(choix) ;               {on récupère le choix de l'utilisateur}
      If ((choix = 'o') OR (choix = 'O')) then    {si c'est « Oui »}
        Begin
           Write ('c''est un bon choix') ;
           Break ;               {on provoque la sortie du Repeat … until}
        End ;
      Inc(i) ;
   Until i >9 ;                  {sinon, la boucle s'effectue complètement}
END.
```

L'instruction Continue permet dans l'une de ces trois boucles de provoquer immédiatement l'itération suivante.

```
Program exemple9b ;
Var i,  : Integer ;
       choix : char ;
BEGIN
   Writeln('recherche des nombres divisibles par 3 et par 5')
   i := 0 ;
   For (i :=0) to 100 do
      Begin
         Writeln (i);           {affiche la valeur de i}
         If ((i MOD 3) <> 0)
             Then continue     {i non divisible par 3, on passe au suivant}
             Else              {i divisible par 3, on teste la division par 5}
             If ((i MOD 5) = 0) then
                 Writeln(i, ' est un des nombres recherchés') ;
       end;     {sinon, la boucle s'effectue complètement}
END.
```

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

```
Program exemple9c ;
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.
```

10.Procédures et fonctions

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.

I. 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 (variables dites globales). 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
...
instructions **;**
...
End ;
BEGIN
nom de procédure ;
END.

```
Program exemple10 ;
Uses crt ;
Var a, b, c : real ;
   Procedure Maths ;
      Begin
          a := a + 10 ;
          b := sqrt(a) ;
          c := sin(b) ;
       End ;
BEGIN
   Clrscr ;                          {nettoyer l'écran}
```

```
   Write('Entrez un nombre :') ;
   Readln(a) ;
   Maths ;                          {appel de la procédure}
   Writeln(c : 10 :5);              {affichage du résultat}
   Repeat Until (keypressed) ;      {pour regarder le résultat…}
END.                                { appuyer sur une touche pour quitter le programme}
```

Ce programme appelle une procédure appelée maths qui effectue des calculs successifs. Cette procédure est appelée depuis une boucle qui ne se s'arrête 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 en calcule le carré `sqrt(a)` du nombre ainsi obtenu, et enfin, on cherche le sinus `sin(b)` de ce dernier nombre.

II. Variables locales et sous-procédures

Une procédure peut avoir ses propres variables locales qui seront réinitialisées à chaque appel (voir la Pile en page 46). 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 ceux des globales si ces dernières ont été déclarées avant la procédure.
- ⇨ 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 cette procédure principale est en cours.
- ⇨ Un sous-procédure ne peut appeler d'autres procédures ou fonctions que si ces dernières font partie du programme principal ou de la procédure principale qui contient la sous-procédure.

Syntaxe :

Procedure *nom de procédure* ;
Var variable : type ;
    Procedure *nom de sous-procédure* ;
    Var variable : type ;
        Begin
        ...
        End ;
Begin
    ...
    instructions ; {y compris l'appel à sous-procédure}
    ...
End ;

III. Passage de paramètres par valeur

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 déclarées 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 n'existant que 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. Ce type d'appel est appelé ***passage de paramètres par valeur***. 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*(variables : types) ;
Begin
...
instructions ;
...
End ;
BEGIN
nom de procédure(variables) ;
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` car le passage d'un tableau en tant que type `Array` à une procédure est impossible.

```
Program exemple11 ;
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);     {On passe nombre à Maths}
        WriteLn(nombre) ;   {Maths n'a pas modifié nombre}
        ReadLn ;
END.
```

Ce programme 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'existence de la variable nombre, ainsi, pour qu'elle puisse l'utiliser, il faut le lui passer en paramètre ! *Attention,* `nombre` *n'est pas modifié... En fait, c'est une copie de* `nombre`*, affectée à* `param` *qui est modifiée. Prendre garde aussi à la compatibilité de type entre* `nombre` *et* `param`.

IV. 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 :

Paramètre d'entrée

⇨ Function nom de fonction (variable : type) : type ;
Var déclaration de variables locales ;
Begin
...
instructions ;
...
nom de fonction := une valeur ; {on retourne la valeur de sortie}
End ;

Paramètre de sortie

Paramètre de sortie (pas de paramètre d'entrée)

⇨ Function nom de fonction : type ;
Var déclaration de variables locales ;
Begin
...
instructions
...
nom de fonction := une valeur ;
End ;

```
Program exemple12 ;
Uses crt ;
Var
resultat, x, n : integer ;

Function exposant(i,j : integer) : integer; {deux entiers en entrée, un entier en sortie}
Var i2 , a : integer ;
  Begin
        i2 := 1 ;
        For a := 1 To j Do i2 := i2 * i;
        exposant := i2 ;
  End ;
BEGIN
  Write ('Entrez un nombre : ');
  Readln (x);
  Write('Entrez un exposant : ');
  Readln (n);
  resultat := exposant(x,n); {on récupère la valeur calculée par exposant pour les paramètres x et n}

  Writeln ( resultat ); {et on l'affiche}
  Readln;               { on  attend l'appui d'une touche}
END.
```

V. Passage de paramètres par adresse (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`.
Ce type d'appel de est appelé ***passage de paramètres par adresse.***

```
Program Exemple13 ;
Uses Crt ;

Procedure Saisie ( var nom : String ) ;  {passage par adresse}
   Begin
       Write('Entrez votre nom : ') ;
       ReadLn(nom) ;
   End ;

Procedure Affichage ( info : String ) ;  {passage par valeur}
  Begin
       WriteLn('Voici votre nom : ', info) ;
  End ;

Var chaine : String ;

BEGIN
  ClrScr ;
  Saisie(chaine) ;
  Affichage(chaine) ;
END.
```

Ce programme 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 *chaîne* (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 envoi, 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 chaîne du programme principal prend la valeur de la variable nom de la procédure. Ensuite, on envoi 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 êtres 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 que l'on veut que leurs changements de valeur affectent les variables globales, alors on devra placer devant chaque déclaration de types différents un `Var`.

Syntaxes :

```
Procedure identifiant(Var v1, v2 : type1 ; v3 : type1) ;
Begin
...
End ;

Procedure identifiant(Var v1 : type1 ; Var v2 : type2) ;
Begin
...
End ;
```

11.Caractères et chaînes de caractères

I. 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 caractère 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 caractères, 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:='Duchmol';
  nom[2]:='U';
  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 "U" majuscule, puis spécifie que la variable ne contient plus que 4 caractères. Ainsi la valeur de la variable nom est devenue : DUch. Mais après, on dit que la variable nom a une longueur de 28 caractères et on s'aperçoit 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).

II. 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);
```

III. Table des caractères 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 fontion renvoit un caractère Char correspondant au caractère d'indice x dans la table ASCII.
Exemple :
k := Chr(64) ;

⇨ **Ord (y) ;**
Cette fontion renvoit l'indice (en byte) correspondant au caractère y dans la table ASCII. C'est la fonction réciproque de Chr.
Exemple :

i := Ord('M') ;

12.Génération de nombres aléatoires

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 est 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 aleatoire;
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);      {test prend la valeur 1 si x = y, 0 sinon}
     If (test)             {si test a la valeur true (c'est a dire 1) }
        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 (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 de type `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`.

13.Tous les types…

Il est possible de créer de nouveaux types de variables sur Turbo Pascal 7.0 en plus de ceux prédéfinis (`integer`, `real`, `string` etc…). 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 le chap. 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.

I. 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 utilise un nouveau type appelé *chaine* qui sert à déclarer la variable *nom*.

II. Type structuré (encore appelé enregistrement ou `record`)

On peut être amené à utiliser des types structurés car dans une seule variable on peut réussir à caser des sous-variables nommées **champs**.

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 := 'Duchmol' ;
        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 := 'Duchmol' ;
   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 := 'Duchmol' ;
      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 := 'Duchmol' ;
      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*.

III. 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.
```

IV. 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). 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'ont 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*.

V. 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. Un tel 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 champ 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.

14.Programmation avancée (pour les mordus… mais ça se complique !)

I. En savoir plus sur les 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 créant un tableau de tableau. Mais cette technique n'est ni la plus jolie, ni la plus pratique. 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 grace 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 élements 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'utliser en même temps la syntaxe des tableaux et des types complexes.

Syntaxe :

```
tab[5].nom := 'Ducretin';
```

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

II. Les 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. **(voir la gestion de la mémoire ci-après, p. 45)**
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 ("*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 renvérait 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 ("*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 quatre 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 conbien 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 bien mais comment fait-on pour accéder au contenu de la variable pointée par le pointeu ? 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* contient donc une adresse mémoire et *P^* contient 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 :

```
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 BigOne : PMatrice;
```

On a donc une variable *BigOne*(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 *BigOne* est d'un type *PMatrice* pointant vers le type *Tab2D*. Donc la taille de *BigOne* sera de 4 octets puisque contenant une adresse mémoire et *BigOne*^ (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 : *BigOne*`^[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 BigOne: PMatrice;
   i, j, x : Integer;
BEGIN
New(BigOne);
For i:=1 to Max Do
   Begin
   For j:=1 to Max Do BigOne ^[i,j] := i+j;
   End;
x := BigOne ^[Max,Max] * Sqr(Max);
WriteLn(Cos(BigOne ^[Max,Max]));
Dispose(BigOne);
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 exactement 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 : On 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 sur 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 sur 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 sur 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 remarquez 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 supérieure 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));      {ici la fonction SizeOf renvoi la taille de la variable Personne}
   For i:=1 To N Do ReadLn(Tab^[i].nom);
   FreeMem(Tab,N*SizeOf(Personne));
   END.
```

III. Gestion de la mémoire

a. 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 se font la part belle, la mémoire allouée aux pointeurs (très gourmands en mémoire) est variable et celle destinée aux variables locales est assez restreinte.

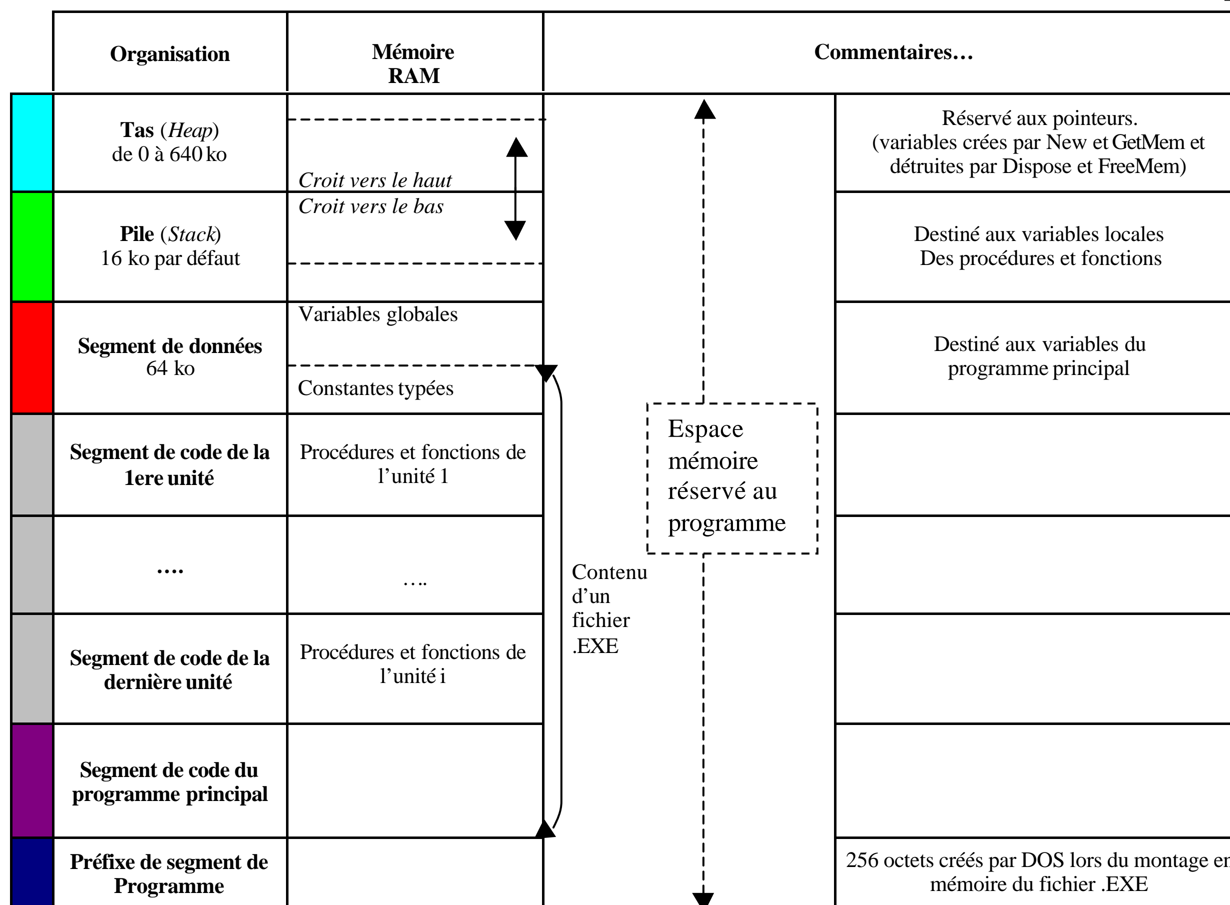
Mais il est toutefois possible de modifier manuellement une telle organisation de la mémoire afin, par exemple, de privilégier la **Pile** grâce au commentaire de compilation suivant : `{$M` *n1*, *n2*, *n3*`}`. Ce type de commentaire est destiné au compilateur *Borland Pascal* qui inscrira les informations spécifiées dans le programme compilé. Un commentaire de compilation se présente entre accolades comme n'importe quel autre commentaire, mais un signe dollar "`$`" 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 déclarez 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 renvoi 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 ("*Différents types de variables*") qui vous indique la taille de chaque type de variable.

Lorsque l'on veut travailler avec de très grandes quantités de données (variables globales stockées dans le segment de données) et dépasser la valeur fatidique de 64 Ko, il faut alors travailler en mode protégé (DPMI), mais c'est une autre histoire….



Occupation de la mémoire par un programme Turbo Pascal. Les procédures et fonctions étant destinées à des calculs intermédiaires, elles n'ont guère besoin de grande quantité de ressource mémoire (en principe…). Quant aux pointeurs, ils sont destinés à la manipulation d'une grande quantité de données. Remarquez qu'il n'est pas possible de déclarer plus de 64Ko de variables globales dans un programme, ce qui constitue l'un des inconvénients majeur du Pascal (on atteint très rapidement cette taille). La taille limite de 64Ko (c'est-à-dire un segment de mémoire) est une vielle relique du DOS qui organisait la mémoire en segment de cette taille parceque les adresses des variables étaient stockées sur 16 octets. Soit, dans ces conditions, un maximum de 2^{16} -1 = 65535 (» 64 Ko) adresses différentes.

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

Dans le chap. ("*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 principal. 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 passée en paramètre).