Version Septembre 2016

Notions d'algorithmique et de programmation avec DELPHI

II^{e} B / I^{re} B

par :

MM. Hubert GLESENER, Jean-Claude HEMMER,

David MANCINI, Alexandre WAGNER

Partie « Graphiques » par MM. Ben Kremer et François Zuidberg

Table des matières

1 Introduction 5
1.1 Généralités 5
1.2 Cycle de développement 5
1.3 Types d'applications 5
1.4 Exemple d'un cycle de développement 6
2 Affectation 8
2.1 Entrées-sorties 8
2.2 Les opérations arithmétiques. 9
2.3 Variables 10
2.4 L'instruction readln 14
2.5 Les constantes 14
2.6 Exercices 15
2.7 Différents types d'erreurs 16
3 Structure alternative 17
3.1 Problème introductif 17
3.2 Syntaxe 18
3.3 Les conditions (expressions logiques) 19
3.4 Exercices 20
4 Structure répétitive 22
4.1 Introduction 22
4.2 Exemple 22
4.3 Terminaison d'une boucle 23
4.4 Boucles for 23
4.5 Exercices 24
4.6 Un algorithme efficace (facultatif) 25
4.7 Exercices 27
5 Fonctions et procédures 29
5.1 Les fonctions 29
5.2 Les procédures 31
5.3 Portée 33
5.4 Passage des variables dans une fonction ou une procédure 37
6 Les structures de données composées 40
6.1 Les tableaux 40
6.2 Les tableaux à deux dimensions 42
6.3 Les enregistrements 46
6.4 Le mot-clé type 48
7 Delphi 54

7.1 Introduction 54
7.2 Les fichiers utilisés en Delphi 54
7.3 L'approche Orientée-Objet 55
7.4 Passage Pascal – Delphi – un premier exemple 56
7.5 Calcul de la factorielle 58
7.6 Equation du second degré 61
7.7 Vérification du numéro de matricule 65
7.8 Calcul matriciel - utilisation du composant TStringGrid 68
8 La récursivité 73
8.1 Exemple 73
8.2 Définition : « fonction ou procédure récursive » 73
8.3 Etude détaillée d'un exemple 74
8.4 Fonctionnement interne 75
8.5 Exactitude d'un algorithme récursif 75
8.6 Comparaison : fonction récursive – fonction itérative 76
8.7 Récursif ou itératif ? 77
8.8 Exercices 78
9 Comptage et fréquence 79
9.1 Algorithme de comptage 79
9.2 Fréquence d'une lettre dans une liste 79
10 Recherche et tri 80
10.1 Introduction 80
10.2 Sous-programmes utiles 80
10.3 Les algorithmes de tri 81
10.4 Les algorithmes de recherche 86
11 Graphisme 88
11.1 Introduction 88
11.2 La surface de dessin 88
11.3 Les éléments graphiques 90
11.4 La souris 103
11.5 Le métronome (Timer) - facultatif 106
11.6 Impression de graphiques - facultatif 109

Première partie :

Cours de II^{e}

Applications en console

1 Introduction

1.1 Généralités

L'objectif de ce cours est d'apprendre l'art de programmer. Au départ un énoncé sera analysé. Il sera ensuite étudié et finalement transformé en un programme bien structuré.

La solution informatique du problème posé est appelée *algorithme*. Celui-ci est indépendant du langage de programmation choisi. Cependant nous devons opter pour un langage de programmation dans lequel l'algorithme sera traduit. Dans ce cours nous allons utiliser le langage *Pascal* que nous allons étendre ensuite à *Delphi*.

Un langage de programmation est un ensemble d'instructions qui permettent de décrire à l'ordinateur une solution possible du problème posé. Il existe une multitude de langages de programmation différents, chacun avec ses avantages et ses désavantages. Exemples : Fortran, Pascal, Ada, C, C++, C#, Java, …

Le langage *Pascal* est un langage de programmation évolué et polyvalent, dérivé de l'*Algol-60*. Il a été développé par Niklaus Wirth (*1934 -) au début des années 1970 à l'École Polytechnique Fédérale de Zurich. Au cours des années le langage *Pascal* a fortement évolué et la plus importante modification est sans doute l'incorporation de la notion de programmation orientée objet. *Delphi* pour sa part est la combination du langage *Object-Pascal,* un dérivé de *Pascal,* auquel on a incorporé des notions graphiques.

1.2 Cycle de développement

Pour développer efficacement un programme nous suivons les étapes suivantes :

- Analyse du problème :
  - Quelles sont les données d'entrée ?
  - Quelles sont les données à la sortie que le programme doit produire ?
  - Comment devons-nous traiter les données d'entrée pour arriver aux données à la sortie ?
- Établissement d'un algorithme en utilisant un langage de programmation compris par le système.
- Traduction du programme *Pascal* en langage machine à l'aide d'un compilateur.
- Exécution et vérification du programme.

1.3 Types d'applications

Il existe deux types d'applications différents : les applications *en console* et les applications à *base graphique*. Les applications en console se limitent à des entrées et des sorties purement textuelles. Les applications en console n'utilisent pas de graphismes. Ce type d'applications était usuel avant l'arrivée de *Windows*.

Les applications à base graphique nécessitent un environnement graphique comme par exemple *Windows* de Microsoft. Ce type d'applications est caractérisé par l'utilisation de fenêtres pour l'entrée et la sortie d'informations.

En II^{e} nous allons nous limiter aux applications en console. Ils se développent plus facilement que les applications à base graphique qui nécessitent des techniques de programmation plus évoluées.

1.4 Exemple d'un cycle de développement

Pour rédiger, compiler et déboguer[1] un programme *Pascal*, nous avons besoin d'un milieu de développement. Dans le cadre du cours de cette année, nous pourrions utiliser tout compilateur répondant aux spécifications du langage *Pascal*, comme par exemple *FreePascal* ou *Turbo Pascal*. Nous allons utiliser le milieu de développement *Delphi* de *Borland* qui permet de développer des applications du type console aussi bien que des applications à base graphique.

1.4.1 Énoncé du premier programme

Pour commencer nous voulons établir un programme qui affiche un simple message sur l'écran, à savoir : « Bonjour tout le monde ! ».

1.4.2 Analyse du problème

Il n'y a pas de données à l'entrée. Il n'y a pas de traitement de données. À la sortie, il y a un message à afficher : « Bonjour tout le monde ! ».

1.4.3 Programme Pascal

```
program Bonjour;
begin
  writeln('Bonjour tout le monde !')
end.
```

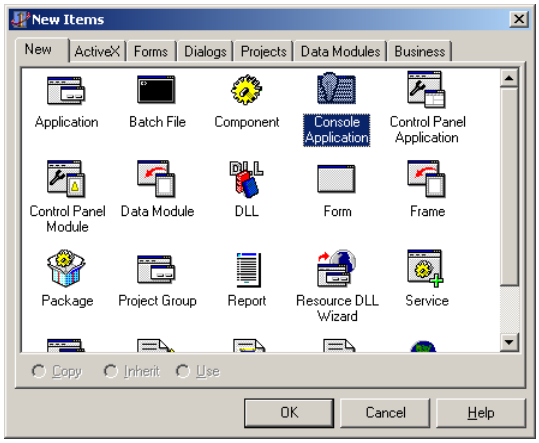
1.4.4 Milieu de développement Delphi

Avant de démarrer *Delphi*, il est impératif de créer pour chaque programme un sous-répertoire individuel. Tous les fichiers relatifs au programme sont alors sauvegardés dans ce sous-répertoire.

Créons donc un tel sous-répertoire que nous appellerons *Bonjour*.

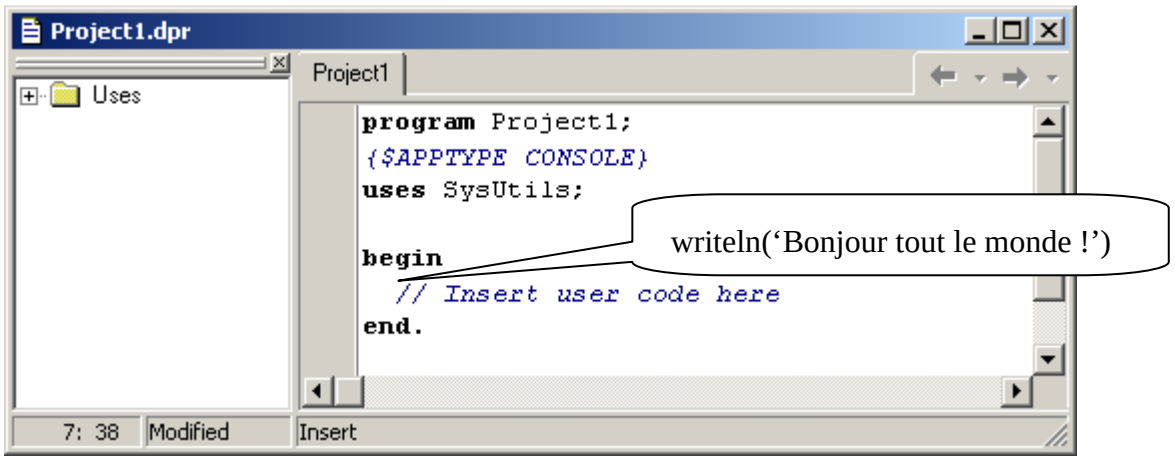
Nous lançons ensuite le programme *Delphi*. Il nous propose immédiatement les structures pour développer une application graphique. Or nous n'utilisons pas ce type d'applications pour l'instant, c'est pourquoi nous fermons ce projet à l'aide du menu : **File → Close All**.

Ensuite nous créons une nouvelle application de type console à l'aide du menu : **File → New**.



Un double clic sur l'option « Console Application » ouvre une nouvelle fenêtre qui reçoit le code de notre programme :

[1] Chercher à éliminer les erreurs.



Recopions ensuite le programme proposé plus haut. N'oublions pas de le sauvegarder à l'aide du menu **File → Save**.

L'extension pour les programmes *Delphi* est « dpr », qui est une abréviation de « delphi project ». Nous constatons alors que dans le texte «**`program`** Project1;», le mot «Project1» est remplacé par le nouveau nom du fichier mais sans l'extension.

1.4.5 Exécution et test du programme

Après avoir copié le programme, nous pouvons essayer de le compiler et de l'exécuter une première fois à l'aide du menu **Run → Run** (raccourci clavier : F9) ou en appuyant sur le bouton (semblabe au bouton « play » d'un magnétoscope) avec le petit triangle vert.

Si nous exécutons le programme, nous constatons qu'une fenêtre apparaît brièvement mais nous ne voyons pas son contenu. Cette fenêtre est supposée contenir les entrées et les sorties de notre programme. Cependant dans notre cas, nous n'apercevons pas la sortie, car la fenêtre disparaît immédiatement. Cette disparition provient du fait que si un programme atteint sa fin, sa fenêtre est immédiatement fermée.

C'est pourquoi **nous devons rajouter à la fin de chaque programme l'instruction *readln***, qui attend que l'utilisateur appuie sur une touche du clavier et permet ainsi de « geler » la fenêtre.

1.4.6 Version définitive du programme

```
program Bonjour;
{$APPTYPE CONSOLE}
begin
  writeln('Bonjour tout le monde !');   {affichage}
  readln;                               {attente}
end.
```

Les parties entre accolades ne font pas partie du programme proprement dit. Il s'agit normalement de commentaires que le programmeur peut ajouter pour expliquer le fonctionnement du programme. Ici, par contre, Delphi a ajouté des commentaires qu'il ne faut pas enlever. Le texte `{$APPTYPE CONSOLE}` en particulier indique au système qu'il s'agit d'une application en console et non pas d'une application Windows.[2]

[2] Pour transférer un ancien programme écrit en Pascal (Turbo Pascal, Free Pascal ou similaire) vers Delphi, il suffit d'ajouter ce commentaire et de changer l'extension « pas » en « dpr ». Le programme va alors s'exécuter comme attendu.

2 Affectation

2.1 Entrées-sorties

Dans la suite nous allons transformer notre premier programme pour qu'il produise un autre effet-net. Les instructions que nous allons y ajouter vont toutes se placer entre `begin` et `end`.

Les instructions `write` et `writeln` (identique à `write` mais suivi d'un saut à la ligne) permettent d'afficher des données à l'écran. Ces instructions sont suivies d'un point-virgule comme presque toute instruction *Delphi*.

Exemples :

`writeln(4785);` {affiche le nombre 4785}

`writeln(47.85);` {affiche le nombre décimal 47,85 (point décimal !!!)}

`writeln('delphi');` {affiche la chaîne de caractères « delphi »}

Retenons ici la règle de syntaxe importante :

Chaque instruction se termine par un point-virgule « ; ».

*Devant **end** on peut laisser de côté ce point-virgule.*

Rappelons qu'il faut terminer par l'instruction `readln` qui fait attendre le programme jusqu'à ce que l'utilisateur appuie sur la touche « enter ». Cela est nécessaire pour laisser à l'utilisateur le temps de lire l'affichage avant que le programme ne se termine et ne ferme la fenêtre.

Après avoir ajouté ces instructions au programme, la partie centrale de celui-ci se présente sous la forme suivante :

```
begin
  writeln(4785);
  writeln(47.85);
  writeln('delphi');
  readln;
end.
```

Si nous exécutons le programme maintenant, l'affichage est le suivant :

```
4785
4.78500000000000E+0001
delphi
```

L'affichage de 4.78500000000000E+0001 peut surprendre. Delphi utilise d'office une écriture, lourde mais précise du nombre 47.85. Il le transforme en notation scientifique. Il faut donc lire : $4.785 \cdot 10^{1}$.

L'instruction `writeln` permet de formater l'affichage de la façon suivante :

```
writeln(47.85:10:8); {10 chiffres au total, dont 8 derrière la virgule)
writeln(4785:6);  {affichage à 6 places, 2 blancs suivis des 4 chiffres 4785 }
```

Exercice :

Apportez ces modifications dans le programme et relancez-le pour vous en convaincre.

Essayez d'autres paramètres d'affichage pour en constater l'effet !

Ces formatages sont nécessaires pour garantir une bonne présentation à l'écran, surtout si on prévoit d'afficher beaucoup d'informations, éventuellement en colonnes ou en tableaux.

2.2 Les opérations arithmétiques.

Delphi permet d'effecteur non seulement les opérations arithmétiques mais toutes les opérations mathématiques usuelles.

- `+` est le signe pour l'addition,
- `-` celui de la soustraction,
- **`*`** celui de la multiplication et
- `/` représente une division.

Les parenthèses peuvent être utilisées comme en mathématiques et elles peuvent être imbriquées à plusieurs niveaux. Les crochets et accolades qui apparaissent dans des expressions mathématiques doivent être remplacés par des parenthèses. Delphi effectue les expressions arithmétiques en appliquant correctement les règles de priorité.

Les opérations **div** et **mod** sont seulement définies pour des nombres entiers :

div donne le quotient (entier) et

mod le reste de la division euclidienne.

`Exemples : 15 div 7 = 2` et `15 mod 7 = 1` car `15 = 7 * 2 + 1`

Les fonctions

sqrt (racine carrée, « square root ») et

sqr (carré, « square »)

sont aussi définies et utilisées comme en mathématiques. Remarquez que les parenthèses sont obligatoires.

Exemples	**Affichage**
`writeln(5+3*7)`	`26`
`writeln((2-7)*5+3/(2+4))`	`-2.45000000000000E+0001`
`writeln(57 div 15)`	`3`
`writeln(57 mod 15)`	`12`
`writeln(sqrt(9))`	`3.00000000000000E+0000`
`writeln(sqr(5))`	`25`
`writeln(sqrt(sqr(3)+sqr(4)))`	`5.00000000000000E+0000`

Exercice 2-1

Écrivez une instruction qui calcule et affiche le résultat de chacune des expressions suivantes :

a) $15+7\cdot 4+2:3$

b) $17-\left[14-3\cdot\left(7-2\cdot 8\right)\right]\cdot 2$

c) $\dfrac{12-5\cdot 7}{13+3\cdot 4}$

d) $\dfrac{(12-5)\cdot 7}{(13+3)\cdot 4}$

e) $\left(\dfrac{2-7}{5}\right)^2-\dfrac{(3-5)^2}{7-5}$

f) $\sqrt{48^2+20^2}$

Exercice 2-2

Effectuez (sans l'aide de l'ordinateur) les expressions suivantes :

a) `86 div 15`
b) `86 mod 15`
c) `(5-(3-7*2)*2)*2`
d) `sqrt(sqr(9)+19)`
e) `145-(145 div 13)*13`
f) `288 div 18`
g) `288/18`

2.3 Variables

Un concept fondamental de l'informatique est celui de la variable. Comme en mathématiques une variable représente une certaine valeur. Cette valeur peut être un nombre, une chaîne de caractères, un tableau, …

Le nom d'une variable n'est pas limité à une seule lettre.

Un nom de variable admissible (accepté par le système) commence par une lettre et peut être suivi d'une suite de symboles qui peuvent être des lettres, des chiffres ou le blanc souligné « _ » (« underscore »).

Un nom valable sera appelé *identificateur* dans la suite. Vu que le nombre de variables d'un programme peut être très grand, **il est utile de choisir des identificateurs qui donnent une indication sur l'usage envisagé de la variable**.

Par exemple : Dans un programme calculant une puissance, les noms de variables `base` et `exposant` seraient appropriés. Il est important de noter que ce choix est seulement fait pour simplifier la lecture et la compréhension du programme et que la signification éventuelle du nom échappe complètement à Delphi.

Delphi ne distingue pas entre majuscules et minuscules. Les identificateurs `a` et `A` ne peuvent donc être discernés. Dans la suite nous n'utiliserons que des lettres minuscules.

2.3.1 Type d'une variable

Chaque variable possède un type précis.

Dans la suite nous utiliserons les types **`integer, real, boolean, char, string`** qui sont utilisés pour représenter :

Type	**Valeur**	**spécifications**
`integer`	nombre entier relatif	entre -2^{31} et $2^{31}-1$
`int64`	nombre entier relatif	entre -2^{63} et $2^{63}-1$
`real`	nombre décimal (en notation scientifique)	voir alinéa suivant
`extended`	nombre décimal (en notation scientifique)	voir alinéa suivant
`boolean`	**true** ou **false**	indique si une condition est vraie ou fausse
`char`	un caractère	lettre, chiffre ou autre caractère
`string`	chaîne de caractères	p.ex. 'abfzr3_45*' ou '1235'

Le type **`real`** demande plus d'attention. Vu que l'écriture décimale d'un nombre réel (mathématique) peut nécessiter une infinité de chiffres décimaux, une représentation fidèle ne peut pas être garantie dans l'ordinateur et un nombre du type **`real`** représente un nombre appelé décimal en mathématiques.

Chaque langage de programmation impose des contraintes aux nombres.

En Delphi un nombre de type **`real`** est représenté par un *nombre en virgule flottante* (« floating point »). C'est une notation scientifique où la mantisse peut avoir jusqu'à 15 chiffres décimaux significatifs et où l'exposant est compris entre –324 et +308. Ces bornes sont assez grandes pour garantir le bon fonctionnement de presque tout programme raisonnable. Un nombre de type **`extended`** peut avoir jusqu'à 19 chiffres décimaux significatifs et un exposant compris entre –4951 et 4932.

Le type **`boolean`** qui admet seulement les deux valeurs **true** et **false** peut surprendre. Ce type est utilisé par Delphi pour représenter l'état de vérité (par exemple d'une condition). Une expression comme *a<5* aura comme valeur **true** si *a* est effectivement inférieur à 5 et **false** dans le cas contraire.

Dans un programme, un caractère ou une chaîne de caractères sont entourés du symbole « ' » (apostrophe). Si la chaîne contient elle-même une apostrophe, alors il faut en mettre deux de suite, sinon Delphi pense que cette apostrophe termine la chaîne. Ainsi l'expression « l'ordinateur » s'écrit `'l''ordinateur'` en Delphi. Il ne faut surtout pas mettre de guillemets anglais « " » comme on a l'habitude de les mettre dans un texte.

Lorsque Delphi affiche un caractère ou une chaîne de caractères par une instruction `write` ou `writeln` les apostrophes au début et à la fin sont omises. De même, une double apostrophe à l'intérieur d'une chaîne est affichée comme simple apostrophe.

Il est aussi déconseillé d'utiliser des lettres accentuées, car leur affichage en console n'est pas correct.

2.3.2 Déclaration d'une variable

Avant la première utilisation dans un programme, une variable doit impérativement être déclarée. Toutes les déclarations de variables suivent directement le mot réservé **`var`** et se présentent sous la forme :

```
var   <ident_a1>,<ident_a2>,...,<ident_an>: <type_a>;
      <ident_b1>,<ident_b2>,...,<ident_bm>: <type_b>;
      ...;
```

La syntaxe est à respecter scrupuleusement :

- Différents identificateurs de variables de même type sont séparés par une virgule (« , ») ;
- l'identificateur et le type sont séparés par un deux-points (« : ») ;
- le type est suivi d'un point-virgule (« ; »).

Ces déclarations se trouvent avant le bloc **begin ... end**.

Exemple :

```
program Project1;
{$APPTYPE CONSOLE}
var  {déclarations de variables}
  a,exposant:integer;
  base,puissance:real;
  mot:string;
begin
  {instructions du programme}
end.
```

Cette déclaration permet au compilateur de réserver efficacement pour chaque variable l'espace mémoire nécessaire pour stocker la valeur ultérieure.

2.3.3 L'affectation

Jusqu'à présent les variables ont été déclarées mais elles ne représentent pas encore de valeur déterminée. L'opération qui consiste à attribuer une valeur à une variable est appelée affectation et elle respecte la syntaxe suivante :

```
<nom_de_variable> := <expression>;
```

Par exemple :

```
a:=45;
base:=12.34;
puissance:=3*3*3*3;
mot:='test';
```

Delphi **effectue d'abord l'expression à droite de « := » et affecte (attribue) le résultat obtenu à la variable dont le nom se trouve à gauche de « := ».**
Pour que l'affectation soit possible, **il faut que le résultat de l'expression de droite existe et soit du même type que la variable !**

Ensuite le nom de la variable peut apparaître dans une expression.

Par exemple :

`nombre:=6;`	la valeur 6 est affectée à la variable `nombre`
`writeln(nombre+2);`	afficher `nombre+2`, c'est-à-dire afficher 8.

L'expression de droite peut aussi contenir des variables, y compris celle qui se trouve à gauche de l'affectation !

`nombre:=6;`	la valeur 6 est affectée à la variable nombre
`nombre:=nombre+2;`	effectuer le membre de droite : `nombre+2=8` et affecter le résultat 8 comme nouvelle valeur à la variable nombre. L'effet de l'instruction est donc `nombre:=8;`
`writeln(nombre);`	afficher la valeur actuelle de `nombre`, donc 8.

2.3.4 Initialisation d'une variable

Il est important de constater que la valeur d'une variable peut varier (d'où le nom) au cours de l'exécution du programme. La première affectation d'une variable est appelée initialisation de la variable. Avant l'initialisation, la variable a une valeur non définie ! L'initialisation d'une variable doit précéder l'apparition de la variable dans une expression.

En particulier une affectation comme

```
terme:=terme+1;
```

n'est utile que si la variable `terme` a été initialisée auparavant. Sinon le membre de droite, contenant terme n'est pas défini.

Il est important de remarquer que le fait de ne pas initialiser la variable ne va pas empêcher le programme de démarrer, mais le résultat sera vraisemblablement faux!

Exercice 2-3 (résolu)

Les variables a, b et c sont de type **`integer`**.

Faites un tableau reprenant la valeur de chaque variable **après** chaque ligne.

Un « ? » indiquera que la valeur de la variable n'est pas connue à cet instant !

Instruction	a	b	c
`a:=7`	7	?	?
`b:=4`	7	4	?
`a:=a+b`	11	4	?
`c:=2*a-3*b`	11	4	10
`b:=(a+c) div 4`	11	5	10
`c:=c-a`	11	5	-1

Exercice 2-4

Même question que pour l'exercice précédent.

Instruction	a	b	c
`a:=37 mod 7;`			
`b:=a*3;`			
`c:=a-2*b;`			
`a:=(a*3) div 5 + a;`			
`b:=(b-2)*(b-1);`			
`b:=b*b;`			

Exercice 2-5

Même question que pour l'exercice précédent, mais les variables sont maintenant de type **`real`** !

Instruction	a	b
`a:=9;`		
`b:=sqrt(a);`		
`b:=2*b/5+a;`		
`a:=sqr(a);`		
`b:=b/2+a/2;`		
`b:=(b+a)/2;`		

2.4 L'instruction readln

Il existe une autre méthode pour attribuer une valeur à une variable : la lecture de la valeur du clavier. L'instruction

```
readln(nombre);
```

attend que l'utilisateur introduise au clavier une valeur, suivie de « enter ». Le programme lit ensuite cette valeur et l'affecte à la variable nombre. Si la valeur n'est pas compatible avec le type de la variable, alors le système arrête le programme et affiche un message d'erreur. Pour éviter ce type de problème, il est utile d'informer l'utilisateur sur ce qu'il est censé introduire :

```
write('Veuillez introduire un nombre: ');
readln(nombre);
```

Il est important de remarquer qu'**il faut introduire une valeur** et non pas une expression à évaluer.

Si le programme attend un nombre et que l'utilisateur introduit par exemple 2+5, alors le système va interrompre l'exécution avec un message d'erreur !

Si le programme attend une chaîne de caractères, alors ce type d'erreur ne peut pas arriver car '2+5' est une chaîne valable. Il n'est évidemment pas garanti que le programme va en faire un usage « intelligent ». Il ne va en aucun cas évaluer l'expression !

2.5 Les constantes

Une constante représente une certaine valeur qui ne change à l'intérieur du programme. Cette valeur peut être un nombre, une chaîne de caractères, un tableau, … Pour la dénomination des constantes, les règles sont les mêmes que pour les variables.

Dans la suite nous n'utiliserons que des lettres majuscules pour les constantes.

Une constante doit impérativement être déclarée. Toutes les déclarations de constantes suivent directement le mot réservé **`const`** et se présentent sous la forme :

```
const <constante_a1> = <valeur>;
      <constante_a2> = <valeur>;
      ...;
```

La syntaxe est à respecter scrupuleusement. On peut noter que contrairement aux variables, l'affectation se fait avec un « = » au lieu d'un « := ».
Ces déclarations se trouvent avant le bloc **begin ... end**.

2.6 Exercices

Exercice 2-6 : « Swap »

a) Écrivez un programme qui lit 2 nombres entiers du clavier et les affecte respectivement aux variables `nombre1` et `nombre2`.

b) Écrivez ensuite des instructions qui permettent d'échanger (« swap ») les valeurs des variables `nombre1` et `nombre2`.

Exercice 2-7 : Moyenne arithmétique

Écrivez un programme qui lit 2 nombres réels du clavier, calcule leur moyenne (arithmétique) et l'affiche.

Exercice 2-8 : Moyenne harmonique

Écrivez un programme qui lit 2 nombres réels strictement positifs du clavier, calcule leur moyenne harmonique et l'affiche.

Rappelons que la moyenne harmonique de 2 nombres strictement positifs est donnée par la formule : $\frac{1}{m_h} = \frac{\frac{1}{a} + \frac{1}{b}}{2}$. (Il est recommandé de transformer la formule avant de l'utiliser dans le programme !)

Exercice 2-9 : TVA

Écrivez un programme qui lit un nombre réel strictement positif qui représente le prix d'un article quelconque (TVA de 15% comprise). Le programme calculera le prix hors TVA de l'article et l'affichera.

Exercice 2-10 : Loi des résistances

Écrivez un programme qui lit 2 nombres réels, strictement positifs, représentant 2 résistances, du clavier. Le programme calculera ensuite la résistance résultante par la loi d'Ohm, au cas où on met ces 2 résistances en parallèle et l'affichera.

Exercice 2-11 : Aire d'un triangle par la formule d'Héron

Écrivez un programme qui lit 3 nombres réels, représentant les mesures des 3 côtés d'un triangle. (Ces nombres doivent donc vérifier l'inégalité triangulaire).

Le programme devra ensuite calculer, à l'aide de la formule d'Héron, l'aire du triangle correspondant à ces mesures et l'afficher.

Exercice 2-12 : Division euclidienne

Écrivez un programme qui lit le dividende et le diviseur non nul d'une division et qui affiche la division euclidienne résultante.

Exemple : Si le dividende est 34 et le diviseur 8, le programme devra afficher 34=8*4+2.

Exercice 2-13 : Suite arithmétique

Écrivez un programme qui lit le premier terme et la raison d'une suite arithmétique. Le programme lira ensuite un nombre naturel n, calculera le n^e terme ainsi que la somme des n premiers termes de la suite et les affichera.

2.7 Différents types d'erreurs

Si un programme ne produit pas le résultat attendu ou pas de résultat du tout, cela peut être dû à des erreurs de nature très variée. Selon l'instant où le problème se manifeste, on distingue les erreurs et les alertes décrites ci-dessous.

2.7.1 Erreur de compilation (« compilation error »)

On parle d'erreur de compilation, si Delphi n'est pas capable de traduire correctement le programme.

L'erreur la plus typique de ce genre est l'erreur de syntaxe (« syntax error »), qui indique que l'écriture n'a pas été respectée : mot clé mal écrit, « ; » manquant, instruction ou expression mal formées... .

D'autres erreurs de compilation apparaissent, si le programme utilise auparavant une variable non déclarée ou si dans une affectation le type de la valeur et de la variable sont incompatibles.

2.7.2 Erreur d'exécution (« runtime error »)

Pour ce type d'erreur, l'exécution du programme commence correctement, mais elle est interrompue par le système, suite à un événement non prévu.

Des exemples typiques de cette sorte d'erreurs sont la division par zéro ou la lecture d'une valeur ne correspondant pas au type de la variable.

2.7.3 Erreur de programmation

Ce type d'erreur ne provoque aucune réaction du système, mais il se manifeste par un résultat incorrect. Dans ce cas il faut revoir l'algorithme. Le programme est mal conçu !

2.7.4 Alertes (« warnings »)

Une alerte est un message que Delphi affiche lors de la compilation pour indiquer non pas une erreur proprement dite, mais une instruction suspecte, de mauvais style.

Par exemple une alerte est affichée si une variable non initialisée est utilisée ou si une variable est bien déclarée mais jamais utilisée ou encore si sa valeur n'est jamais utilisée.

Une alerte sert souvent d'indication pour trouver une erreur de programmation.

3 Structure alternative

3.1 Problème introductif

3.1.1 Énoncé

Les compteurs du gaz affichent 5 chiffres et permettent d'indiquer des consommations allant de 00000 à 99999 m^3. Lorsque la consommation est plus élevée que cette valeur, le compteur recommence à 00000.

Pour calculer la consommation, on retranche le dernier affichage connu de l'affichage actuel. La consommation ainsi obtenue est multipliée par le prix du gaz au m^3 afin d'obtenir le prix à payer.

De nos jours, le prix à payer est calculé par ordinateur. Concevez un programme permettant de réaliser ce calcul.

3.1.2 Première solution

```
program gaz1;
{$APPTYPE CONSOLE}
var a1,a2,conso:integer;
  pm3,prix:real;
begin
  write('Entrez le dernier affichage connu et l''affichage actuel : ');
  readln(a1,a2);
  write('Entrez le prix au m3 : ');
  readln(pm3);
  conso:=a2-a1;
  prix:=conso*pm3;
  writeln('Le prix a payer est de : ',prix);
  readln;
end.
```

3.1.3 Exemples d'exécution

a1=78000

a2=98000

pm3=0.23

Instruction	a1	a2	pm3	conso	prix
`readln a1,a2`	78000	98000	?	?	?
`readln pm3`	78000	98000	0.23	?	?
`conso:=a2-a1`	78000	98000	0.23	20000	?
`prix:=conso*pm3`	78000	98000	0.23	20000	4600

a1=98000

a2=18000

pm3=0.23

Instruction	a1	a2	pm3	conso	prix
Readln a1,a2	98000	18000	?	?	?
readln pm3	98000	18000	0.23	?	?
conso:=a2-a1	98000	18000	0.23	-80000	?
prix:=conso*pm3	98000	18000	0.23	-80000	-18400

On constate que dans le premier exemple, le résultat est correct. Par contre, le résultat du deuxième exemple n'a pas de sens. Il ne peut y avoir ni de consommation ni de prix négatif.

La raison pour laquelle on obtient ces résultats est que le programme ne tient pas compte du fait que le compteur peut dépasser 100000 et que dans ce cas, il recommence à 0. Dans le deuxième cas l'état du compteur n'est pas 18000 mais vaut 18000+100000=118000.

En conséquence, il faudrait ajouter 100000 à l'affichage actuel du compteur au cas où cette valeur est inférieure au dernier affichage connu.

La structure alternative permet de réaliser ce genre d'opérations conditionnelles.

3.1.4 Deuxième solution

```
program gaz2;
{$APPTYPE CONSOLE}
var a1,a2,conso:integer;
    pm3,prix:real;
begin
  write('Entrez le dernier affichage connu et l''affichage actuel : ');
  readln(a1,a2);
  write('Entrez le prix au m3 :');
  readln(pm3);
  if a2>=a1 then
    conso:=a2-a1
  else
    conso:=a2-a1+100000;
  prix:=conso*pm3;
  writeln('Le prix a payer est de : ',prix);
  readln;
end.
```

On vérifie que le programme ci-dessus fournit les résultats voulus dans les cas de figure précédents. Par contre, le cas où le compteur fait plusieurs fois le tour pour arriver au-delà de 200000 n'est pas traité.

3.2 Syntaxe

3.2.1 Syntaxe simplifiée

```
if <condition> then
  <instruction>;
```

3.2.2 Syntaxe complète

```
if <condition> then
  <instruction>
else
  <instruction>;
```

Remarque importante :

- L'instruction **`if ... then ... else`** est à considérer comme **une seule instruction**. Elle se termine donc par un seul point-virgule « ; » à la fin.
- Il n'y a pas de « ; » après **`then`** ni avant **`else`**
- S'il y a plusieurs instructions, il faut les mettre dans un bloc commençant par **`begin`** et se terminant par **`end.`**

3.3 Les conditions (expressions logiques)

3.3.1 Les opérateurs relationnels

Opération	Opérateur en Delphi
égal à	=
différent de	<>
strictement supérieur à	>
strictement inférieur à	<
supérieur ou égal à	>=
inférieur ou égal à	<=

Deux variables de type **`string`** (chaîne de caractères) peuvent être comparées à l'aide des opérateurs relationnels.

'ALBERTA' > 'ALBERT' et 'BERNARD' > 'ALBERTA'

L'ordre de classement des chaînes est l'ordre lexicographique (celui appliqué dans les lexiques). Les majuscules précèdent les minuscules.

3.3.2 Les opérateurs logiques AND, OR, NOT

Les opérateurs logiques (NOT, AND, OR respectivement en français non, et, ou) permettent de combiner plusieurs conditions :

p.ex. (A>3) et (B<2) n'est vrai que si les deux conditions sont vérifiées à la fois ;

et

(A>3) ou (B<0) est vrai si au moins une des conditions est vraie.

Valeur de X	Valeur de Y	X AND Y	X OR Y
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE

Valeur de X	NOT X
TRUE	FALSE
FALSE	TRUE

3.3.3 Règles de priorité des opérateurs

1. **NOT**, **-,** **+** (signe)
2. *, /, **DIV**, **MOD**, **AND**
3. +, -, **OR**
4. = , <= , >= , < , > , <>

3.4 Exercices

Exercice 3-1

Quel est le résultat des opérations suivantes ?

- `6 <= 10`
- `3 = 3`
- `NOT false`
- `(3 <= 1) AND (2 > 0)`
- `(3 > 1) AND (2 > 0)`
- `5 > 4`
- `NOT (3 < 2)`
- `(3 > 1) OR (2 > 0)`
- `3 <> 3`
- `NOT true`
- `(3 <= 1) OR (2 > 0)`

Exercice 3-2

Ecrivez un programme qui affiche le quotient de deux nombres lus au clavier. Respectez le cas où le diviseur est égal à 0.

Exercice 3-3

Ecrivez un programme qui calcule la valeur absolue d'un nombre (positif ou négatif) sans utiliser la fonction prédéfinie `abs`.

Exercice 3-4

Ecrivez un programme qui affiche le plus grand de 2 nombres.

Exercice 3-5

Ecrivez un programme qui affiche le plus grand de 3 nombres.

Exercice 3-6

Écrivez un programme qui vérifie si un nombre lu au clavier est divisible par 2.

Exercice 3-7

Modifiez les valeurs de trois variables A, B, C entrées au clavier de manière à avoir $A \leq B \leq C$.

Exercice 3-8

Afficher la moyenne de 3 nombres lus au clavier. La moyenne est arrondie vers le haut. Le programme affiche un message « moyenne insuffisante », si elle est inférieure à 30, sinon il affiche « moyenne suffisante ».

Exercice 3-9

Imprimez le signe du produit de deux nombres sans faire de multiplication.

Exercice 3-10

Imprimez le signe de la somme de deux nombres sans faire d'addition.

Exercice 3-11

Vérifiez la validité d'un numéro de compte chèque postal.

Le numéro de contrôle est le reste de la division euclidienne du numéro principal par 97. Si la division est sans reste alors le numéro de contrôle est 97.

Exercice 3-12

Soit une équation de la forme $ax^2+bx+c=0$.

a) Résolvez cette équation en supposant que : $a \neq 0$.

b) Écrivez un nouveau programme qui tient compte du fait que les paramètres a, b, c peuvent être nuls.

4 Structure répétitive

4.1 Introduction

Les programmes rencontrés jusqu'à présent sont d'une utilité très limitée. En effet le gain de temps réalisé à l'aide de l'ordinateur est quasi nul (par rapport à une calculatrice par exemple). L'unique avantage provient du fait qu'un même programme peut être réutilisé pour différentes valeurs des données. Cette situation va changer radicalement dans ce chapitre. Le mécanisme de structure répétitive permet d'exécuter plusieurs fois une certaine partie d'un programme !

4.2 Exemple

Soit la partie suivante d'un programme :

```
var i,puiss:integer;
...
puiss:=1;
i:=1;
while i<=15 do
  begin
    puiss:=puiss*2;
    i:=i+1;
  end;
```

Quel est l'effet-net de ce programme ?

D'abord les variables `puiss` et `i` sont initialisées à 1. Ensuite on arrive à la nouvelle instruction

```
while <condition> do <instruction>;
```

qui comme son nom l'indique exécute instruction aussi longtemps que condition est vérifiée. L'instruction suivant le **`do`** est appelée « corps de boucle » ; elle peut être constituée d'une instruction simple ou d'un bloc **`begin ... end`**.

Dans l'exemple :

1. La condition est d'abord testée. Elle est vérifiée, car `i=1` et ainsi `i<=15`.
2. Ensuite `puiss` est multipliée par 2 et i augmentée de 1.
3. L'exécution reprend au point 1 (avec les nouvelles valeurs des variables).
4. La condition est à nouveau testée. Elle est vérifiée, car `i=2` et ainsi `i<=15.`
5. Ensuite `puiss` est à nouveau multipliée par 2 et i augmentée de 1.
6. L'exécution ne quitte la boucle que si la condition n'est plus vérifiée ! Cela arrive lorsque i atteint la valeur 16 (i est entier et augmentée de 1 à chaque passage dans la boucle).

   La boucle est donc exécutée 15 fois, car si on part de `i:=1`, il faut ajouter 15 fois le nombre 1 à i pour arriver à 16.

   En même temps `puiss` aura été multipliée 15 fois de suite par 2.

   Vu que `puiss` avait la valeur 1 avant le premier passage de boucle `puiss` aura la valeur $1*2*2*2*2*2\ldots = 2^{15} = 32768$.

Exercice 4-1

On donne la partie de programme (toutes les variables sont de type **integer**) :

```
resultat:=1;
facteur:=5;
while facteur > 1 do
  begin
    resultat:=resultat*facteur;
    facteur:=facteur-1;
  end;
writeln(resultat);
```

Simulez l'exécution de ce programme en représentant dans un tableau les valeurs successives des variables. Que va afficher le programme ?

Complétez (entête, déclarations, etc.) le programme pour qu'il soit accepté par Delphi et vérifiez votre résultat !

4.3 Terminaison d'une boucle

Lors de la conception d'une boucle, il est essentiel de veiller à ce que la boucle se termine, c'est-à-dire qu'après un nombre fini de passages la condition de boucle va finir par ne plus être vérifiée.

```
i:=1;
while i<=5 do
  begin
    puiss:=puiss*2;
  end;
```

Cette boucle est mal conçue, car la seule variable i de la condition n'est pas modifiée dans le corps de boucle. À chaque passage l'évaluation de la condition donne le même résultat et le programme ne va plus sortir de la boucle.

4.4 Boucles for

4.4.1 Boucles `for … to`

Dans beaucoup de cas on peut simplifier l'écriture du programme en utilisant une boucle **for**.

Exemple d'une boucle `for` :

```
puiss:=1;
for i:=1 to 5 do puiss:=puiss*2;
```

Dans la boucle **for**, moins flexible que **while**, la condition est remplacée par un compteur (dans l'exemple précédent le compteur est la variable `i`). Ce compteur est initialisé à une certaine valeur (ici 1, par `i:=1`) et augmenté de 1 à chaque passage de boucle. La boucle va être parcourue jusqu'à ce qu'une valeur finale (ici 5) soit atteinte.

Les valeurs initiale et finale sont calculées une fois pour toute avant l'entrée dans la boucle. Elles ne sont plus réévaluées plus tard.

Dès l'entrée dans la boucle le nombre de passages est donc connu.

Le compteur et les valeurs initiale et finale seront tous de type **integer**.

Si la valeur initiale est supérieure à la valeur finale, alors le corps de boucle n'est pas exécuté du tout.

Remarque :

Une boucle **for** de la forme

```
for i:=a to b do <instruction>;
```

peut toujours être réécrite en boucle **while** de la manière suivante sans que l'effet-net du programme ne change :[3]

```
i:=a;
while i<=b do begin <instruction>; i:=i+1 end;
```

Le passage d'une boucle **while** à une boucle **for** n'est pas toujours possible sans changer la nature de l'algorithme. Cela est clair vu qu'une boucle **for** se termine toujours alors que ce n'est pas le cas de toute boucle **while**.

4.4.2 Boucles `for … downto`

Il existe une variante de la boucle **for** où le compteur est **diminué** de 1 à chaque passage :

```
for i:=a downto b do <instruction>;
```

qui a le même effet-net que

```
i:=a;
while i>=b do begin <instruction>; i:=i–1 end;
```

4.5 Exercices

Exercice 4-2

On donne la partie de programme (toutes les variables sont de type **integer**) :

```
resultat:=0;
terme:=7;
for i :=0 to 5 do
  resultat:=resultat+terme;
writeln(resultat);
```

Simulez l'exécution de ce programme en représentant dans un tableau les valeurs successives des variables. Que va afficher le programme ?

Complétez (entête, déclarations, etc...) le programme pour qu'il soit acceptable par Delphi et vérifiez votre résultat !

Exercice 4-3

Réécrivez le programme précédent à l'aide d'une boucle **while**.

Complétez ensuite le programme pour qu'il soit accepté par Delphi et vérifiez votre résultat !

[3] À condition que les valeurs des variables i et b ne soient pas modifiées à l'intérieur du bloc <instruction>.

Exercice 4-4 Somme de nombres lus du clavier

Ecrivez un programme qui calcule et affiche la somme de nombres réels lus successivement au clavier.

Le programme continue jusqu'à ce que l'utilisateur entre le nombre 0. S'il entre 0 comme premier nombre, le programme affichera 0.

Allez-vous utiliser une boucle **`for`** ou une boucle **`while`** ? Expliquez votre choix !

Exercice 4-5 Somme de n nombres lus du clavier

Ecrivez un programme qui lit d'abord le nombre naturel n, représentant le nombre de termes à ajouter. Ensuite le programme va lire n nombres réels du clavier, calculer leur somme et l'afficher.

Allez-vous utiliser une boucle **`for`** ou une boucle **`while`** ? Expliquez votre réponse et comparez avec l'exercice précédent !

Exercice 4-6 (*) Table de multiplication

Ecrivez un programme qui calcule et affiche une table de multiplication (« Einmaleins ») de $0 \cdot 0$ jusqu'à $10 \cdot 10$. L'affichage doit se faire sous forme de tableau « bien formaté ».

4.6 Un algorithme efficace (facultatif)

Le premier programme calcule une puissance (de base réelle non nulle et d'exposant naturel) par un algorithme simple basé sur la définition par multiplications successives : $\begin{cases} a^0 = 1 \\ a^{n+1} = a^n \cdot a \end{cases}$

```
program puissance;
{$APPTYPE CONSOLE}
var
  expo:integer;
  base,puiss:real;
begin
  write('Introduisez la base (reelle non nulle): ');
  readln(base);
  writeln('Introduisez l''exposant (naturel): ')
  readln(expo);
  puiss:=1;
  while expo<>0 do
    begin
      puiss:=puiss*base;
      expo:=expo-1 ;
    end;
  writeln('la puissance vaut',puiss)
end.
```

Ce premier programme effectue toujours n multiplications pour une puissance d'exposant n.

Dans le deuxième programme, le nombre de multiplications va se réduire de façon spectaculaire. Il suffit d'ajouter une ligne supplémentaire à la définition pour traiter les exposants pairs.

$$\begin{cases} a^0 = 1 \\ a^{2n} = \left(a^2\right)^n \\ a^{2n+1} = a^{2n} \cdot a \end{cases}$$

Le programme n'est pas très différent de la puissance *simple*.

À l'intérieur de la boucle, l'exécution diffère selon la valeur de la variable `expo`.

Si `expo` est pair, on peut appliquer la formule $a^{2n} = \left(a^2\right)^n$. Cela consiste à remplacer la base par son carré et à réduire l'exposant de moitié.

Si `expo` n'est pas pair, le programme applique le même algorithme que la puissance par produits successifs.

Remarquons que dans ce cas, l'exposant est diminué de 1 et qu'il va donc devenir pair pour le prochain passage dans la boucle ! Donc l'exposant est réduit de moitié après au plus deux passages dans la boucle. Ainsi pour n'importe quel exposant inférieur à 1024 le nombre de produits à calculer ne va pas dépasser 20 ($2^{10} = 1024$), alors que 2^{1000} nécessite exactement 1000 produits avec l'algorithme simple !

Examinons le fonctionnement du programme à l'aide d'un exemple d'exécution. Pour calculer $2^{13} = 8192$, les variables `base`, `expo` et `puiss` vont prendre successivement les valeurs reprises dans le tableau suivant.

On remarquera que l'expression $I = base^{\text{expo}} \cdot puiss$ garde la même valeur après chaque passage de la boucle. Cette expression est un **invariant**. La notion d'invariant est utile pour montrer l'exactitude d'un programme.

Base	Expo	puiss	$I = base^{\text{expo}} \cdot puiss$
2	13	1	$I = 2^{13} \cdot 1 = 8192$
2	$12 = 13 - 1$	$2 = 1 \cdot 2$	$I = 2^{12} \cdot 2 = 8192$
$4 = 2 \cdot 2$	$6 = 12 : 2$	2	$I = 4^6 \cdot 2 = 8192$
$16 = 4 \cdot 4$	$3 = 6 : 2$	2	$I = 16^3 \cdot 2 = 8192$
16	$2 = 3 - 1$	$32 = 2 \cdot 16$	$I = 16^2 \cdot 32 = 8192$
256	1	32	$I = 256^1 \cdot 32 = 8192$
256	0	8192	$I = 256^0 \cdot 8192 = 8192$

Et voici le programme :

```
program puissance_rapide;
{$APPTYPE CONSOLE}
var
  expo:integer;
  base,puiss:real;
begin
  write('Introduisez la base (reelle non nulle): ');
  readln(base);
  writeln('Introduisez l''exposant (naturel): ')
  readln(expo);
  puiss:=1;
  while expo<>0 do
    begin
      if (expo mod 2)=0 then
        begin
          base:=base*base;
          expo:=expo div 2
```

```
        end
      else
        begin
          puiss:=puiss*base;
          expo:=expo-1 ;
        end;
    end;
  writeln('la puissance vaut',puiss)
end.
```

4.7 Exercices

Exercice 4-7 Suite arithmétique

Écrivez un programme qui lit le premier terme et la raison d'une suite arithmétique. Ensuite le programme lira un nombre naturel **n** et il calculera par sommation et affichera le n^e terme et la somme des n premiers termes de la suite.

Dans cet exercice, contrairement à l'exercice de même nom du chapitre 2, on n'utilisera pas les formules des suites arithmétiques mais seulement la définition par récurrence.

Exercice 4-8 Suite géométrique

Même question pour une suite géométrique.

Exercice 4-9 Factorielle

La factorielle d'un nombre naturel n, notée n! est définie par :

$$\begin{cases} 0! = 1 \\ (n+1)! = n! \cdot (n+1) \end{cases}$$

Ainsi si **n** n'est pas nul on a:

n!=n·(n-1)·(n-2)·...·3·2·1

et en particulier 1!=1, 2!=2·1=2, 3!=3·2·1=6, 4!=43·2·1=24 etc.

a) Écrivez un programme qui lit un nombre naturel **n** et qui calculera et affichera ensuite **n!** par une boucle **for...to**.

b) Même question mais avec une boucle **for...downto**.

c) Même question mais avec une boucle **while**.

Exercice 4-10 (*) PGCD par l'algorithme d'Euclide (par soustraction)

Les formules d'Euclide pour trouver le pgcd de 2 nombres naturels non nuls sont :

$$\begin{cases} \text{pgcd}(a,a) = a \\ \text{pgcd}(a,b) = \text{pgcd}(b,a) \\ \text{pgcd}(a,b) = \text{pgcd}(a-b,b) \ si\ a > b \end{cases}$$

Écrivez un programme qui lit les 2 nombres naturels **a** et **b** et qui calcule et affiche leur pgcd en utilisant ces formules.

Exercice 4-11 (*) PGCD par l'algorithme d'Euclide (par division)

Les formules suivantes, aussi attribuées à Euclide, mènent plus vite au résultat :

$$\begin{cases} \text{pgcd}(a,0) = a \\ \text{pgcd}(a,b) = \text{pgcd}(b,a) \\ \text{pgcd}(a,b) = \text{pgcd}(a \bmod b, b) \ si\ a > b > 0 \end{cases}$$

Écrivez un programme qui lit les 2 nombres naturels **a** et **b** et qui calcule et affiche leur pgcd en utilisant ces formules.

Exercice 4-12 (*) Test de primalité

Écrivez un programme qui lit un nombre naturel non nul **n** et qui vérifie ensuite si **n** est premier. Vous pourrez améliorer le programme en n'essayant que les diviseurs potentiels inférieurs ou égaux à $\sqrt{n}$.

Exercice 4-13 (**) Factorisation première

En vous basant sur l'exercice précédent, écrivez un programme qui lit un nombre naturel non nul **n** et qui calcule et affiche ensuite la décomposition en facteurs premiers de ce nombre :

Exemple : Pour 360, le programme affichera : 360=2·2·2·3·3·5

Exercice 4-14 (***) Factorisation première (produit de puissances)

Même question qu'à l'exercice précédent, mais on demande d'afficher le résultat comme produit de puissances.

Exemple : Pour 360, le programme affichera : 360=2^3·3^2·5^1

5 Fonctions et procédures

5.1 Les fonctions

5.1.1 Introduction

Exemple

Écrivons un programme qui calcule le nombre de combinaisons de n objets p à p.

Rappelons que ce nombre se détermine par la formule :

$$\binom{n}{p} = \frac{n!}{p!(n-p)!}$$

En regardant cette formule nous constatons que pour chaque calcul d'un nombre de combinaisons nous devons effectuer trois fois le calcul d'une factorielle et ainsi répéter trois fois le même code. Les fonctions ont été développées pour éviter ces redondances dans les programmes.

Grâce à elles il suffit de noter une seule fois le code de calcul de la factorielle et de l'utiliser à chaque besoin.

Établissons d'abord le calcul de la factorielle sous forme de fonction.

```
function fact(a:integer):integer;
var i, res : integer;
begin
  i:=2;
  res:=1;
  while i<=a do
    begin
      res:=res*i;
      i:=i+1;
    end;
  result:= res;
end;
```

L'identificateur **a** est un paramètre dont la valeur va être fixée à chaque nouvel appel de la fonction.

Il est important de remarquer que la fonction **fact** retourne une valeur qui sera transmise au programme appelant.

Une fonction, une fois définie comme telle, peut être utilisée sous son nom dans le programme principal, qui va donc avoir la forme:

```
program factorielle;
{$APPTYPE CONSOLE}
var n, p, res_c   : integer;
{************************************}
function fact(a:integer) :integer;
var i, res : integer;

begin
```

```
  i:=2;
  res:=1;
  while i<=a do
    begin
      res:=res*i;
 i:=i+1;
    end;
result:= res;
end;
{************************************}
begin
  write('n = ');
  readln(n);
  write('p= ');
  readln(p);
  res_c:= fact(n) div (fact(p)*fact(n-p));
  write('Nombre de combinaisons de ',n,' objets ',p,' a ',p,' =
',res_c);
  readln;
end.
```

5.1.2 Définition

Une fonction est une partie de code d'un programme que nous pouvons utiliser plusieurs fois. Chaque fois que nous utilisons la fonction nous pouvons lui transmettre d'autres valeurs qu'elle utilisera pendant son exécution. Le code de la fonction est placé directement après les déclarations de variables (du programme principal).

Syntaxe de la définition d'une fonction

```
function
<nom>(<paramètre1>:<type1>;<paramètre2>:<type2>;…):<type_res>;
var  …
begin
    <instructions>
    result := <valeur à transmettre>
end;
```

Lorsqu'on utilise plusieurs paramètres de même type alors on peut utiliser la syntaxe simplifiée :

```
function <nom>(<paramètre1A>,<paramètre1B>:<type1>;…):<type_res>;
```

> Avant le mot clef **end** doit toujours figurer l'affectation qui indique le résultat à faire passer au programme appelant. Dans cette affectation, on utilise comme variable à gauche l'identificateur `result` ou le nom de la fonction.

Syntaxe de l'appel d'une fonction dans le programme appelant

```
<nom_de_variable>:=<nom>(valeur1,valeur2, … ) ;
```

Les valeurs `valeur1, valeur2, ...` sont affectées aux paramètres formels `<paramètre1>, <paramètre2>,...` dans le même ordre qu'ils interviennent dans la partie déclarative de la fonction. Dans le corps de la fonction les paramètres formels se comportent comme des variables et

ils peuvent être utilisés comme telles. La variable <nom_de_variable> doit être du type <type_res>.

On remarquera que dans la définition, des paramètres de types différents sont séparés par un « ; », alors que dans l'appel tous les paramètres sont suivis d'un « , ».

Une fonction peut contenir elle-même une ou plusieurs fonctions dont elle a besoin.

Exercice 5-1

Écrivez une fonction qui retourne le maximum de 3 valeurs.

Exercice 5-2

Écrivez une fonction qui détecte si une chaîne de caractères est vide.

Exercice 5-3

Écrivez une fonction qui compte le nombre de chiffres contenus dans une chaîne composée de chiffres et de lettres.

Exercice 5-4

Écrivez un programme qui lit le numérateur et le dénominateur d'une fraction et fournit la fraction simplifiée.

5.2 Les procédures

Exemple

Écrivez un programme **moy** qui lit successivement trois valeurs positives **a**, **b** et **c** et détermine la moyenne arithmétique. La valeur de **a** doit être inférieure à 10, celle de **b** inférieure à 20 et celle de **c** inférieure à 30.

Si les valeurs ne vérifient pas ces conditions, le programme émet un message d'erreur, aussitôt la valeur saisie. Si les valeurs sont correctement saisies, le programme affiche la moyenne arithmétique.

Ici aussi nous voyons que le message d'erreur peut être plusieurs fois le même. Il n'est donc pas nécessaire de mettre plusieurs fois le code servant à l'afficher. De plus si nous voulons modifier le texte des messages, il suffit de le changer une fois.

Pour ce faire nous utilisons les procédures.

La différence entre une procédure et une fonction se trouve dans le fait qu'une fonction renvoie une valeur au programme appelant tandis qu'une procédure effectue un traitement (affiche un message à l'écran) mais ne renvoie pas de valeur.

```
program moy;
{$APPTYPE CONSOLE}
var
  a, b, c : integer;
  moy     : real;
(*****************************************)
procedure erreur();
 begin
   writeln('Valeur incorrecte. Recommencez.');
 end;
(*****************************************)
```

```
begin
  a:= 10;
  b:= 20;
  c:= 30;
  while a>=10 do
    begin
     write('Valeur 1 (<10): ');
     readln(a);
     if a>=10 then
      erreur();
    end;
  while b>=20 do
    begin
     write('Valeur 2 (<20): ');
     readln(b);
     if b>=20 then
      erreur();
    end;
  while c>=30 do
    begin
    write('Valeur 3 (<30): ');
    readln(c);
    if c>=30 then
      erreur();
    end;
  moy:= (a+b+c)/3;
  writeln('La moyenne de ',a,',',b,' et ',c,' vaut ',moy:5:2);
  readln;
end.
```

5.2.1 Définition

Une procédure est aussi une partie de code que nous pouvons utiliser plusieurs fois.

La différence entre une fonction et une procédure réside dans le fait qu'une procédure ne transmet pas de valeur au programme appelant. Comme effet-net une procédure peut afficher un message à l'écran ou modifier des variables du programme.

Syntaxe de la définition d'une procédure

```
procedure <nom>(<paramètre1>:<type1>;<paramètre2>:<type2>;…);
var  …
begin
     <instructions>
end;
```

Syntaxe de l'appel en Delphi d'une procédure dans le programme appelant

```
<nom>(valeur1, valeur2 …) ;
```

Ici aussi, les valeurs `valeur1, valeur2, ...` sont affectées aux paramètres formels `<paramètre1>, <paramètre2>,...` dans le même ordre qu'ils interviennent dans la partie déclarative de la procédure. Dans le corps de la procédure les paramètres formels se comportent comme des variables et peuvent être utilisés comme telles.

5.2.2 Avantages des fonctions et procédures

- **Nous évitons les redondances :** Avec ces notions nous ne répétons pas inutilement du code. Ceci rend les programmes plus petits.
- **Nous retrouvons facilement les fonctions et/ou procédures :** Si nous voulons modifier un programme nous n'avons pas besoin de passer en revue toutes les pages du code.
- **La lisibilité est améliorée :** Il est plus facile de relire du code, surtout si nous ne l'avons pas rédigé nous-mêmes.
- **Les fonctions et procédures sont portables :** Une fonction et/ou une procédure une fois écrite pour un certain programme peut être réutilisée pour un autre programme sans pour autant la réécrire. Nous pouvons ainsi construire des bibliothèques de telles fonctions ou procédures.

5.2.3 Exercices

Adaptez les programmes des chapitres précédents en les écrivant à l'aide de fonctions et de procédures chaque fois que cela est utile.

5.3 Portée

On appelle *portée d'une fonction, d'une procédure ou d'une variable* la partie du code dans laquelle elles conservent la signification définie par la déclaration, c'est-à-dire dans laquelle elles peuvent être utilisées comme elles ont été définies.

5.3.1 Portée d'une fonction/procédure

Règle 1 :

Une fonction ou une procédure n'est connue que si sa définition précède l'endroit d'où elle est appelée.

Exemple :

```
program por11;
{$APPTYPE CONSOLE}
{*****************************************}
function f: integer;
var f1 : integer;
begin
  f1 := 23;
  result := f1;
end;
procedure p;
begin
  writeln(f);
end;
{*****************************************}
begin
  p;
  readln;
end.
```

Ce programme donnera comme résultat 23.

Si nous définissons la procédure **p** avant la fonction **f**, le programme marquera comme erreur qu'il ne trouve pas la fonction **f**.

Règle 2 :

Une fonction ou procédure définie à l'intérieur d'une autre fonction ou procédure n'est connue que dans celle-ci.

Exemple :

```
program por12;
{$APPTYPE CONSOLE}
var v : integer;
{*********************************************}
procedure p1(a:integer);
  {***********************************}
  function carre(p: integer): integer;
  var res: integer;
  begin
    res:=p*p;
    result:=res;
  end;
  {***********************************}
begin
  writeln('a*a = ',carre(a));
end;
{*********************************************}
begin
  v:=4;
  p1(v);
  readln;
end.
```

Ce programme donnera comme résultat 16.

Si nous remplaçons l'appel **p1(v)** par **writeln('v*v = ',carre(v));** le compilateur nous donne l'erreur *Undeclared identifier: 'carre'*, puisque la fonction **carre** est déclarée dans la procédure **p1** et ainsi elle n'est pas connue à l'extérieur de cette procédure**.**

5.3.2 Portée des variables

Nous distinguons entre deux types de variables, les *variables locales* à une fonction/procédure et *les variables globales*.

Une variable est *locale* à une fonction/procédure, si elle est déclarée dans la celle-ci.

Exemple :

```
program por21;
var a: integer;
{$APPTYPE CONSOLE}
{p1**********************}
procedure p1;
var a_1: integer;
  {p11**************}
  procedure p11;
  var a_11 : integer;
```

```
  begin
    …
  end;
  {p11**************}
begin
  …
end;
{p1**********************}
begin
  …
end.
```

Dans cet exemple la variable **a** est globale, tandis que les variables **a_1** et **a_11** sont des variables internes respectivement à **p1** et à **p11.**

Règle 3 :

La portée d'une variable locale est strictement limitée à la fonction ou à la procédure où elle est déclarée.

La portée d'une variable est interrompue chaque fois que dans la lecture du code nous rencontrons le code d'une fonction ou d'une procédure interne dans lequel le même identificateur est utilisé.

Exemple :

```
program por22;
{$APPTYPE CONSOLE}
{**************************************}
procedure p1;
var a : integer;
  procedure p11;
  var a : integer;
  begin
    a := 12;
    write('Valeur de a dans  p11 :');
    writeln(a);
  end;

begin
  a:= 5;
  write('Valeur de a avant p11 :');
  writeln(a);
  p11;
  write('Valeur de a apres p11 :');
  writeln(a);
end;
{**************************************}
begin
  p1;
  readln;
end.
```

Voici un tableau qui montre le déroulement du programme.

Programme principal	procedure p1	procedure p11	a
program Project1; {$APPTYPE CONSOLE}			
begin			
p1;	procedure p1;		
	var a : integer;		
	a:= 5;		5
	write('Valeur de a avant p11 :');		5
	writeln(a);		5
	p11;	procedure p11;	5
		var a : integer;	
		begin	
		a := 12;	12
		write('Valeur de a dans p11 :');	12
		writeln(a);	12
		end;	5
	write('Valeur de a apres p11 :');		5
	writeln(a);		5
	end;		
readln;			
end.			

5.3.3 Exercices

Exercice 5-5

Soit le code suivant :

```
program expor;
{$APPTYPE CONSOLE}
var a, b, c : integer;
procedure p1();
var a, b:integer;
begin
a:= 2;
b:= 4;
writeln('a= ',a,',  b= ',b,',  c= ',c);
end;
{************************************}
```

```
begin
a:= 10;
b:= 15;
c:= 30;
writeln('a= ',a,',  b= ',b,',  c= ',c);
p1();
writeln('a= ',a,',  b= ',b,',  c= ',c);
readln;
end.
```

Déterminez les valeurs qui seront sorties par le programme.

5.4 Passage des variables dans une fonction ou une procédure

Exercice : Écrivez une procédure qui échange deux variables.

Écrivons notre programme de la façon suivante:

```
program pass1;
{$APPTYPE CONSOLE}
var a,b : integer;
procedure ech(v1, v2: integer);
var temp: integer;
begin
  temp:= v1;
  v1:= v2;
  v2:= temp
end;

begin
  write('a= '); readln(a);
  write('b= '); readln(b);
  ech(a,b);
  write('a= ',a,'   b= ',b)
  readln;
end.
```

Ce programme ne réalisera pas l'effet-net escompté, mais laissera inchangées les valeurs des variables.

Ceci est dû au fait qu'en Delphi, le passage des paramètres se fait, sauf indication spéciale, *par valeurs.*

Passons en revue les valeurs des différentes variables lors du déroulement du programme:

Supposons qu'on donne à ***a*** la valeur 5 et à ***b*** la valeur 8.

`program pass1;`					
`var a,b : integer;`	création de a	création de b			

`begin`					
`write('a= '); readln(a);`	5				
`write('b= '); readln(b);`		8			
`procedure ech( v1 , v2: integer);`			création de v1 avec la valeur 5	création de v2 avec la valeur 8	
`var temp: integer`					création de temp
`temp:= v1;`					5
`v1:= v2;`			8		
`v2:=temp`				5	
`end; {ech)`			destruction de v1	destruction de v2	destruction de temp
`write('a= ',a,' b= ',b)`	sortie de la valeur de a: 5	sortie de la valeur de b: 8			
`end.`	destruction de a	destruction de b			

Au moment du passage des valeurs une copie est faite, sur laquelle agit la procédure **`ech`**. En sortant de la procédure **`ech`** toute référence à cette copie est détruite et nous nous retrouvons avec les anciennes valeurs de **`a`** et de **`b`**.

Une autre méthode de passage des valeurs est celle qu'on appelle normalement par *référence* ou par *variable*. Cette dernière appellation est utilisée en Delphi.

Avec cette méthode de passage des valeurs une référence est passée au programme appelé qui permet d'accéder directement à la variable du programme appelant.

Règle :

Pour arriver en Delphi à un passage des valeurs par *variable* nous devons faire précéder les différents paramètres formels par le préfixe VAR.

Dans notre exemple la ligne de définition de la procédure **`ech`** aura donc la forme :

```
procedure ech(var V1, V2: integer);
```

Si nous voulons donc qu'une variable définie dans le programme appelant change de valeur par une action dans le programme appelé, nous devons passer la valeur par référence.

5.4.1 Exercices

Exercice 5-6

Utilisez la procédure `ech` pour classer 4 valeurs lues au clavier lors du déroulement du programme.

Exercice 5-7

Soit le code suivant :

```
program pass;
{$APPTYPE CONSOLE}
var a,b : integer;

procedure p1(a,b :integer);
begin
  a:=a+2;
  b:=b*b;
  writeln('Dans la procedure p1');
  writeln('a=',a,' b=',b);
end;

procedure p2(a: integer; var b :integer);
begin
  a:=a+2;
  b:=b*b;
  writeln('Dans la procedure p2');
  writeln('a=',a,' b=',b);
end;

begin
  a:=5;
  b:=7;
  writeln('Avant la procedure p1');
  writeln('a=',a,' b=',b);
  p1(a,b);
  writeln('Apres la procedure p1');
  writeln('a=',a,' b=',b);
  p2(a,b);
  writeln('Apres la procedure p2');
  writeln('a=',a,' b=',b);
  readln ;
end.
```

Quel est l'effet-net de ce programme ?

6 Les structures de données composées

Jusqu'à présent nous n'avons utilisé que des données de type simple : *entier*, *réel*, *booléen* et *chaînes de caractères*. Une variable de type simple ne peut représenter qu'une seule donnée.

Dans ce chapitre nous allons découvrir les données de type structuré. Ces données ont la propriété que chaque identificateur individuel peut représenter des données multiples. On peut traiter l'ensemble de ces données multiples aussi bien que les données individuelles. Les données de type structuré se manipulent donc collectivement ou individuellement.

6.1 Les tableaux

6.1.1 Généralités

Un tableau est une structure de données qui regroupe des données de type identique. On peut définir des tableaux d'entiers, de réels, de booléens, de chaînes de caractères ou bien de tout autre type identique. L'accès à chaque élément d'un tableau se fait moyennant un indice.

6.1.2 Déclaration

Comme pour les variables de type simple, les tableaux doivent être déclarés. Cette déclaration doit comporter un nom et le type de données contenues dans le tableau. En plus on doit indiquer le nombre d'éléments. On obtient la syntaxe suivante :

```
var nom: array [A..B] of <type>
```

avec :

`nom` : nom du tableau
`A` : début de l'indice
`B` : fin de l'indice
`type` : type de donnée des éléments du tableau

Exemples :

- Tableau de 100 entiers :

  ```
  var entiers : array [1..100] of integer;
  ```

- Tableau de 50 noms :

  ```
  var noms : array [51..100] of string;
  ```

- Tableau de 30 réels :

  ```
  var reels: array [30..59] of real;
  ```

- Tableau de 15 booléens :

  ```
  var booleens: array [31..45] of boolean;
  ```

6.1.3 Accès aux éléments

Après avoir déclaré un tableau, nous pouvons utiliser les différents éléments du tableau comme des variables : Nous pouvons faire des accès en écriture (affectation) ou en lecture.

L'accès aux différents éléments du tableau est effectué moyennant un indice. Cet indice est noté entre crochets après le nom du tableau.

Exemples :

Les exemples suivants utilisent les déclarations ci-dessus.

- Écriture de la valeur 5 dans l'élément 3 du tableau Entiers :

```
Entiers[3]:=5;
```

- Affichage de la valeur de l'élément 3 du tableau Entiers :

```
writeln(Entiers[3]);
```

- Affichage de toutes les valeurs du tableau Noms :

```
for I:=51 to 100 do
  writeln(Noms[I]);
```

6.1.4 Remarques

Comme pour les variables simples, les éléments d'un tableau doivent être initialisés avant tout accès en lecture.

On doit respecter les limites de l'indice : Tout accès à un élément portant un indice non défini lors de la déclaration est interdit.

6.1.5 Exemple

Écrivez un programme qui demande à l'utilisateur 10 entiers, les stocke dans un tableau et les affiche ensuite.

```
program exemple;
{$APPTYPE CONSOLE}
var
  I:integer;
  Tableau:array[1..10] of integer;
begin
  for I:=1 to 10 do
      readln(Tableau[I]);
  for I:=1 to 10 do
    writeln('La ',I,'ième valeur vaut : ',Tableau[I]);
end.
```

6.1.6 Exercices

Exercice 6-1

Écrivez un programme qui demande à l'utilisateur 5 réels, les stocke dans un tableau et affiche à la fin le contenu du tableau sur l'écran.

Exercice 6-2

Remplissez un tableau avec 20 entiers aléatoires[4]. Affichez-le puis recherchez le minimum et le maximum du tableau et affichez-les.

[4] Pour obtenir un nombre aléatoire, il faut utiliser la fonction **`random`** (consulter l'aide en ligne (F1) pour plus de détails).

Exercice 6-3

Remplissez un tableau avec 10 entiers aléatoires. Calculez la somme, la moyenne et l'écart-type des valeurs du tableau. Affichez le contenu du tableau ainsi que la somme, la moyenne et l'écart-type.

Exercice 6-4

Remplissez un tableau avec 100 entiers aléatoires compris entre 10 et 25. Comptez le nombre d'occurrences (fréquence) d'un entier entré par l'utilisateur.

Exercice 6-5

Rédigez un programme qui remplit un tableau avec 10 entiers aléatoires.

Affichez-le.

Inversez ensuite l'ordre des nombres dans le tableau et affichez ensuite le tableau réarrangé.

Exercice 6-6

On vous donne le numéro d'un jour d'une année. En supposant qu'il ne s'agit pas d'une année bissextile, déterminez le jour et le mois en question.

Exercice 6-7 (**) Conversion décimale → binaire

Écrivez un programme qui lit un nombre naturel (en notation décimale) non nul *n.* Le programme transformera et affichera ensuite ce nombre en notation binaire.

Exemple : Pour 43, on affichera : 101011 car $43 = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$

Exercice 6-8 (**) Conversion binaire → décimale

Écrivez un programme qui lit un nombre naturel (en notation binaire) non nul *n.* Le programme transformera et affichera ensuite ce nombre en notation décimale.

Exemple : Pour 101011, on affichera : 43.

Le programme n'a pas besoin de vérifier si le nombre donné est effectivement en écriture binaire (seulement chiffres 0 et 1).

Exercice 6-9 (***) Simulation d'un tirage Lotto

Un joueur peut choisir 6 nombres entiers entre 1 et 49. Faites la saisie des nombres proposés par le joueur et mémorisez-les à l'aide d'un tableau de booléens. Veillez à ce les 6 nombres soient distincts deux à deux.

Effectuez ensuite le tirage. Utilisez à nouveau un tableau de booléens pour mémoriser le tirage. Veillez à ce qu'il y ait exactement 6 nombres différents.

Déterminez ensuite combien de nombres le joueur a deviné correctement et affichez le tirage ainsi que les nombres correctement devinés.

6.2 Les tableaux à deux dimensions

6.2.1 Généralités

En mathématiques, les éléments a_{ij} d'une matrice A(n,m) sont groupés en n lignes et m colonnes de la manière suivante :

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1m} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2m} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3m} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nm} \end{bmatrix}$$

Une matrice A(n, m) est carrée si n = m, c'est-à-dire si le nombre de lignes est égal au nombre de colonnes.

En informatique, les matrices à 2 dimensions sont représentées par des tableaux à 2 dimensions.

6.2.2 Déclaration

```
var nom : array [A..B, C..D] of <type>
```

avec :

`nom` : nom du tableau

`A` : début de l'indice de ligne

`B` : fin de l'indice de ligne

`C` : début de l'indice de colonne

`D` : fin de l'indice de colonne

`type` : type de données dans le tableau

6.2.3 Accés aux éléments

L'accès aux différents éléments du tableau est effectué moyennant les indices de ligne et de colonne. Ces indices sont notés entre crochets après le nom du tableau et séparés par une virgule.

6.2.4 Exemple

Le programme suivant déclare et initialise le tableau suivant : $A = \begin{bmatrix} 0 & 6 \\ 1 & 3 \end{bmatrix}$

```
program exemple ;
var a : array[1..2, 1..2] of integer ;
begin
  A[1,1]:=0;
  A[1,2]:=6;
  A[2,1]:=1;
  A[2,2]:=3;
end.
```

6.2.5 Exercices

Exercice 6-10

Écrivez une procedure qui remplit un tableau à deux dimensions avec des valeurs aléatoires, ainsi qu'une procedure qui permet d'afficher le tableau.

Exercice 6-11

Étant donné l'ordre n d'un tableau carré, écrivez une procédure pour construire en mémoire un tableau U tel que :

$$u_{ij} = \begin{cases} 1 \; si \; i = j \\ 0 \; si \; i \neq j \end{cases}$$

Exercice 6-12

Écrivez une procédure pour mettre à 0 les éléments de la diagonale principale d'un tableau carré A(n, n).

Exercice 6-13

Écrivez une procedure pour multiplier un tableau donné A par un réel k donné.

$$3 \cdot \begin{bmatrix} 4 & 5 & 0 \\ 1 & 4 & 7 \\ 2 & 9 & 2 \end{bmatrix} = \begin{bmatrix} 12 & 15 & 0 \\ 3 & 12 & 21 \\ 6 & 27 & 6 \end{bmatrix}$$

Exercice 6-14

Écrivez une fonction pour calculer la somme des éléments d'un tableau donne.

Exercice 6-15

Écrivez une procédure pour calculer la somme de deux tableaux donnés.

$$\begin{bmatrix} 1 & 3 & 7 \\ 2 & 0 & 3 \end{bmatrix} + \begin{bmatrix} 4 & -3 & -6 \\ 5 & -9 & 8 \end{bmatrix} = \begin{bmatrix} 5 & 0 & 1 \\ 7 & -9 & 11 \end{bmatrix}$$

Exercice 6-16

Écrivez une procédure qui réalise le transfert d'un tableau donné vers un vecteur.

$$\begin{bmatrix} 3 & 7 & 2 & 1 \\ 5 & 0 & 3 & 8 \\ 2 & 3 & 1 & 0 \end{bmatrix} \Rightarrow [3 \quad 7 \quad 2 \quad 1 \quad 5 \quad 0 \quad 3 \quad 8 \quad 2 \quad 3 \quad 1 \quad 0]$$

Exercice 6-17

Écrivez une procédure qui réalise le transfert dans le sens opposé de l'exercice précédent.

Exercice 6-18

Écrivez une procédure qui calcule le produit de deux tableaux donnés.

$$\begin{bmatrix} 3 & 7 & 2 \\ 5 & 0 & 3 \\ 2 & 3 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 2 & -1 \\ 3 & 1 & 0 & 2 \\ 2 & -3 & 0 & -2 \end{bmatrix} = \begin{bmatrix} 25 & 4 & 6 & 7 \\ 6 & -4 & 10 & -11 \\ 11 & 2 & 4 & 2 \end{bmatrix}$$

Exercice 6-19

Écrivez une procédure qui transpose un tableau donné.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \Rightarrow t_A = \begin{bmatrix} a_{11} & a_{21} & a_{31} & a_{41} \\ a_{12} & a_{22} & a_{32} & a_{42} \\ a_{13} & a_{23} & a_{33} & a_{43} \\ a_{14} & a_{24} & a_{34} & a_{44} \end{bmatrix}$$

Exercice 6-20

Triangle de Pascal. On rappelle que :

$(a + b)^0 = 1$

$(a + b)^1 = 1a + 1b$

$(a + b)^2 = 1a^2 + 2ab + 1b^2$

$(a + b)^3 = 1a^3 + 3a^2b + 3ab^2 + 1b^3$

$(a + b)^4 = 1a^4 + 4a^3b + 6a^2b^2 + 4ab^3 + 1b^4$

On peut obtenir les coefficients des termes du type akbn-k en construisant de proche en proche les éléments du triangle de Pascal :

n = 0 1

n = 1 1 1

n = 2	1	2	1				
n = 3	1	3	3	1			
n = 4	1	4	6	4	1		
n = 5	1	5	10	10	5	1	
n = 6	1	6	15	20	15	6	1

Construisez et imprimez un tableau dont les éléments représentent les nombres du triangle de Pascal ;; un nombre ne figurant pas dans le triangle sera représenté par 0 dans le tableau. La donnée en entrée est l'exposant n dans $(a+b)^n$.

Exercice 6-21

Recherchez les éléments dans un tableau donné qui sont à la fois maximum sur leur ligne et minimum sur leur colonne. Imprimez le nombre d'éléments trouvés. Exemples :

$$\begin{bmatrix} 1 & 8 & 3 & 4 & 0 \\ 6 & \underline{7} & 2 & 7 & 0 \end{bmatrix} \qquad \begin{bmatrix} 4 & 5 & 8 & 9 \\ 3 & 8 & 9 & 3 \\ 3 & 4 & 9 & 3 \end{bmatrix} \qquad \begin{bmatrix} 3 & 5 & 6 & \underline{7} & \underline{7} \\ 4 & 2 & 2 & 8 & 9 \\ 6 & 3 & 2 & 9 & 7 \end{bmatrix} \qquad \begin{bmatrix} 1 & 2 & \underline{3} \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Exercice 6-22 (***)

Carrés magiques. On donne la dimension (impaire) n d'un tableau carré M. Construisez et imprimez ce tableau en le remplissant avec les nombres de 1 à n^2 de telle sorte que la somme des éléments d'une ligne soit égale à la somme des éléments d'une colonne et à la somme des éléments de chaque diagonale.

Principe : Placez la valeur 1 au milieu de la première ligne. Placez ensuite les autres nombres (2, 3, …) en parcourant le tableau en suivant une diagonale montante vers la gauche et en tenant compte des règles suivantes :

- Si une case est déjà occupée alors le nombre est placé sous la case précédemment remplie.
- Si la case à remplir dépasse la limite supérieure du carré, alors le nombre est placé dans la dernière ligne, sans changer de colonne.
- Si la case à remplir dépasse la limite gauche du carré, alors le nombre est placé dans la dernière colonne, sans changer de ligne.
- Si la case à remplir est située à en haut à gauche à l'extérieur du carré, alors le nombre est placé dans la deuxième ligne et sur la première colonne.

Exemples :

$$\begin{bmatrix} 28 & 19 & 10 & 1 & 48 & 39 & 30 \\ 29 & 27 & 18 & 9 & 7 & 47 & 38 \\ 37 & 35 & 26 & 17 & 8 & 6 & 46 \\ 45 & 36 & 34 & 25 & 16 & 14 & 5 \\ 4 & 44 & 42 & 33 & 24 & 15 & 13 \\ 12 & 3 & 43 & 41 & 32 & 23 & 21 \\ 20 & 11 & 2 & 49 & 40 & 31 & 22 \end{bmatrix} \qquad \begin{bmatrix} 15 & 8 & 1 & 24 & 17 \\ 16 & 14 & 7 & 5 & 23 \\ 22 & 20 & 13 & 6 & 4 \\ 3 & 21 & 19 & 12 & 10 \\ 9 & 2 & 25 & 18 & 11 \end{bmatrix} \qquad \begin{bmatrix} 6 & 1 & 8 \\ 7 & 5 & 3 \\ 2 & 9 & 4 \end{bmatrix}$$

Remarque : Si i = (n + 1)/2, alors $S_{commune} = M(i,i) \cdot n$

6.3 Les enregistrements

6.3.1 Généralités

Un enregistrement est une structure de données qui permet de regrouper des données n'ayant pas besoin d'être de même type. Les données qui composent un enregistrement sont appelés : *champs*. Un champ peut avoir n'importe quel type : integer, real, boolean, string, array ou même un autre enregistrement.

6.3.2 Déclaration

Comme pour les variables de type simple, les enregistrements doivent être déclarés. Cette déclaration doit comporter un nom et doit énumérer les différents champs contenus dans l'enregistrement. Pour chaque champ il faut spécifier son nom et son type. On obtient la syntaxe suivante :

```
nom:record
      champ1:type1;
      champ2:type2;
      champ3:type3;
      …
      champN:typeN;
    end;
```

avec :

`nom` :	nom de l'enregistrement,
`champ1` … `champN` :	noms des différents champs de l'enregistrement,
`type1` … `typeN` :	type de donnée du champ concerné.

Exemples :

- Enregistrement décrivant une date :

```
date:record
       jour:integer;
       mois:string;
       annee:integer;
     end;
```

- Enregistrement décrivant un vecteur du plan (ici 2 vecteurs sont déclarés) :

```
vecteur1,vecteur2:record
                  x:real;
                  y:real;
                  end;
```

- Enregistrement décrivant un élève :

```
eleve:record
        nom:string;
        prenom:string;
        classe:string;
        naissance:record
                    annee:integer;
                    mois:string;
                    jour:integer;
```

```
                end;
        end;
```

- Enregistrement décrivant une voiture :

```
voiture:record
          marque:string;
          modele:string;
          cylindree:integer;
          places:integer;
          essence:boolean;
          options:string;
        end;
```

6.3.3 Accès aux éléments

Après avoir déclaré un enregistrement, nous pouvons utiliser les différents champs de l'enregistrement comme des variables, nous pouvons faire des accès en écriture (affectation) ou en lecture.

L'accès aux différents champs de l'enregistrement est effectué moyennant la notation suivante : **`enregistrement.champ`**

Exemples :

Les exemples suivants utilisent les déclarations ci-dessus.

- On inscrit la date du 3 juin 2003 dans l'enregistrement `date` :

```
date.jour:=3;
date.mois:='juin';
date.annee:=2003;
```

- Le vecteur2 est égal à 3 fois le vecteur1 :

```
vecteur2.x:=3*vecteur1.x;
vecteur2.y:=3*vecteur1.y;
```

- Affichage de la norme du vecteur2 :

```
writeln('Norme : ',sqrt(sqr(vecteur2.x)+sqr(vecteur2.y));
```

- L'élève Toto Tartempion de la classe 2B1 est né le 3 mai 1985.

```
eleve.nom:='Tartempion';
eleve.prenom:='Toto';
eleve.classe:='2B1';
eleve.naissance.jour:=3;
eleve.naissance.mois:='mai';
eleve.naissance.annee:=1985;
```

Remarque

Comme pour les variables simples, les champs d'un enregistrement doivent être initialisés avant tout accès en lecture.

6.3.4 Exemple

Écrivez un programme qui demande à l'utilisateur d'entrer 2 vecteurs du plan et qui calcule et affiche la somme et le produit scalaire de ces deux vecteurs.

```
program Exemple;
var
  scalaire:integer;
  vecteur1,vecteur2,vecteur3:record
                               x:integer;
                               y:integer;
                             end;
begin
  writeln('Entrez les composants du premier vecteur :')
  readln(vecteur1.x,vecteur1.y);
  writeln('Entrez les composants du deuxième vecteur :')
  readln(vecteur2.x,vecteur2.y);
  {Calcul de la somme}
  vecteur3.x:=vecteur1.x+vecteur2.x;
  vecteur3.y:=vecteur1.y+vecteur2.y;
  {Calcul du produit scalaire }
  scalaire:=vecteur1.x*vecteur2.x+vecteur1.y*vecteur2.y;
  writeln('La somme vaut :(',vecteur3.x,';',vecteur3.y,')';
  writeln('Le produit scalaire vaut :',scalaire);
end.
```

6.3.5 Exercices

Exercice 6-23

Donnez la déclaration d'un enregistrement qui permet de mémoriser les données relatives à un article d'un supermarché.

Exercice 6-24

Donnez la déclaration d'un enregistrement qui permet de mémoriser les données relatives à un livre d'une bibliothèque.

6.4 Le mot-clé type

Il est possible de créer de nouveaux types de variables dans Delphi. 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 champ de base de données destiné à l'âge, il utilisait un byte (1 octet contre 2).

6.4.1 Type simple

On déclare les nouveaux types simples 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;
```

6.4.2 Type structuré (enregistrement)

Syntaxe:

```
type <nom_du_ type> = record
     champ1:type1;
     champ2:type2;
     champ3:type3;
end;
```

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

Exemple:

```
program exemple25a ;

type formulaire = Record
  nom:string[20];
  age:byte;
  sexe:char;
  nb_enfants:0..15;
end;

var personne:formulaire;
begin
  personne.nom := 'Etiévant' ;
  personne.age := 18 ;
  personne.sexe := 'M' ;
  personne.nb_enfants := 3 ;
end.
```

L'utilisation de ces champs se fait ainsi : <**variable**>.<**champ**>

6.4.3 Type intervalle

Les types intervalles très utilisés ici ont rigoureusement les mêmes propriétés que ceux dont ils sont tirés. Ils peuvent être de type nombre entier (`integer`) ou caractères (`char`)

Syntaxe:

```
type <nom_du_ 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;
```

6.4.4 Type énuméré

Un type énuméré est un type dont les variables associées n'auront qu'un nombre très limité de valeurs (au maximum 256 différentes possibles).

Exemple:

```
type jours = (dim, lun, mar, mer, jeu, ven, sam);
```

6.4.5 Exercice (facultatif)

Cet exercice (séparé en plusieurs parties) consiste à effectuer certaines opérations sur des polynômes. Pour ce faire, on va créer un nouveau type structuré pour la représentation des polynômes.

Dans le cas des opérations sur les polynômes les éléments importants à regrouper sont les **coefficients** et le **degré** du polynôme. Pour les coefficients on utilise une variable du type tableau (array) et pour le degré une variable du type entier (integer).

Ainsi on déclare le nouveau type de variable structuré pour les opérations sur les polynômes de la façon suivante:

```
type poly = record
  c:array[0..100] of extended;
  d:integer
end;
```

Exercice 6-25

Écrivez une procédure `str2poly` qui transforme un polynôme donné sous forme de texte (la notation est celle utilisée couramment dans les logiciels mathématiques tels que Dérive, GeoGebra, …) en une variable de type `poly`.

Fonctions/Procédures utiles:

- La fonction `pos` sert à extraire des parties de chaînes de caractères. Sa syntaxe est la suivante :

  ```
  pos(s1, s:string): integer;
  ```

  où :

s1	sous-chaîne à rechercher dans la chaîne s ;
s	chaîne dans laquelle on recherce s1 ;

  Si la chaîne `s` ne contient pas `s1`, la fonction retourne 0.

- La fonction `copy` sert à extraire des parties de chaînes de caractères. Sa syntaxe est la suivante :

  ```
  Copy(s, index, count: Integer): string;
  ```

  où :

S	chaîne de laquelle est extraite la partie ;
Index	indice de début de la chaîne à extraire (commence par 1) ;
Count	nombre de caractères à extraire.

Si on souhaite extraire un seul caractère de la chaîne `s`, on peut utiliser accéder aux différents éléments à l'aide de []. Par exemple s[1] donnera le premier caractère de la chaîne. Contrairement à la fonction `copy`, le résultat obtenu est de type `char`.

- La procédure `delete` sert à effacer une partie d'une chaîne de caractères. Sa syntaxe est la suivante :

  ```
  Delete(var s, index, count: Integer): string;
  ```

  où :

S	chaîne de laquelle on veut effacer la partie ;
Index	indice de début de la partie à effacer;
Count	nombre de caractères à effacer.

Exercice 6-26

Écrivez une procédure `poly2str` qui transforme une variable de type `poly` en un polynôme sous forme de texte. Cette procédure servira à afficher les résultats.

Exercice 6-27

Demandez à l'utilisateur d'entrer deux polynômes.

Calculez (en utilisant des fonctions) la somme, la différence et le produit de ces deux polynômes et affichez le polynôme résultant de chaque opération.

Exercice 6-28

Demandez à l'utilisateur d'entrer un polynôme.

Calculez à l'aide d'une fonction la dérivée de ce polynôme et affichez le polynôme résultant.

Demandez à l'utilisateur une valeur réelle pour évaluer le polynôme ainsi que sa dérivée en utilisant le schéma de Horner.

Exercice 6-29 (**) Tableau des valeurs

Écrivez un programme qui :

- demande à l'utilisateur d'entrer un polynôme ;
- remplit ensuite un tableau de valeurs, représenté par un tableau dont les éléments sont des enregistrements, de première composante X et de deuxième composante Y, X variant de -5 à 5 ;
- affiche le tableau des valeurs.