Mini-cours de PASCAL

1ère ANNÉE

Florent BOUCHOUX
florent.bouchoux@insa-lyon.fr

Contenu du mini-cours

Ce mini-cours couvre *l'intégralité* du programme d'informatique de première année, partie Pascal de l'INSA de Lyon ; autant pour les filières classiques que pour la filière *SHN* – Sportifs de Haut Niveau.

Le cours est très exhaustif pour traiter le programme entier, mais tous les chapitres n'ont pas la même importance :

- notions de base
- fonctions
- types avancés

} chapitres *importants* à connaître

- gestions des fichiers
- programmation avancée
- conseils

} chapitres *complémentaires*

Je vous conseille fortement d'imprimer le ***formulaire*** situé tout à la fin du cours ; qui est très utile en TD.

N'hésitez pas à me faire part de vos remarques et suggestions !

Bonne lecture,

Florent Bouchoux
florent.bouchoux@insa-lyon.fr

Bon vent !

Table des matières

CHAPITRE 1

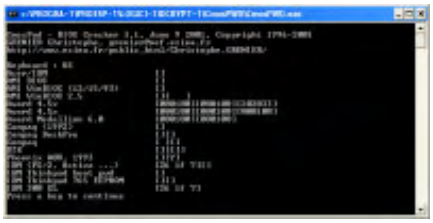
Notions de base

1■ Introduction à la programmation

L'objectif de la programmation est de créer des logiciels. Il est possible de créer :

- des programmes graphiques, tels que ceux que l'on utilise sous Windows ;
- des programmes dits *console*, où le programme ne peut afficher que du texte.

Programme graphique



Programme console

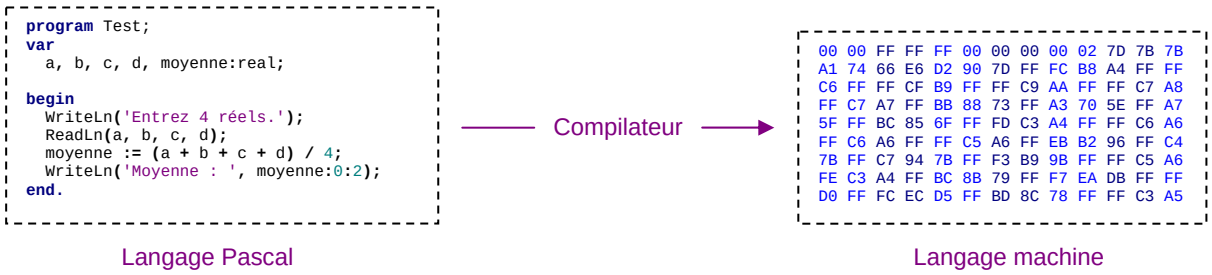
En première année, nous ne réaliserons que des *programmes console*.

Que se passe t-il pour l'ordinateur lorsque l'on exécute un programme ? Il va lire le fichier exécutable – dont le nom se termine en général par .exe – du programme comme une suite de 0 et de 1. Cette suite de 0 et de 1 est appelée langage machine et est directement exécutable par le micro-processeur de l'ordinateur.
A titre d'illustration, ouvrons un fichier .exe pour voir ce qu'il contient – ici affiché en *hexadécimal*, base 16 :

```
00 00 FF FF FF 00 00 00 00 02 7D 7B 7B 1E
A1 74 66 E6 D2 90 7D FF FC B8 A4 FF FF D4
C6 FF FF CF B9 FF FF C9 AA FF FF C7 A8 FF
FF C7 A7 FF BB 88 73 FF A3 70 5E FF A7 71
5F FF BC 85 6F FF FD C3 A4 FF FF C6 A6 FF
FF C6 A6 FF FF C5 A6 FF EB B2 96 FF C4 92
7B FF C7 94 7B FF F3 B9 9B FF FF C5 A6 FF
FE C3 A4 FF BC 8B 79 FF F7 EA DB FF FF EF
D0 FF FC EC D5 FF BD 8C 78 FF FF C3 A5 FF
```

Cette suite de nombres est compréhensible par l'ordinateur, mais pas par un être humain. Afin de pouvoir réaliser facilement des logiciels, on va être amené à utiliser un *langage de programmation* – par exemple le langage Pascal – suffisamment formel pour indiquer à l'ordinateur ce qu'il doit faire, mais plus compréhensible que le langage machine.

Il est alors nécessaire d'utiliser un programme pour transformer le programme écrit en langage Pascal en langage machine. On appelle ce programme *compilateur* ; le processus de transformation s'appelle *compilation*.



Un programme Pascal est composé d'*instructions*, qui sont au Pascal ce que les mots sont au Français. En général, les instructions sont suivies d'un point-virgule `;`.
Il sera précisé explicitement les instructions qui ne sont pas suivies d'un point-virgule ; attention à ne pas oublier ces rares exceptions !

2■ Structure d'un programme

De manière générale, la structure d'un programme simple est toujours la même :

```
program Exemple;                Nom du programme
var
  {variables}                   Déclaration des variables

{fonctions}                     Déclaration des fonctions

begin
  {instructions}                Programme en lui-même
end.
```

On peut constater que le programme est découpé en quatre parties :

- la *première ligne* contenant l'instruction **`program`** suivie du nom du programme ;
- la partie située après **`var`** permettant de déclarer les *variables* allant être utilisées dans le programme ;
- la partie contenant les *fonctions* éventuelles ;
- et enfin le *programme en lui-même*, composé d'instructions Pascal entourées de **`begin`** et **`end.`**.

Nous verrons par la suite la signification exacte de tous ces termes ; cette partie *Structure d'un programme* servira surtout dans les chapitres suivants.
On notera pour l'instant qu'il n'y a pas de point-virgule après les instructions commençant un *bloc*, c'est-à-dire une suite d'instructions décalées vers la droite :

- après le **`var`** commençant un bloc de déclaration des variables ;
- après le **`begin`** commençant le programme en lui-même.

Lorsque l'ordinateur exécute le programme, il commence au **`begin`** ; le programme se termine au **`end.`**. Cela signifie que tout ce qui se situe au dessus du **`begin`** est lu par l'ordinateur *comme une information* mais n'est *pas* exécuté, *sauf si on le demande explicitement* – voir le chapitre concernant les *fonctions*.

Cette structure de base doit être apprise par cœur, car elle constitue le *squelette* du programme.

Il est conseillé, lors de la création d'un programme, de commencer par écrire cette structure. En effet, une fois cette structure créée, le programme est *fonctionnel* et peut être exécuté. Il est bien entendu qu'à ce stade, il ne fait strictement rien puisqu'il n'y a aucune instruction *utile*.

`Remarque.` Si aucune variable n'est déclarée, le **`var`** peut être omis.

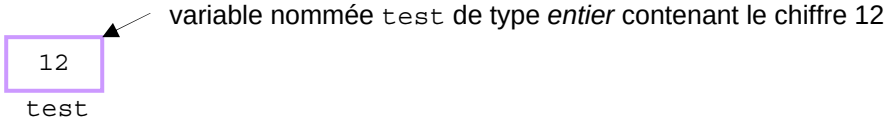
3■ Les variables

3.1 Notion de variable – types simples

Les *variables* constituent l'aspect le plus important de la programmation. Nous nous intéresserons dans un premier temps aux variables *simples*.

Une variable est une sorte de boîte, de bulle portant un nom et contenant une *information d'un certain type* : nombre réel, entier relatif, lettres, etc.
Ce type est prédéfini à l'avance lorsque l'on *crée* – ou *déclare* – la variable. Ainsi, une variable prévue pour contenir un nombre ne peut en aucun cas contenir une lettre.

On pourra se représenter une variable par une *case* :



Chaque type de variable possède un nom anglais.
Sans entrer dans des types plus complexes, on citera 4 types de variables *usuels* :

- **Integer** nombre entier relatif
- **Real** nombre réel (à virgule)
- **Boolean** vaut soit **true** (vrai), soit **false** (faux)
- **String** *chaîne de caractères* – ex. : « ABCD123 »

Les trois premiers types sont des types *mathématiques*, c'est-à-dire qu'il est possible d'effectuer des opérations mathématiques – addition, division, opérateurs, etc. – sur des variables de ce type.

Le dernier type n'est présenté qu'à titre de complément : ce type, bien que très utile en pratique, n'est pratiquement pas utilisé à l'INSA en première année. Concrètement, une variable de type **string** peut contenir une *chaîne de caractères.*
Une chaîne de caractères est une suite de *caractères*, et par caractère on entend tout ce qui peut être écrit au clavier. Ainsi, 'ABC123( )' est une chaîne de caractères.

On retiendra simplement que les chaînes de caractères sont *toujours* placés entre des apostrophes (').

3.2 Déclaration d'une variable

Admettons que l'on souhaite utiliser une variable de type Integer, pouvant donc contenir un entier. Il est nécessaire de *déclarer* cette variable, c'est-à-dire la *créer*. Une fois la variable crée, elle sera – sauf cas particulier – utilisable durant tout le programme ; il n'est pas possible de la *détruire*.

La déclaration des variables se fait *toujours* en dehors du programme en lui-même, après une instruction **var** de la manière suivante :

```
var
  {nom de la variable}:{type};
  Variable1 : integer;
```

Ici, on a *déclaré* la variable nommée `Variable1` de type `integer`.
Cela signifie que la variable est maintenant crée et disponible ; il est possible de placer des données dedans.

`Attention.` Tant que l'on n'a pas placé de données dans `Variable1`, son contenu est *imprévisible* ; la variable n'est *pas vide* et ne contient pas forcément 0.

Si l'on souhaite déclarer plusieurs variables de même type, il suffit d'utiliser la syntaxe :

```
var
  Variable1, Variable2, Variable3 : integer;
  Test1, Test2 : string;
```

3.3 Utilisation d'une variable

Une fois la variable déclarée, il est possible de placer des données dedans : c'est ce que l'on appelle techniquement *l'affectation*. Cette affectation peut se faire n'importe où dans le programme entre les blocs **begin** et **end.**.

L'affectation se réalise de la manière suivante :

```
{nom de la variable} := {donnée à placer dedans};
Variable1 := 12;
```

Dans notre exemple, on *affecte* la valeur 12 à la variable `Variable1` qui est de type Integer.

`Remarque.` Le *symbole d'affectation* est un deux points – égal et non simplement un égal. Remplacer le `:=` par un `=` est un source d'erreur fréquente, signalée par l'ordinateur par un message du type « *:= attendu mais = trouvé* ».

Une fois la variable *déclarée* et *affectée* d'une valeur, il est possible de l'utiliser comme en mathématiques avec des *opérateurs* comme l'addition, la multiplication, etc. On donnera l'exemple suivant :

```
program Exemple;
var
  A, B, C : integer;

begin
      A := 5;
      B := 10;
      C := A + B;            // C vaut alors 5 + 10 = 15
end.
```

L'intérêt des variables est surtout d'effectuer des *tests* sur leur valeur. Par exemple, on souhaite savoir si `Variable1` contient le nombre 12.
Ici, comme `Variable1` contient effectivement 12, *l'expression logique* `Variable1 = 12` vaudra *vrai*, c'est-à-dire **true**.
En revanche, *l'expression logique* `Variable1 = 1` vaudra *faux* (**false**).

Pour plus de détails, on pourra se reporter à la section *Opérateurs*.

Exercice d'application

Créer un programme qui déclare trois variables x, y et z pouvant contenir des nombre entiers. Affecter 5 à x, 3 à y.
Incrémenter (c'est-à-dire ajouter 1 à) x et y, puis affecter à z la somme de x et y.
Que vaut l'expression logique `z = 25` ?

Après avoir écrit la structure de base, on déclare les variables après l'instruction **`var`** ; les variables sont de type *Integer*.

On obtient le programme :

```
program Exercice1;
var
  x, y, z : integer;

begin
      x := 5;
      y := 3;                // affectation des variables

      x := x + 1;            // équivaut à x := 6;
      y := y + 1;            // équivaut à y := 4;

      z := x + y;            // équivaut à z := 10;
end.
```

L'expression `z = 25` vaut **false** puisque, à la fin du programme, z vaut 10.

3.4 Les constantes

Une variable peut, par définition, *changer de valeur* à tout moment. Une *constante*, en revanche, est une variable qui possède une valeur *qui ne change jamais*. Toute tentative de changement de valeur produira une *erreur de compilation.*

L'intérêt des constantes dans un programme n'est pas directement perceptible, car elles permettent surtout de *faciliter la mise à jour* de gros programmes en cas de modifications.

On déclare les constantes de la même manière que les variables, en début de programme. L'instruction à utiliser n'est pas **`var`** mais **`const`** ; il faut placer cette instruction *avant* le **`var`**.

```
program Exemple;
const
  {constante} = {valeur};
  Test = 1;
  Message = 'Bonjour!';
var
  {déclaration des variables}
```

On déclare ici deux constantes, `Test` qui vaut toujours le nombre 1 et `Message`, qui vaut toujours la chaîne de caractères `'Bonjour'`.
Remarquez que, contrairement à la déclaration d'une fonction, ici :

- on n'utilise pas le signe `:=` mais `=`
- *aucun type n'est donné*, la constante peut valoir un nombre, une chaîne de caractères, etc. sans avoir besoin de signaler le type ;
- l'*affectation* d'une constante se fait *hors* du bloc **`begin`** - **`end.`**.

Le fait que l'on n'ait pas besoin de préciser le type vient du fait qu'en réalité, l'ordinateur *remplace* les noms des constantes dans le programme par leur valeur *avant* d'exécuter le programme. Le type n'a alors *aucune importance*.

4■ Fonctions utiles

A ce stade, on présentera deux *fonctions* permettant au programme *d'interagir avec l'utilisateur*. L'étude formelle des fonctions sera l'objet du chapitre suivant.

4.1 La fonction WriteLn

La fonction `WriteLn` permet *d'afficher un message à l'écran*, en sautant une ligne à chaque message. Ce qu'il faut afficher à l'écran est placé entre parenthèses juste à côté du nom `WriteLn`.

Ainsi, l'exemple suivant affichera à l'écran, à l'exécution du programme :
`Bonjour !`

```
program Exemple;

begin
      WriteLn('Bonjour !');
end.
```

On notera que le texte à afficher est une *chaîne de caractères*, il est donc placé *entre apostrophes*.

La fonction `WriteLn` peut accepter entre ses parenthèses une ou *plusieurs* variables séparées par des virgules, *de n'importe quel type simple*. Cela signifie que `WriteLn` peut *afficher directement* un nombre entier, ou un réel.

On pourra ainsi écrire :

```
program Exemple;
var
  i:integer;
begin
      i := 125;
      WriteLn('i vaut ', i, '. Merci !');
end.
```

Ce qui affichera à l'écran `i vaut 125. Merci !`

Cependant, la notation standard *à l'écran* pour les variables de type *Real* est la notation scientifique, peu exploitable. On peut, à l'aide d'une notation, définir *le nombre de chiffres à afficher après la virgule* :

signifie : i avec 5 chiffres après la virgule

```
WriteLn('i vaut ', i:0:5, '. Merci !');
```

Cette notation ne doit être utilisée qu'avec `WriteLn` ; elle n'a aucun sens en elle-même.

`Remarque.` Il existe la fonction `Write` qui s'utilise exactement de la même manière. Cependant, la fonction *ne saute pas de ligne entre chaque message* et tout s'affiche par conséquent à la suite sur l'écran.

4.2 La fonction ReadLn

La fonction `RealLn` est l'opposée de `WriteLn`. Cette fonction permet à l'utilisateur *d'entrer des données* au cours de l'exécution du programme, et de placer ces données *dans une variable*.

Concrètement, la fonction va afficher à l'écran un *curseur clignotant*, l'utilisateur peut alors entrer des données. Les données sont alors placées *dans la variable située entre les parenthèses* à droite de `RealLn`.

Dans l'exemple suivant, le programme attend que l'utilisateur entre un nombre, et place ce nombre dans la variable `i`.

```
program Exemple;
var
  i:integer;

begin
      ReadLn(i);
end.
```

Bien évidemment, si l'utilisateur entre une lettre, *le programme plante* puisqu'il est *impossible* de placer une chaîne de caractères dans une variable de type Integer.

Comme pour la fonction `WriteLn`, il est *possible* de placer plusieurs variables dans la parenthèse, séparées par des virgules. Dans ce cas, l'utilisateur devra entrer *successivement* plusieurs données, qui seront placées dans les variables dans la parenthèse.

Notons qu'en pratique, il est souhaitable *d'indiquer à l'utilisateur qu'il doit entrer des données*, et de préciser lesquelles. En effet, la fonction `RealLn` ne fait qu'afficher un curseur clignotant. On utilisera donc presque toujours la fonction `WriteLn` juste avant `RealLn` afin d'afficher un message demandant à l'utilisateur d'entrer des données ; comme dans l'exemple suivant :

```
program Exemple;
var
  x, y, z:integer;

begin
      WriteLn('Entrez successivement x, y puis z.');
      ReadLn(x, y, z);
end.
```

`Remarque.` *Si le programme n'est pas lancé depuis la console*, le programme s'exécute très rapidement puis se ferme. En effet, en général, *aucune instruction* ne bloque le programme avant le **`end.`**. Une fois cette instruction atteinte, le programme *s'arrête*. Pour bloquer le programme avant le **`end.`** et empêcher qu'il ne se ferme, *il est courant* de placer l'instruction `RealLn;` juste avant le **`end.`**.

Exercice d'application

Créer un programme qui demande 4 nombre réels à l'utilisateur et qui en affiche la moyenne à 10^{-2} près.

Comme d'habitude, commencer par écrire la structure de base.
On utilisera 5 variables : 4 pour stocker les nombres entrés par l'utilisateur, et une pour stocker le résultat du calcul de la moyenne.
Les variables sont *toutes* de type Real (nombres à virgule) puisque la moyenne peut ne pas être un entier.
Comme demandé, la fonction `WriteLn` affiche la moyenne avec deux chiffres après la virgule.

On obtient le programme :

```
program Exercice2;
var
  a, b, c, d, moyenne:real;

begin
      WriteLn('Entrez successivement 4 réels.');
      ReadLn(a, b, c, d);

      moyenne := (a + b + c + d) / 4;
      WriteLn('La moyenne est ', moyenne:0:2, '.');
end.
```

5■ Les tableaux

Afin de ne pas compliquer les choses, on ne s'intéressera dans ce chapitre qu'aux *tableaux à une dimension*. Les tableaux à plusieurs dimensions seront abordés dans le chapitre 3 *Types avancés*.

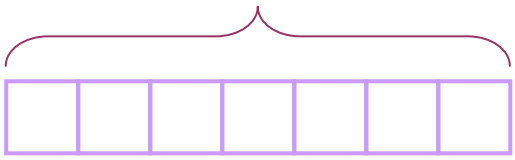
5.1 Notion de tableau – notations

Les variables sont très intéressantes mais elles sont limitées : dans une variable, il n'est possible de placer *qu'une seule donnée*.
Si l'on désire créer un programme qui devra manipuler plusieurs centaines de données, il est bien évident que l'on ne va pas déclarer plusieurs centaines de variables.

Un tableau est une variable qui peut contenir plusieurs données. Le terme de *tableau* est trompeur puisqu'il fait penser à un *tableau à double entrée* ; or, en programmation, un tableau simple ne se présente pas sous cette forme, mais plutôt sous la forme d'une *succession de cases*, chacune de ces cases pouvant contenir une donnée.

Visuellement, on pourra se représenter un tableau de la manière suivante :

Tableau `Exemple` composé de 7 cases

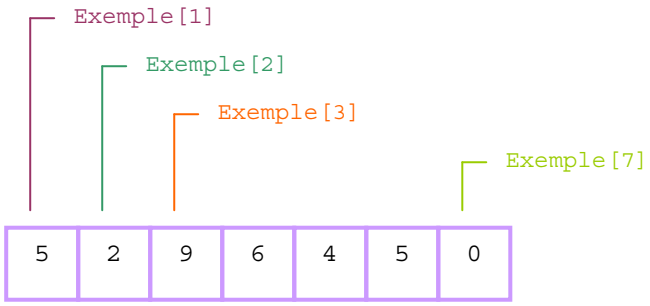


Comme pour les variables, le type de données pouvant être stocké dans les cases est *prédéfini à l'avance* et ne peut être changé une fois le tableau crée. Cela signifie que *toutes les cases* d'un même tableau contiennent *forcément* le même *type* de donnée.
De même, le *nombre de cases* d'un tableau est *fixé lors de sa création* et ne peut être changé.

Chaque case possède un *nom* formé de la manière suivante :

```
nom du tableau[numéro de la case]
```

Prenons l'exemple d'un tableau `Exemple` formé de cases de type Integer (les cases ne pourront contenir que des *nombres entiers*), les cases étant *déjà remplies*. Lors de la création du tableau, on *a décidé que le numéro de la première case serait 1* (pour certaines raisons exposées plus loin, il arrive que l'on choisisse 0).
Chaque case porte donc un *nom distinct* :



En pratique dans le programme, le tableau ne peut être utilisé *qu'à travers ses cases*. Cela signifie que *le nom du tableau seul ne signifie rien pour l'ordinateur*, il faut *toujours* préciser un numéro de case avec la syntaxe *`nom_du_tableau`*`[`*`numéro`*`]`.

Ainsi, lorsque l'on voudra afficher à l'écran *l'intégralité du tableau*, il faudra en réalité afficher séparément la valeur de *chacune des cases*. De même, si l'on souhaite comparer deux tableaux, on ne peut utiliser une expression logique du type `Tableau1 = Tableau2`. *Il faudra comparer séparément chacune des cases* des deux tableaux.

Cependant, il est possible à titre d'exception – nous verrons plus tard pourquoi – de *copier l'intégralité d'un tableau dans un autre tableau* en utilisant la syntaxe :

```
Tableau1 := Tableau2;
```

à la condition *absolument indispensable* que les deux tableaux soient *de même type*, c'est-à-dire qu'ils doivent posséder le même nombre de cases, et le type des cases doit être le même.

5.2 Utilisation d'un tableau

Les cases, via la syntaxe *nom_du_tableau*[*numéro*] s'utilisent comme des variables. Ainsi, un tableau est simplement un ensemble de variables – souvent nombreuses.

Par exemple, si l'on reprend le tableau `Exemple` :

```
begin
      Exemple[1] := 5;
      Exemple[2] := 2;
      WriteLn('La somme des 2 premières cases est :');
      WriteLn(Exemple[1] + Exemple[2]);
end.
```

Comme on peut s'en douter, le programme provoque l'affichage du chiffre `7`.

On voit donc que l'on *peut utiliser les cases d'un tableau comme des variables*. Tout ce qui est valable pour des variables *l'est également pour les cases* d'un tableau.

5.3 Déclaration d'un tableau

Lorsque l'on déclare une variable, il suffit de lui donner un *nom* et un *type*.
Pour déclarer un tableau, il est nécessaire de définir :

- le *nom* du tableau
- le type de ses cases – on dira par raccourci le *type du tableau*
- le nombre de cases
- le numéro de la première case.

Cela se fait dans la section **`var`** de la manière suivante :

```
var
  {nom du tableau}:array[{1ere case}..{dernière}] of {type};
  Exemple:array[1..7] of integer;
```

Ici, `Exemple` est un tableau contenant 7 cases de type Integer (nombres entiers) ; le numéro de la première case étant `1` et celui de la dernière `7`.

Dans la grande majorité des cas, le numéro de la première case sera `1`. Ainsi, si l'on souhaite créer un tableau de *n* cases, *le numéro de la dernière case vaudra n*.

6■ Opérateurs

Cette section présente les *opérateurs* et *opérandes* utilisables en Pascal sous la forme d'un formulaire ; la connaissance de tous ces opérateurs n'est pas nécessaire *mais cependant conseillée*. Les opérateurs les plus importants et les plus utilisés sont présentés en *italique*.

Opérateurs de comparaison :

- *égal* `=`
- *différent* `<>`
- *supérieur strictement* `>`
- *supérieur ou égal* `>=`
- *inférieur strictement* `<`
- *inférieur ou égal* `<=`

Opérateurs de calcul standard :

- *multiplication* `*`
- *division* `/`
- *reste de la division Euclidienne* *a* `mod` *b*
- racine carrée `sqrt(`*n*`)`
- puissance *n'existe pas*
- nombre aléatoire `random(`*n*`)` donnant un entier aléatoire < *n*

  `Remarque.` Utiliser l'instruction `Randomize;` avant d'utiliser `random`, sinon les nombres générés seront à chaque fois les mêmes.

Opérateurs booléens :

- *et* `and`
- *ou* `or`
- *non* `not`
- ou exclusif `xor`

  (une condition ou l'autre est vraie, mais pas les deux à la fois)

7■ Les tests conditionnels

Les *tests conditionnels* permettent d'exécuter – ou de ne pas exécuter – une séquence d'instructions *en fonction d'une condition*, c'est-à-dire d'une *expression logique*.

On distinguera deux types de tests conditionnels, le premier étant *de loin le plus utilisé* :

- le *bloc* **`If Then Else`**
- le *bloc* **`Case of`**

7.1 Bloc *If Then Else*

If Then Else est la traduction en anglais de *Si Alors Sinon*. Concrètement, la commande s'utilise de la manière suivante :

```
If {condition} then
  begin
      {instructions à exécuter si condition est vraie}
  end
else
  begin
      {instructions à exécuter si condition est fausse}
  end;
```

Si l'on ne souhaite pas exécuter d'instructions dans le cas où la condition est fausse – ce qui est souvent le cas – alors le **`else`** devient inutile. On dit qu'il s'agit d'un bloc *If Then* et la syntaxe devient la suivante :

```
If {condition} then
  begin
      {instructions à exécuter si condition est vraie}
  end;
```

On remarquera que :

- les instructions à exécuter sont placées entre un **`begin`** et un **`end;`**, on dira alors que *ces instructions constituent un bloc ;*
- *il n'y a jamais de point-virgule* **`;`** juste avant le **`else`** – ici à la ligne d'au-dessus. Cela constitue une *exception* car, en principe, un **`end`** *est toujours suivi d'un point-virgule*. Si on place un point-virgule avant un **`else`**, le compilateur affichera un message d'erreur et le programme ne pourra pas s'exécuter.

`Remarque.` S'il n'y a qu'une seule instruction à exécuter dans un bloc d'instructions, on peut se passer du **`begin`** et du **`end`**. Ne pas oublier de rajouter ces instructions si l'on ajoute des instructions dans le bloc.

Il est tout à fait possible *d'imbriquer* plusieurs tests **If**. En effet, un **else** correspond toujours au dernier **If** rencontré ; on utilise par conséquent souvent une structure dite *If Else If* :

```
if {condition1} then
      {bloc exécuté si condition1 vraie}
else if {condition2} then
      {bloc exécuté si condition2 vraie; mais condition1 fausse}
else if {condition3} then
      {bloc exécuté si condition3 vraie; mais condition1&2 fausses}
else
      {bloc exécuté si condition1&2&3 fausses}
```

Exercice d'application

Créer un programme qui demande un nombre entier entre 1 et 10 à l'utilisateur. Le programme doit afficher la parité du chiffre ; ou afficher un message d'erreur si l'utilisateur a entré un chiffre supérieur à 10.

Comme d'habitude, commencer par écrire la structure de base. Puis réfléchir au choix des variables : on n'en utilisera qu'une seule, qui contiendra le chiffre entré par l'utilisateur ; cette variable sera donc de type Integer.

Le programme ne doit afficher *qu'un message*. Ainsi, si le nombre est supérieur à 10, le programme ne doit pas afficher sa parité mais seulement le message d'erreur.
Essayer plusieurs structures *If Then Else.* L'ordre des messages dans la structure a son importance, en essayer plusieurs.
Le programme fonctionnel est le suivant :

```
Program Exercice3;
var
  Nb : integer;

begin
      WriteLn('Entrez un chiffre entre 1 et 10.');
      ReadLn(Nb);
      if Nb > 10 then
            WriteLn('Erreur! Chiffre supérieur à 10.')
      else if (Nb mod 2) = 0 then
            WriteLn('Chiffre pair.')
      else
            WriteLn('Chiffre impair.');
end.
```

Ici, un chiffre supérieur à 10 doit arrêter le calcul tout de suite. On place donc *le message d'erreur en premier*, les autres messages ne pouvant s'afficher *que si le message d'erreur ne s'est pas affiché*. Une structure *If Then Else* classique permet ensuite, grâce au calcul du reste de la division de Nb par 2, de déterminer la parité de Nb et d'afficher le message approprié.

7.2 Bloc *Case of*

L'instruction *Case of* permet de tester le contenu d'une variable de type *Integer* (c'est-à-dire des nombres entiers *uniquement*) ou *Boolean* et d'exécuter des instructions suivant la valeur de cette variable.

La syntaxe à utiliser est la suivante :

```
Case {variable} of
  {valeur1} : {bloc d'instructions}
  {valeur2} : {bloc d'instructions}
  {valeur3} : {bloc d'instructions}
  end;
```

On donnera l'exemple suivant :

```
Case Nb of
  1 : WriteLn('Nb vaut 1.');
  2 : WriteLn('Nb vaut 2.');
  3 : WriteLn('Nb vaut 3.');
 end;
```

Dans cet exemple, il n'y a qu'une seule instruction par condition. Dans le cas où on souhaiterait exécuter plusieurs instructions par condition, il faudrait constituer un *bloc d'instructions* en les entourant d'un **begin** et d'un **end**, exactement comme dans la structure *If Then Else*.

Il est également possible d'effectuer un test sur *une plage de valeurs*, comme suit :

```
Case Nb of
  1..3 :  WriteLn('Nb est compris entre 1 et 3.');
  4..10 : WriteLn('Nb est compris entre 4 et 10.');
 end;
```

Cette instruction *Case of* est relativement peu utilisée, notamment en raison du fait qu'une structure *If Else If*, très usitée, peut la remplacer.
Elle est néanmoins utile lorsqu'il faut effectuer de nombreux tests sur une variable de type Integer.

8■ Les boucles

Comme dans tout langage de programmation, les boucles sont indispensables et très souvent utilisées, même dans des programmes simples.

Elles permettent de répéter des instructions plusieurs fois. Bien sûr, répéter plusieurs fois les mêmes instructions sans aucun changement ne présente pas un grand intérêt. Pour être efficaces, les boucles doivent être utilisées avec les variables : si, à chaque fois que les instructions sont exécutées, les données dans les variables changent, le résultat de la boucle sera différent à chaque fois.

Il existe 3 grands types de boucles :

- les boucles **`For`** permettant de réaliser une boucle (répétition d'instructions) *n fois* ;
- les boucles **`While`**, permettent de répéter les instructions tant qu'une condition est vraie ;
- les boucles **`Until`** permettent de répéter les instructions jusqu'à ce qu'une condition soit vraie.

8.1 Boucle *For*

Une boucle *For* permet la répétition d'un *bloc d'instructions* un nombre de fois prédéfini avant l'exécution de la boucle.

De manière générale, les boucles – et en particulier la boucle *For* – utilisent une variable de type Integer (nombre entier) appelée *compteur*, la plupart du temps appelée `i`. Cette *variable compteur* est augmentée de 1 à chaque nouvelle répétition du bloc d'instructions.
Admettons qu'au départ, `i` vaut `1`. Après `10` répétitions du bloc d'instructions, `i` vaudra `10`. *C'est cette variable qui permet de savoir à l'ordinateur combien de fois il a répété le bloc d'instructions*, et donc d'arrêter la répétition une fois le *nombre de fois prédéfini* atteint.

Ainsi, la syntaxe de la boucle *For* est la suivante :

```
For i := {valeur initiale} to {valeur finale prédéfinie} do
  {bloc d'instructions à exécuter n fois}
```

Dans la grande majorité des cas, on fixera au départ de la boucle `i := 1`. Ainsi, la valeur finale de `i` correspond *au nombre de fois que le bloc d'instructions sera répété*.

L'exemple suivant effectue la somme des entiers de 1 à 100 inclus ; à la fin du programme, `somme` vaut `5050`. Il ne faut pas oublier *d'initialiser la variable* `somme` à zéro au début du programme, car son contenu est sinon *imprévisible*.

```
Program Exemple;
var
  i, somme : integer;

begin
      somme := 0;                    // ne pas oublier !
      For i := 1 to 100 do
            somme := somme + i;
end.
```

`Attention.` La variable compteur `i` peut servir dans la boucle – comme ici pour effectuer un calcul – mais *elle sert également à l'ordinateur* pour savoir combien de fois il a répété le bloc d'instructions. Ainsi, *il ne faut jamais modifier cette variable dans la boucle*, sous peine de provoquer un plantage du programme.

Exercice d'application

Créer un programme qui déclare un tableau de 100 nombres entiers.
Remplir ce tableau par 2, 4, 6, 8, …, `200`.

On souhaite déclarer un tableau de 100 cases. On prendra donc `1` comme numéro pour la première case, et `100` pour la dernière. Le tableau est de type Integer.
Le remplissage du tableau se fait par une boucle *For* – les tableaux sont en effet très souvent associés à des boucles pour pouvoir traiter des quantités de données importantes. On sait que `i` – variable de type Integer qu'il ne faut pas oublier de déclarer – est incrémenté de 1 à chaque répétition du bloc d'instructions. La syntaxe `Tableau[i]` permet donc à chaque répétition d'accéder à une case *différente*.

On obtient le programme :

```
program Exercice;
var
  arrTest : array[1..100] of integer;
  i : integer;

begin
     For i := 1 to 100 do
           arrTest[i] := 2*i;
end.
```

Notons que l'on peut, *pour plus de clarté à la relecture*, nommer les tableaux par des noms du type `arrNomDuTableau`, le `arr` précisant qu'il s'agit d'un tableau (pour le programmeur, et non pour l'ordinateur).

8.2 Boucle *While*

La boucle *While* – *tant que* en anglais – permet la répétition d'un bloc d'instructions tant qu'une condition est vraie.

```
While {condition} do
  {bloc d'instructions à exécuter tant que condition vraie}
```

Si la condition du *While* est *toujours* vraie, l'ordinateur exécutera une *boucle infinie*, provoquant un blocage du programme et un ralentissement de l'ordinateur. Avant d'exécuter un programme comportant une boucle *While*, *il est donc souhaitable* de s'assurer que la condition peut devenir fausse (**false**). En particulier, si le bloc d'instructions ne manipule pas de variables, la condition *ne pourra pas* devenir fausse.

Il faut insister sur le fait que contrairement à la boucle *For* dans laquelle une *variable compteur* est incrémentée à chaque répétition, ici, *aucune variable n'est automatiquement changée à chaque répétition*.
Si l'on souhaite utiliser une *variable compteur*, il est nécessaire de l'incrémenter *manuellement dans la boucle*, c'est-à-dire dans le bloc d'instructions répété.
Ainsi, les deux structures suivantes sont équivalentes :

```
For i := 1 to 100 do
begin
      {...}
end;

i := 1;
While i <= 100 do
begin
      {...}
      i := i + 1;
end;
```

Il est important de bien comprendre comment l'ordinateur exécute une telle boucle. Au début de la boucle, il teste si `condition` est vraie. Si elle est fausse, *il n'entre pas dans le bloc d'instructions* mais poursuit l'exécution du programme après ce bloc. Si elle est vraie, il commence à exécuter le bloc d'instructions à répéter.
Une fois le bloc terminé – c'est-à-dire que l'on est arrivé au **`end;`** du bloc – l'ordinateur *revient* à la ligne du *While*. Il re-teste alors `condition`, et ainsi de suite.
Ce mode de fonctionnement implique la condition n'est testée qu'au niveau de la ligne du *While*. Ainsi, si `condition` devient brutalement fausse durant l'exécution du bloc d'instructions à répéter, la boucle ne va pas s'arrêter immédiatement. L'ordinateur va en effet terminer le bloc avant de re-tester `condition`, pour constater qu'elle est fausse. L'exécution du programme se poursuit alors après le bloc d'instructions à répéter.

8.3 Boucle *Until*

Cette boucle est souvent nommée boucle *Repeat Until*. Son mode de fonctionnement est quasi-identique à la boucle *While* ; simplement, la boucle *Until* répète un bloc d'instructions *jusqu'à* ce qu'une condition soit vraie.

La syntaxe de cette boucle est la suivante :

```
Repeat
  {bloc d'instructions sans Begin..End}
Until {condition};
```

On retiendra que :

- le bloc d'instructions à répéter est placé entre **`Repeat`** et **`Until`** ; par conséquent, *il n'est pas nécessaire d'utiliser la structure classique* d'un bloc où les instructions sont entourées par **`Begin`** et **`End`** ;
- il y a un *point-virgule* après la condition.

L'intérêt de cette boucle peut paraître limité ; en effet, si l'on souhaite répéter un bloc d'instructions jusqu'à ce qu'une condition soit vraie, *on peut simplement* utiliser une boucle *While* de la manière suivante :

```
While not {condition} do
  {bloc d'instructions à exécuter tant que condition fausse}
```

Dans ce contexte, la *seule* différence entre les boucles *While* et *Until* est *la manière dont l'ordinateur les exécute.*
En effet :

- dans le cas de la boucle *While*, l'ordinateur teste `condition` *avant* d'entrer dans la boucle. Si `condition` est fausse, *il ne rentre pas dans le bloc* d'instructions ;
- dans le cas de la boucle *Until*, l'ordinateur entre dans le bloc d'instructions à répéter et teste `condition` à la fin de ce bloc, à l'endroit du **`Until`**. Cela signifie que *même si `condition` est vraie, l'ordinateur exécutera le bloc d'instructions une fois.*

Ainsi, sauf si l'on sait ce que l'on fait, on *préfèrera l'utilisation* de la boucle *While*, plus prévisible.

8.4 Instruction de *sortie de boucle*

Dans certaines circonstances, il peut être utile, à un certain moment durant l'exécution d'une boucle, de sortir de cette dernière *immédiatement*.
On utilise alors l'instruction **`break;`** qui provoque une sortie immédiate et définitive de la boucle.
Cette instruction fonctionne pour n'importe quel type de boucle.

Pour une *programmation propre*, certains recommandent de ne jamais utiliser cette instruction. Cependant, elle peut *dans certains cas* permettre d'éviter de trop compliquer les choses mais son emploi doit dans tous les cas être limité au strict minimum.

Exercice d'application

Créer trois programmes qui demandent deux nombres entiers a et b à l'utilisateur, et qui affichent a^b ; chaque programme utilisant un type de boucle différent. L'utilisateur ne doit pas pouvoir entrer un b négatif ; s'il le fait, le programme redemandera de redonner a et b.

Les trois programmes comportent la même structure. Il s'agit de découper la tâche à effectuer en petits morceaux de code :

- la demande de a et b, puis la vérification que b est positif ou nul ;
- le calcul ;
- l'affichage du résultat.

Réfléchissons aux variables nécessaires : on utilisera bien entendu a et b, une variable contenant le résultat appelée `result` et une variable compteur `i`. Ces variables sont toutes de type Integer.

La structure du programme est la suivante :

```
program Exercice;
var
  a, b, result, i : integer;
begin
      {demande a et b, vérifications}

      {calcul}

      {affichage du résultat}
end.
```

Demande de a *et* b

Si b `< 0`, le programme doit *revenir en arrière* et re-demander a et b à l'utilisateur.
Cela implique une boucle, car un bloc *If Then Else* ne permet pas de revenir en arrière dans le programme.
Afin que le programme affiche au moins une fois la demande, on utilisera une boucle *Until* de la manière suivante :

```
repeat
      WriteLn('Veuillez entrer a et b.');
      ReadLn(a, b);
until b >= 0;
```

Remarquons qu'il n'est pas nécessaire, ici d'initialiser b – c'est-à-dire écrire au début `b := 0;`. En effet, l'ordinateur ne testera cette condition qu'*après* le *ReadLn*, b sera alors affecté. Cependant, si l'on avait souhaité utiliser une boucle *While*, il aurait été nécessaire d'initialiser b car le test aurait été fait *avant* de passer dans la boucle.

Calcul – boucle *For*

On obtient le code suivant :

```
result := 1;
For i := 1 to b do
        result := result * a;
```

On notera simplement qu'il est nécessaire d'initialiser `result` à `1` avant de commencer le calcul.
Essayez de faire ce calcul *en lisant le programme*, comme le ferait l'ordinateur, avec a et b petits. Remarquez alors que *la boucle For ne s'arrête que lorsque i est supérieur strictement à b*.

Calcul – boucle *While*

On obtient :

```
result := 1;
i := 1;
while i <= b do
begin
        result := result * a;
        i := i + 1;                     // ne pas oublier
end;
```

On constate qu'il est nécessaire d'incrémenter manuellement la variable compteur i. L'oubli de cette incrémentation provoquera une boucle infinie.
Par ailleurs, on prendra garde à ne pas oublier les initialisations de `result` et de `i`.

Calcul – boucle *Until*

```
result := 1;
i := 1;
repeat
        result := result * a;
        i := i + 1;                     // ne pas oublier
until i > b;
```

Le programme est quasiment identique à celui obtenu pour la boucle *While*.
Cependant, le bloc d'instructions à répéter n'est pas entouré de **`begin`** et **`end;`**.

Affichage du résultat

Il se fait simplement à l'aide de :

```
WriteLn('Le résultat vaut : ', result);
```

CHAPITRE 2

Les fonctions

1■ Présentation – procédures et fonctions

1.1 Présentation

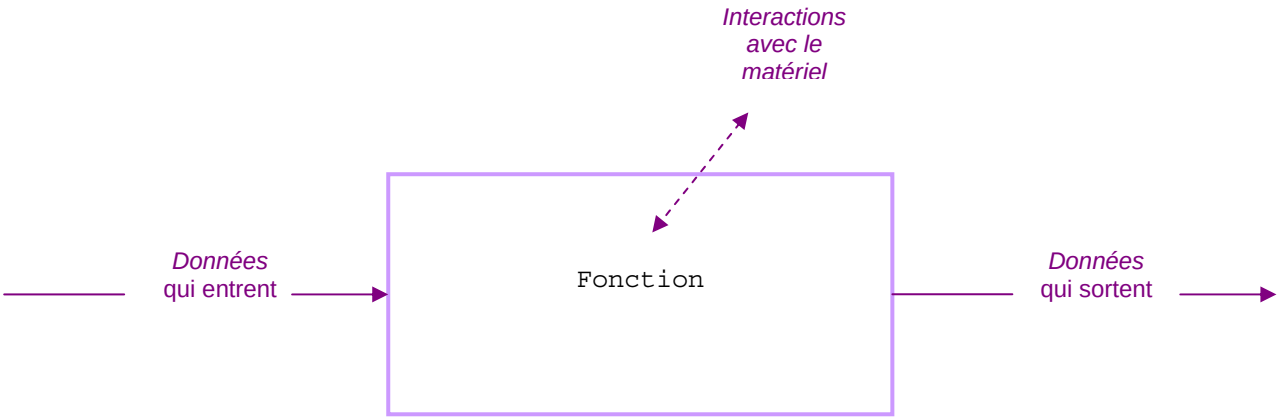
Jusqu'ici, les programmes réalisés étaient fort simples et ne permettaient que la résolution *d'un seul* problème. Cependant, on le comprend aisément, des programmes plus complexes – allant jusqu'à la programmation de logiciels comme *Word* – ne peuvent être écrits *d'un bloc* comme nous l'avons fait jusqu'à maintenant, car ils permettent de résoudre *plusieurs problèmes*, le programme serait alors bien trop long pour pouvoir être lu et modifié facilement par un programmeur.

Dans l'exercice précédent, nous avons commencé à découper le programme en plusieurs parties, afin de diviser le travail et de n'avoir qu'à se concentrer sur une seule tâche.

C'est l'idée des *fonctions* : comme son nom l'indique, une *fonction* va permettre de réaliser une fonction, un petit travail. Ainsi, le programme devient une succession de petits travaux, c'est-à-dire de fonctions.

Nous avons déjà utilisé, sans le savoir, des fonctions : *WriteLn* et *ReadLn*. On comprend bien que pour l'ordinateur et le microprocesseur, afficher un message à l'écran est bien plus compliqué qu'un simple *WriteLn* : pour simplifier, il s'agit d'envoyer des commandes – peu compréhensibles ! – à la carte graphique qui va ensuite afficher un message à l'écran. Il faut ensuite envoyer une autre commande pour demander de passer à la ligne.
Or, dans un programme Pascal, on ne s'occupe pas de cela : c'est la fonction *WriteLn* qui s'en charge ; elle réalise la *fonction* d'afficher quelque chose à l'écran.

Visuellement, on pourra se représenter la notion de *fonction* par une *cellule biologique* :



On constate donc qu'une fonction peut – mais ne le fait pas toujours – :

- *recevoir des données* – une ou plusieurs ;
- *renvoyer des données* – une ou plusieurs ;
- *interagir avec le matériel* – afficher un message à l'écran, imprimer, etc.

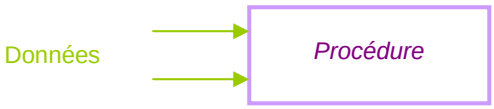
A l'instar d'une cellule biologique, le rôle d'une fonction est de *traiter des données*. Cela signifie qu'en général, la fonction effectue un calcul à partir des données qui entrent, et renvoie – données qui sortent – un résultat.
Les fonctions peuvent aussi interagir avec le matériel ; par exemple afficher un message à l'écran à l'aide de *WriteLn*, ouvrir et lire un fichier, etc.

En pratique, *une fonction n'est pas exécutée spontanément par l'ordinateur* ; il est toujours nécessaire que le programme demande que la fonction soit exécutée, on dit alors que *l'on appelle la fonction*.

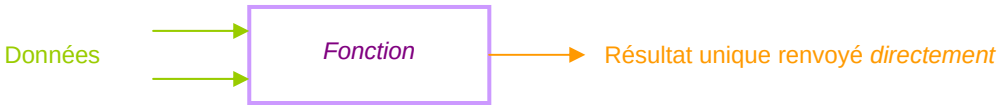
1.2 Différence *fonction – procédure*

La relative complexité des fonctions vient de la multiplicité des mécanismes que la fonction peut utiliser pour recevoir ou renvoyer des données.

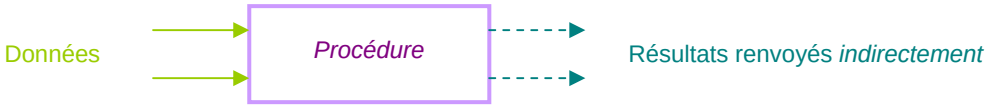
Nous donnons ci-dessous la plupart des cas de figure rencontrés.



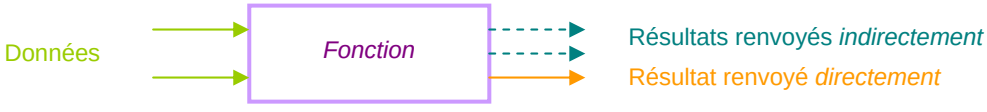
Une *procédure* est une fonction qui ne renvoie pas de résultat *directement*. En revanche, elle peut recevoir en entrée un nombre illimité de données ; elle peut également ne pas en recevoir du tout.



Comme la procédure, la *fonction* peut recevoir en entrée un nombre illimité de données – ou pas du tout. Cependant, elle renvoie *toujours directement un seul* résultat. Nous verrons par la suite à quoi correspond un renvoi *direct* ; on retiendra pour l'instant qu'un renvoi direct ne permet que le renvoi *d'un résultat unique*.



Si l'on souhaite renvoyer plusieurs résultats, on doit utiliser un renvoi *indirect*. On peut utiliser ce mécanisme indifféremment avec une procédure ou une fonction. Ici, dans le cas d'une procédure : on n'utilise qu'un renvoi *indirect*.



On peut combiner un renvoi *direct* et *indirect* avec une fonction. Dans ce cas, un seul résultat est renvoyé *directement*, les autres résultats sont renvoyés *indirectement*.

2■ Portée des variables

Il faut imaginer une fonction – ou une procédure – comme une *cellule* biologique.
Il est important de retenir que le travail qu'effectue une fonction se fait *indépendamment du reste du programme*.

Les seuls liens de la fonction avec le reste du programme sont :

- les données entrantes ;
- les données sortantes ;
- les *variables globales* du programme.

La fonction peut accéder aux variables déclarées dans le programme en lui-même, c'est-à-dire déclarées en dehors d'une fonction. Ces variables sont appelées *variables globales.* Toutes les variables que nous avons déclarées jusqu'à maintenant étaient des variables *globales*, puisque nous le faisions *en dehors d'une fonction*.

Cependant, comme la cellule biologique possède ses propres outils, la fonction peut posséder ses *propres variables*, qui ne sont accessibles ni par les autres fonctions, ni par le reste du programme. Ces variables sont appelées *variables locales*.

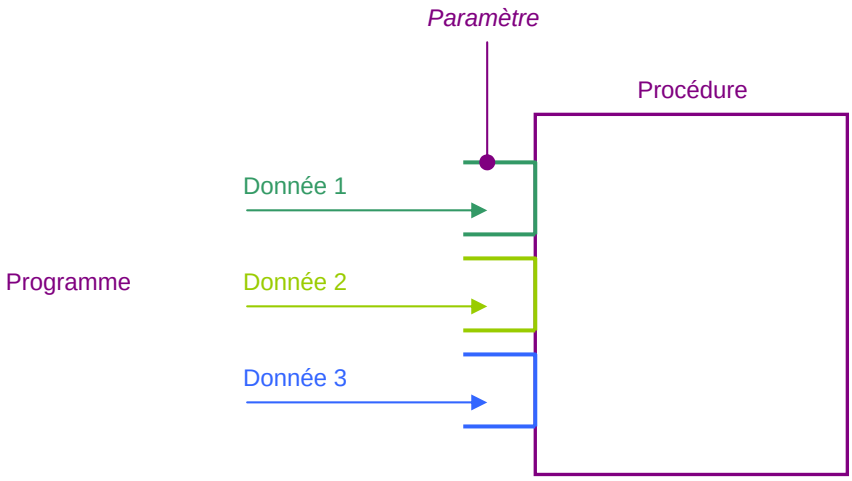
Il est fortement recommandé d'éviter d'utiliser un même nom pour une variable globale et une variable locale. En effet, les instructions dans la fonction peuvent accéder aux variables locales *et* globales.
Dans le reste de ce mini-cours, nous avons choisi de faire précéder les noms de variables locales par `m_`. Ainsi, une variable globale et une variable locale ne peuvent porter le même nom.

3■ Procédures

3.1 Notion de paramètre

Rappelons qu'une procédure est une fonction *qui ne renvoie pas de données*.
Dans ce contexte, son rôle consiste la plupart du temps à *interagir avec le matériel* ; sinon une telle fonction n'aurait strictement aucun intérêt.
En revanche, une procédure peut recevoir des données. Pour ce faire, il est nécessaire d'introduire un mécanisme appelé *paramètres*.

L'idée est que les données entrantes – envoyées à la procédure par le programme – sont placées dans des *variables appelées paramètres* :

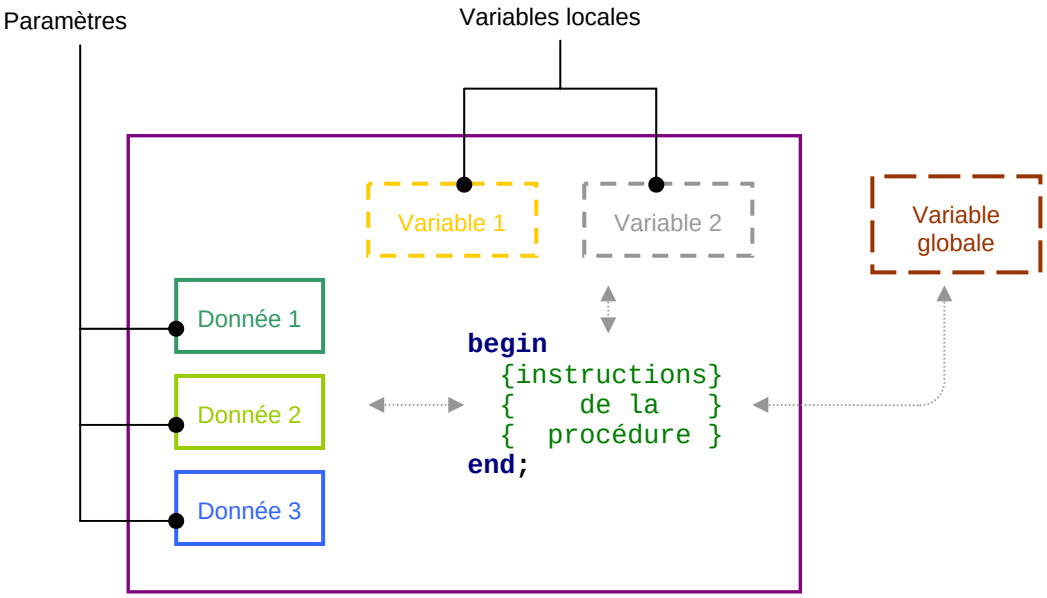


Passage de paramètres à une procédure

L'entrée des données dans la procédure est appelée formellement le *passage de paramètres à une procédure.*

Les données entrantes sont alors *placées dans des variables*. Ces variables sont les *paramètres* ou *arguments*. Ainsi, la procédure peut manipuler :

- la valeur des paramètres qui viennent *d'entrer* ;
- les variables locales ;
- les variables globales.



Les données accessibles par la procédure en fonctionnement

Remarquons qu'une fois les paramètres entrés dans la procédure, *ils ne peuvent plus ressortir*. Cela signifie que la procédure *ne peut pas renvoyer de données* au programme.

3.2 Structure d'une procédure

La *structure générale* d'une procédure est la suivante :

```
procedure {nom} ({paramètres séparés par ;});
  var
    {déclaration des variables locales}
  begin
    {instructions}
  end;
```

Cette structure doit être placée après la déclaration des *variables globales*, de la manière suivante – revoir le chapitre 1.1. :

```
program {nom du programme};
var
  {déclaration des variables globales}

procedure {nom} ({paramètres séparés par ;});
  var
    {déclaration des variables locales}
  begin
    {instructions de la procédure}
  end;

begin
  {instructions du programme en lui-même}
end.
```

Les paramètres sont des variables, chaque paramètre est donc d'un type prédéfini.
On peut utiliser autant de paramètres qu'on le souhaite, voire pas du tout – dans ce cas, la parenthèse qui suit le nom de la procédure est laissée vide.

Par exemple, si l'on souhaite utiliser deux paramètres, l'un de type Integer et l'autre de type Real, on pourra utiliser la syntaxe suivante :

1er paramètre de type Integer *2ème paramètre de type Real*

```
procedure Exemple(param1:integer; param2:real);
```

Le *premier* paramètre est nommé `param1`, le *deuxième* `param2`. Ces paramètres sont utilisables comme des *variables* par la procédure, excepté qu'elles contiennent des valeurs *fournies par le programme*.

L'ordre dans lequel apparaissent les paramètres est important ; cet ordre est en effet utilisé lorsque l'on souhaite exécuter la procédure en lui fournissant les valeurs des paramètres – c'est-à-dire lorsque l'on *appelle* la procédure.

Exercice d'application

Créer une procédure permettant d'afficher *n* fois un *message*. La procédure prendra donc comme paramètres un entier et une chaîne de caractères.

Comme pour un programme, il est recommandé de commencer par écrire la structure d'une procédure.
Il y a deux paramètres – ou arguments – que le programme va devoir fournir à la procédure :

- un entier, le paramètre est donc de type Integer. Nous appellerons ce paramètre n ;
- une chaîne de caractères, le paramètre est de type String. Nous l'appellerons `message`.

L'ordre des paramètres est laissé au choix, on prendra ici n en premier, `message` étant le deuxième paramètre.

Afficher n fois `message` nécessite l'utilisation d'une boucle, et donc d'une variable compteur. On définira la variable `m_i` comme variable compteur locale. Rappelons que par convention dans ce mini-cours, le préfixe `m_` signifie qu'il s'agit d'une variable locale.

Le nom choisi pour cette procédure est `AfficherMessage`. Il doit être suffisamment clair pour pouvoir être utilisé facilement dans le programme.

```
procedure AfficherMessage(n:integer; message:string);
  var
      m_i : integer;

  begin
      for m_i := 1 to n do
            WriteLn(message);
  end;
```

3.3 Appel d'une procédure

Une procédure n'est jamais exécutée spontanément par l'ordinateur, qui commence l'exécution du programme au **`begin`** du programme en lui-même.

Il est ainsi nécessaire d'*appeler la procédure*, c'est-à-dire demander à l'ordinateur de l'exécuter. Un appel peut se faire n'importe où dans le programme, dans d'autres fonctions et même dans la procédure en elle-même – voir le chapitre sur la récursivité. Au moment de l'appel, il est nécessaire de fournir à la procédure les valeurs des paramètres – revoir le schéma *Passage de paramètres à une procédure*.

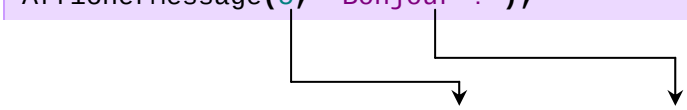
On utilisera alors la syntaxe suivante :

```
{nom de la procédure}({valeur des paramètres séparées par ,});
```

Les valeurs des paramètres, *à placer dans l'ordre des paramètres*, sont séparées par une virgule et sont placées entre parenthèses. On retrouve la syntaxe de l'instruction *WriteLn*.

Ainsi, si nous souhaitons afficher 5 fois le message `Bonjour !` à l'écran en utilisant la procédure `AfficherMessage` créée dans l'exercice précédent, on écrira :

```
AfficherMessage(5, 'Bonjour !');
```



```
procedure AfficherMessage(n:integer; message:string);
```

Ainsi, dans la procédure, le paramètre n vaudra 5 et `message` vaudra `Bonjour !`.

Il peut arriver qu'une procédure ne prenne pas de paramètres. Dans ce cas, on ne placera *pas de parenthèses* à droite du nom de la procédure, puisqu'il n'y a pas de paramètres.

Dans le cas précédent, nous avons passé *directement* le *nombre* 5 et la *chaîne de caractères* `Bonjour !` à la procédure. Il est possible – et courant – de passer des *variables* à la procédure. Bien entendu, c'est alors le *contenu* de ces variables qui est passé, le contenu des variables est ainsi copié dans les paramètres.

Prenons un exemple où nous avons déclaré une variable `i` de type Integer. Cette variable vaut 10. On pourrait alors utiliser la syntaxe :

```
AfficherMessage(i, 'Bonjour !');
```

Cela provoquerait 10 fois l'affichage de `Bonjour !`.

Il faut bien comprendre ce qui se passe pour l'ordinateur : le *contenu* de `i`, c'est-à-dire la valeur 10, est *copiée* dans le paramètre `n` de type Integer. Ainsi, si *dans la procédure*, on modifie la valeur de `n`, *on ne changera pas* pour autant la valeur de `i`.
Cela signifie que le passage de paramètres permet au programme d'entrer des données dans la procédure, mais cette dernière ne peut renvoyer de données vers le programme. Nous allons à présent voir comment on peut renvoyer des données vers le programme, soit *directement* à l'aide d'une *fonction*, soit *indirectement*.

4■ Renvoi *direct* – fonctions

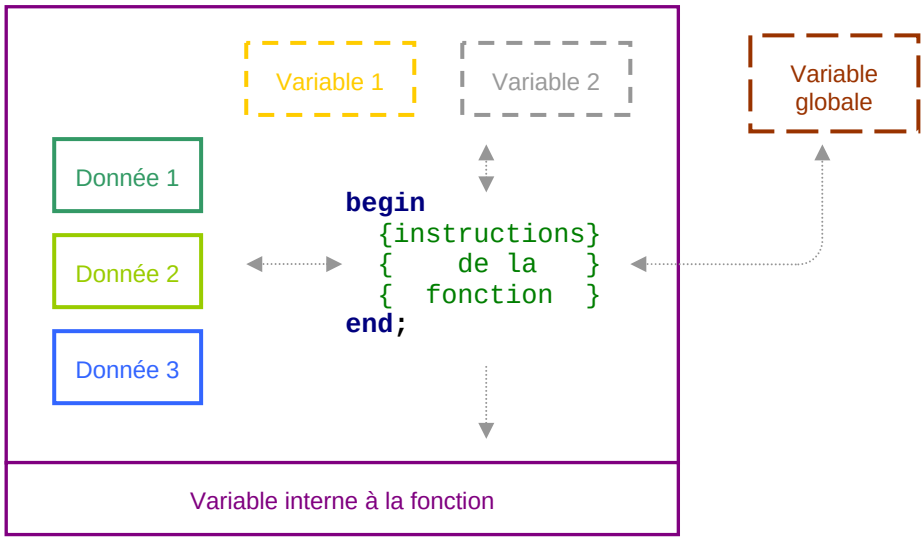
Nous avons vu qu'une procédure peut recevoir des données, mais ne peut en renvoyer vers le programme. Cependant, il est souvent utile d'en renvoyer ; par exemple si l'on souhaite renvoyer le résultat d'un calcul vers le programme.

Nous traiterons dans cette section d'un mécanisme que l'on appelle *renvoi direct*. Il permet de renvoyer directement *un seul et unique résultat* vers le programme ; et ne peut être utilisé qu'avec des *fonctions* – et non les procédures.

`Remarque.` Nous présenterons ici la manière de *renvoyer* un résultat au programme de la manière prévue par le langage Pascal. N'oublions cependant pas que les fonctions peuvent accéder aux variables *globales* du programme et peuvent par conséquent renvoyer un résultat très simplement en changeant le contenu de ces variables.

4.1 Renvoyer le résultat

Le concept est que l'on va associer *une* variable à la fonction.



Le résultat à renvoyer est placé dans cette variable interne. Le programme pourra alors récupérer le résultat stocké dans cette variable.

Cette variable – que l'on nommera aussi *variable spéciale* – possède bien sûr un type. Ce dernier dépend du type de résultat à renvoyer. Il se définit lors de la déclaration de la fonction, de la manière suivante :

```
function {nom}({paramètres}) : {type de résultat à renvoyer};
```

Comment placer un résultat dans cette variable spéciale ? Il suffit de considérer que cette variable possède comme nom *le nom de la fonction*. Ainsi, si nous avons créé une fonction `Test`, il suffit d'écrire :

```
Test := 5;
```

si l'on souhaite renvoyer le résultat 5, exactement comme si `Test` était une variable.

4.2 Récupérer le résultat

Le résultat, placé dans la variable spéciale, est renvoyé au programme lorsque l'on appelle la fonction.

Reprenons notre fonction `Test` qui renvoie 5. Déclarons une variable de type *Integer* nommée `Nombre`. Alors, si l'on écrit dans le programme en lui-même :

```
Nombre := Test({valeurs des paramètres de Test});
```

`Nombre` vaudra 5.

Cela signifie que l'entité :

nom_de_la_fonction(valeur_des_paramètres)

est la variable spéciale appartenant à la fonction que l'on vient d'appeler – rappelons que l'utilisation de cette syntaxe déclenche l'exécution de la fonction par l'ordinateur et constitue un *appel*.

`Attention.` Ce que *nous* appelons la variable spéciale n'est pas une vraie variable, il ne s'agit que d'un *concept* permettant de renvoyer une donnée. Ainsi, le contenu de cette variable ne peut être lu *qu'une fois* – à l'appel de la fonction en utilisant l'entité ci-dessus – et surtout, *son contenu est effacé* à chaque nouvelle exécution de la fonction.

`Attention.` Insistons sur le fait qu'il ne faut pas confondre fonction et procédure. Une *fonction* renvoie *toujours* un résultat par sa variable spéciale, alors qu'une *procédure* ne possède pas de variable spéciale.

Cela signifie qu'une déclaration du type :

```
function Exemple(arg1:integer);
```

provoquera une erreur, puisque l'on n'a pas précisé le type de la variable spéciale.

Exercice d'application

Créer un programme qui demande à l'utilisateur deux entiers a et b, et qui affiche a^b. Bien entendu, dans le cadre de ce chapitre, il est nécessaire de créer une fonction !

Il faut toujours commencer par découper le programme en plusieurs étapes qui constitueront autant de fonctions.
Ici, le problème est simple et nous n'allons créer qu'une fonction : celle qui calcule a^b et qui renvoie le résultat.

Cette fonction prend deux arguments entiers : `m_a` et `m_b`, et renvoie un seul résultat de type entier.
Calculer a^b nécessite une boucle – nous utiliserons ici une boucle *For*. La fonction nécessite donc une variable compteur locale que nous appellerons `m_i` ; ainsi qu'une variable locale pour stocker le résultat au fur et à mesure qu'il est multiplié par `m_a`, nous appellerons cette variable `m_result`.

On obtient la fonction :

```
function Puissance(m_a:integer; m_b:integer):integer;
  var
    m_i, m_result : integer;
  begin
    m_result := 1;
    For m_i := 1 to m_b do
            m_result := m_result * m_a;

      Puissance := m_result;
  end;
```

L'appel se fait dans le programme en lui-même de la manière suivante :

```
Program Exercice;
var
  a, b : integer;

{fonction}

begin
      WriteLn('Entrez a et b.');
      ReadLn(a, b);
      WriteLn('Le résultat vaut : ', Puissance(a, b));
end.
```

Remarquons que l'on a déclaré les variables globales a et b destinées à contenir ce que l'utilisateur vient d'entrer. Lors de l'appel de la fonction, *le contenu de* a *est copié dans* `m_a`, et de même pour b dans `m_b`. Ainsi, la fonction peut travailler *avec ses propres variables* et ne pas influer sur le reste du programme, même si elle modifie ses variables.
L'erreur à ne pas commettre est de nommer les arguments de `Puissance` a et b, comme les deux variables globales : lorsque l'on parlerait de a, l'ordinateur ne saurait pas si l'on parle de la variable globale, ou du paramètre de `Puissance`.

5■ Renvoi *indirect* – passage par adresse

5.1 Présentation

Lorsqu'il s'agissait de renvoyer un seul résultat, les choses restaient simples car nous avons introduit une variable spéciale, intimement liée à la fonction.
Or, une variable ne peut contenir *qu'une donnée*.
Pour renvoyer plusieurs résultats, il va falloir utiliser un autre mécanisme que l'on appelle renvoi *indirect* ou *passage par adresse* et qui fonctionne non seulement pour les fonctions mais aussi pour les procédures.

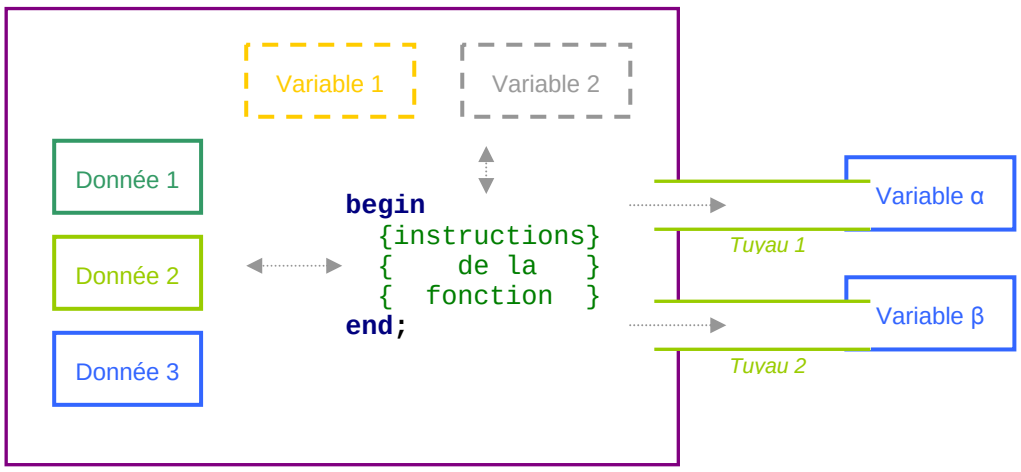
Cependant, le concept de la variable spéciale permettant un renvoi *direct* est toujours valable, ainsi :

- si l'on utilise une *fonction*, *l'un des résultats* sera renvoyé comme pour une fonction standard, par un renvoi direct utilisant la variable spéciale. Les autres résultats seront renvoyés indirectement par le *passage par adresse*.
- si l'on utilise une *procédure*, on n'utilisera que le *passage par adresse* pour renvoyer *tous* les résultats.

L'idée du *passage par adresse* est que l'on va permettre à la fonction d'accéder à des variables auxquelles elle n'aurait normalement pas accès, soit parce qu'elle n'en a pas le droit – variables locales d'autres fonctions – soit parce qu'elle n'en connaît pas le nom. Or, le programme en lui-même a lui aussi accès à ces variables. Ainsi, la fonction pourra renvoyer des données en modifiant la valeur des ces variables.

On pourra objecter que cela est déjà possible, puisque la fonction peut accéder aux variables globales et, en les modifiant, renvoyer un résultat. Certes, mais dans ce cas, la fonction doit connaître le nom des ces variables globales. L'idée du passage par adresse est de permettre à la fonction d'accéder à des variables *sans en connaître le nom*.

Pour ce faire, on va connecter la fonction à un "tuyau" qui est relié à une variable à laquelle la fonction n'a pas accès – souvent parce qu'elle n'en connaît pas le nom.
Chaque tuyau possède un nom, ainsi, *sans connaître le nom de la variable qu'elle modifie* mais en connaissant seulement le nom du tuyau, la fonction peut renvoyer des résultats.



Ce système est appelé *passage par adresse.* L'intérêt est qu'à présent, la fonction ne s'occupe plus de savoir quelle variable elle modifie ; elle ne se préoccupe que du nom du tuyau.
Il faut bien comprendre que, normalement, une fonction doit pouvoir être utilisée *dans n'importe quel programme*. Avec ce système, on s'assure que la fonction accède aux variables auxquelles le programmeur souhaite que la fonction accède, et ceci quel que soit le programme, *quelles que soient les variables globales déclarées*.

`Remarque.` Nous avons parlé de *tuyau* pour faire comprendre le principe du passage par adresse. Ce vocabulaire est utilisé *à but pédagogique*, il faudrait normalement parler de *pointeur*. Cependant, nous continuerons à utiliser ce mot afin de bien faire comprendre ce qui se passe sans entrer dans des considérations par trop techniques.
Attention cependant à ne pas utiliser ce mot dans une copie, il ne veut rien dire en lui-même !

5.2 Mise en œuvre

En pratique, un *tuyau* s'utilise exactement comme une variable ou plutôt comme un paramètre que l'on peut modifier.

Admettons que l'on souhaite créer une procédure `Test` qui prend *un* paramètre et qui va renvoyer, grâce à un passage par adresse, *deux* résultats. On n'utilise pas la variable spéciale dans cet exemple, d'où l'appellation *procédure*.
On la *déclarera* de la manière suivante, avec un **`var`** précédant le nom du tuyau :

```
procedure Test(arg1:integer; var tuyau1:integer; var tuyau2:real);
```

On constate que le premier tuyau est nommé `tuyau1`, le second `tuyau2`. Il est nécessaire de préciser le type de variable auquel le tuyau va être relié ; ici `tuyau1` peut être relié à une variable de type *Integer* et `tuyau2` à une variable de type *Real*.

Lors de l'appel de la procédure, on connecte `tuyau1` et `tuyau2` à des variables de la manière suivante :

```
var
  vInt : integer;
  vReal : real;
begin
      {...}
      Test(12, vInt, vReal);
```

Ici, dans la procédure :

- `arg1` vaut 12
- `tuyau1` est relié à `vInt` ;
- `tuyau2` est relié à `vReal`.

Il est alors possible d'utiliser `tuyau1` *comme si c'était une variable* de type *Integer*, et `tuyau2` comme une variable *Real*. Cependant, *toute modification faite sur ces deux pseudo-variables entraînera en réalité la modification des variables* `vInt` et `vReal`.
De cette manière, la procédure peut placer des résultats dans ces deux variables en utilisant seulement les noms `tuyau1` et `tuyau2`, sans s'occuper de savoir quelles variables elle modifie réellement.

Afin d'éviter que le nom des tuyaux ne soit le même que celui des variables vers lesquelles ils sont connectés, on placera *par convention* le préfixe `p_` avant le nom des tuyaux – `p` pour pointeur.

Exercice d'application

Créer une fonction qui renvoie par renvoi *indirect* deux entiers positifs `a` et `b`, entiers qu'elle demande d'entrer à l'utilisateur. Si `a` et `b` sont positifs, la fonction doit renvoyer par un renvoi *direct* **false**, si `a` *ou* `b` est négatif, alors elle doit renvoyer **true**.

Commençons par écrire la première ligne de la fonction :

- la fonction ne demande pas de paramètres ;
- elle renvoie par un *passage par adresse* deux entiers, on nommera les tuyaux pouvant être reliés à des variables de type *Integer* `p_a` et `p_b` ;
- la fonction – ou sa variable spéciale – doit être de type *Boolean* (**true** ou **false**).

```
function Demande(var p_a:integer; var p_b:integer):boolean;
```

Lors de l'exécution de la fonction, `p_a` et `p_b` sont connectés à des variables définies ailleurs dans le programme, ces variables étant de type *Integer*. Il suffit donc de remplir ces variables en demandant deux entiers à l'utilisateur.
Ensuite, on effectue un test **if** sur la valeur de `p_a` et `p_b` afin de vérifier que les valeurs sont positives. Si c'est le cas, on place **false** dans la variable spéciale, sinon on place **true** dans cette variable.

On obtient alors :

```
function Demande(var p_a:integer; var p_b:integer):boolean;
  begin
      WriteLn('Veuillez entrer a et b.');
      ReadLn(p_a, p_b);

      if (p_a >= 0) and (p_b >= 0) then
            Demande := false
      else
            Demande := true;
  end;
```

Exercice d'application

A l'aide de la fonction crée dans l'exercice précédent, on souhaite réaliser un programme qui demande deux entiers `a` et `b` positifs à l'utilisateur. Si ces entiers sont effectivement positifs, alors on affiche a^b à l'aide de la fonction `Puissance` prenant comme paramètres deux entiers et renvoyant un entier – fonction qu'il n'est pas demandé de créer. Si `a` ou `b` est négatif, alors le programme doit afficher un message d'erreur.

Tout d'abord, il faut réfléchir aux variables dont nous aurons besoin.
La fonction `Demande` demande deux nombres à l'utilisateur, qu'elle renvoie dans deux variables de type *Integer* qui doivent déjà exister. On déclarera donc deux variables de type *Integer*, `a` et `b`.

On sait que si les deux nombres qu'a entré l'utilisateur sont positifs, `Demande` renverra **false**. Par conséquent, il ne faut appeler `Puissance` et afficher le résultat que si `Demande` renvoie **false**. Si `Demande` renvoie **true**, alors on affiche un message d'erreur.

On obtient donc le programme suivant :

```
Program Exercice;
var
  a, b : integer;

{fonctions}

begin
  if Demande(a, b) = true then
    WriteLn('Les entiers doivent être positifs.')
  else
    WriteLn('Le résultat vaut ', Puissance(a, b));
end.
```

A ce moment, `Demande` est exécutée ; c'est-à-dire que la demande de `a` et `b` est faite à l'utilisateur. Les entiers entrés sont placés, par l'intermédiaire des tuyaux `p_a` et `p_b`, dans `a` et `b`. `Demande` renvoie alors **true** ou **false**, suivant les valeurs des entiers.

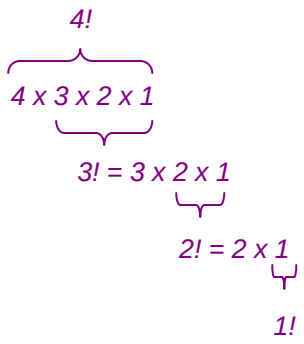
Le programme est très court et compréhensible, d'où l'intérêt d'utiliser des fonctions pour découper le programme en petits éléments.

6■ La récursivité

Nous avons précédemment précisé qu'il était possible qu'une fonction s'appelle *elle-même.* A priori, l'intérêt peut paraître limité, mais la *récursivité* permet d'effectuer certains calculs mathématiques, en particulier avec *les suites définies par récurrence*.

Par exemple, on souhaite calculer la factorielle d'un nombre entier *en utilisant la récursivité*.

Pour cela, on peut procéder de la manière suivante – par exemple si l'on souhaite calculer la factorielle de 4 :



Dans ce calcul, on constate une récurrence : pour calculer *4!*, il suffit de calculer *3!*, puis *2!*, et ainsi de suite.

Créons une fonction `Factorielle` acceptant comme paramètre un entier *n*, et renvoyant l'entier *n!*. On voit qu'il suffit que la fonction s'appelle elle-même suivant ce même principe pour calculer facilement la factorielle de n'importe quel nombre entier.

On voit cependant qu'arrivé au calcul de *0!*, il est nécessaire que la fonction renvoie 1 – par convention. En d'autres mots, *il ne faut pas que l'appel de* `Factorielle(0)` *provoque l'appel de* `Factorielle(-1)` mais renvoie directement 1.
Il ne faut pas, en effet, que cet enchaînement provoque une *infinité d'appels*, ce qui provoquerait un plantage du programme.

On utilisera donc la fonction suivante :

```
function Factorielle(n:integer):integer;
begin
      if n > 0 then
            Factorielle := n * Factorielle(n-1)
      else
            Factorielle := 1;
end;
```

Il faut ici faire la différence entre `Factorielle` qui permet de renvoyer un résultat, et `Factorielle(n-1)` qui constitue un appel de la fonction.

`Remarque.` La récursivité, issue des mathématiques, n'est pas un outil indispensable en programmation. En effet, tout ce qui peut être réalisé grâce à une récursivité peut également être réalisé – bien plus simplement pour l'ordinateur – par une ou plusieurs boucles.
Ainsi, dans la pratique et sauf cas exceptionnel, on évitera l'utilisation de la récursivité.

CHAPITRE 3

Types avancés

Jusqu'à maintenant, nous n'avons utilisé que des *types simples*. Dans ce chapitre, nous allons voir qu'il existe d'autre types plus complexes et moins utilisés :

- les intervalles
- les énumérations
- les ensembles.

De plus, il est possible de définir soi-même *de nouveaux types de variables* qui sont appelée *enregistrements*.

Tous ces *types* se déclarent dans un bloc **type**, celui-ci étant placé *juste avant* le **var** :

```
type
  {déclaration des types}

var
  {déclaration des variables}
```

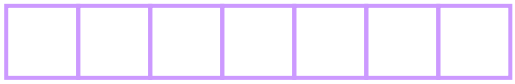
On notera qu'il est possible de définir des types *locaux* – c'est-à-dire déclarés *dans* une fonction – ne pouvant être utilisés que dans la fonction ; et des types *globaux* pouvant être utilisés dans tout le programme.

1■ Les tableaux à plusieurs dimensions

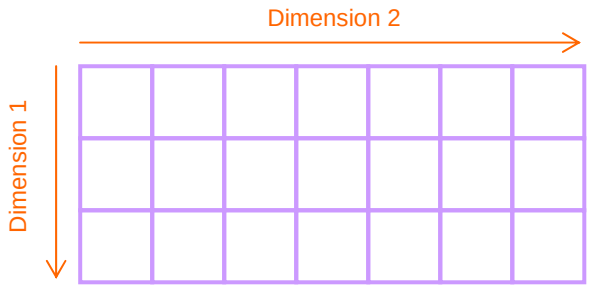
1.1 Présentation

Les tableaux à plusieurs dimensions constituent un type simple et doivent donc être déclarés dans un bloc **var**.

Nous n'avons étudié jusqu'à présent que les tableaux *à une dimension* :



Il est possible de généraliser la notion de tableau pour créer des *tableaux à plusieurs dimensions*. Ainsi, un tableau à deux dimensions constitue une *matrice* :



Matrice 3x7 `Exemple`

Il est bien sûr possible de créer des tableaux à trois dimensions et plus.
Cependant, dans la pratique, de tels tableaux sont très peu utilisés ; nous ne présenterons ainsi que les tableaux à deux dimensions.

1.2 Déclaration d'un tableau à deux dimensions

Comme pour les tableaux à une dimension, les *tableaux à deux dimensions* se déclarent dans un bloc **`var`** de la manière suivante :

```
var
  {nom} : array[{1ère dimension}, {2ème dimension}] of {type};
  Exemple : array[1..3, 1..7] of integer;
```

La syntaxe utilisée est donc très proche de celle de la déclaration d'un tableau à une dimension.

`Remarque.` Rappelons que le nombre de cases d'un tableau est défini une fois pour toutes lors de la déclaration et *ne peut être changé*.

1.3 Utilisation d'un tableau à deux dimensions

Un tableau à deux dimensions s'utilise exactement de la même manière qu'un tableau à une dimension, excepté que les éléments du tableau sont désignés par :

```
nom du tableau[dimension 1][dimension 2]
```

On remarquera qu'il est souvent nécessaire, avec ce type de tableau, d'utiliser des *boucles imbriquées*. Par exemple, si l'on souhaite afficher l'intégralité du tableau à l'écran – ici sous forme de matrice – on devra utiliser :

```
Program Exemple;
var
  i, j : integer;

begin
      for j := 1 to 7 do                    // lit dimension 2
      begin
            for i := 1 to 3 do              // lit dimension 1
                  Write(Exemple[i][j], ' ');
            WriteLn(' ');                   // passe à la ligne
      end;
end.
```

1.4 Utilisation avec les fonctions

Nous avons vu, dans le chapitre précédent, comment passer des paramètres à une fonction ; puis comment on pouvait renvoyer des résultats au programme.

Il est possible d'utiliser un tableau – à simple voire double dimension – *comme paramètre d'une fonction ou d'une procédure* ; par ailleurs, une fonction peut effectuer le *renvoi direct* d'un tableau, et on peut aussi renvoyer un tableau par *renvoi indirect*.

Pour cela, il est nécessaire de définir un type pour le tableau que l'on souhaite utiliser avec la fonction, qui se réalise dans un bloc **type** :

```
type
  {nom du type} = array[{intervalle}] of {type des cases};
  t_arrTableau = array[1..12] of integer;
```

On pourra alors utiliser le nom du type de tableau comme type d'un paramètre, d'une variable spéciale ou d'un tuyau.

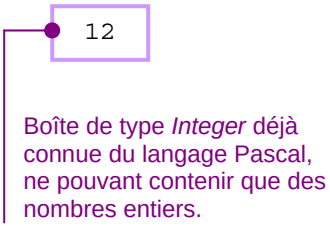
2■ Notion de type

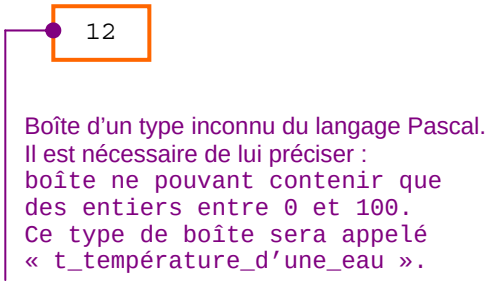
Avant de continuer, il faut bien comprendre la différence entre les types que l'on étudie à présent, et les types *simples* étudiés précédemment.

Plaçons nous dans le cas d'un type simple – par exemple le type *Integer*.
Ce type est *prédéfini dans le langage* Pascal, ce qui signifie qu'il est possible de déclarer *directement* une variable de type *Integer*, puisque l'ordinateur connaît déjà ce type.

Or, les types que nous allons étudier à présent – prenons l'exemple du type *intervalle* – ne *sont pas prédéfinis* dans le langage Pascal, il n'est donc pas possible de déclarer une variable de type intervalle *directement*.
Avant de pouvoir le faire, il est nécessaire de *déclarer le type* intervalle, ce qui permettra alors de déclarer une variable de ce type.

Plus visuellement, on pourra se représenter le fait que le type de boîte – variable – contenant les données n'est pas connu du langage Pascal ; il faut donc lui préciser les spécifications de la boîte que l'on va utiliser avant de pouvoir l'utiliser pour placer des données dedans.





Ainsi, tout le travail va consister à fournir à l'ordinateur les spécifications de la boîte que l'on veut utiliser. On dit alors que l'on *déclare un type*.

Une fois *un type* de variable déclaré, il est possible de l'utiliser pour plusieurs variables, exactement comme pour un type simple.
Il est donc *absolument nécessaire* de faire la différence entre :

- le type, qui constitue la spécification d'un type de boite ;
- les variables de ce type, qui sont les boîtes en elles-mêmes.

Dans notre exemple visuel, le *type* s'appelle `t_température_d'une_eau`, la *variable* contenant `12` peut quand à elle prendre n'importe quel nom – par exemple `Test`.

Afin de ne pas faire la confusion, nous ajouterons *par convention* le préfixe `t_` avant le nom des types.

3■ Les intervalles

3.1 Déclaration – utilisation

Un intervalle *est un type de variable* et non une variable.

Les variables de *type intervalle* sont en réalité des variables de type *Integer* pouvant contenir des entiers *contenus dans un certain intervalle*.

La déclaration de ce type se fait de la manière suivante :

```
type
  {nom} = {min}..{max};
  t_exemple = 0..100;
```

Attention, *on utilise le signe égal* **=**, comme pour toutes les déclarations de type.
Ici, `t_exemple` n'est pas une *variable* mais un *type*.

Si l'on souhaite déclarer des variables de type `t_exemple` – c'est-à-dire pouvant contenir des entiers compris entre 0 et 100 inclus – il faudra faire la déclaration comme si l'on utilisait un type standard, c'est-à-dire :

```
var
  Essai, Test : t_exemple;
```

Où `Essai` et `Test` sont deux variables de type `t_exemple`.

3.2 Utilisation pour déclarer un tableau

Il est également possible d'utiliser un *type* intervalle pour déclarer un tableau. Ainsi :

```
type
  t_exemple = 0..100;

var
  Tableau : array[t_exemple] of integer;
```

`Tableau` est un tableau contenant 100 cases.

L'utilisation de cette syntaxe n'est toutefois pas recommandée, en raison du fait que son utilisation systématique conduit à des programmes difficiles à relire – par ailleurs, elle présente un intérêt limité.

4■ Les énumérations

4.1 Variable de type énumération

Une variable de type *énumération* est en réalité une variable de type *Integer*, c'est-à-dire contenant des nombres entiers.
La différence avec une variable de type *Integer* est que *chaque entier* que peut contenir une variable de type énumération *possède un nom*. Le nom de chaque entier est fixé lors de la déclaration du type énumération.

Prenons un exemple : on déclare un type énumération `t_jour` qui associe :

- `0` est appelé `lundi` ;
- `1` est appelé `mardi` ;
- `2` est appelé `mercredi` ;
- `3` est appelé `jeudi` ;
- `4` est appelé `vendredi` ;
- `5` est appelé `samedi` ;
- `6` est appelé `dimanche`.

Déclarons alors une variable `Test` de type `t_jour`.
Il sera alors possible d'utiliser une syntaxe du type :

```
Test := lundi;
```

En réalité, `Test` contiendra *dans la mémoire de l'ordinateur* la valeur `0`.
Cela signifie que, *d'une certaine manière*, la variable `Test` est une variable de type *Integer* ne pouvant contenir que des entiers compris entre 0 et 6 inclus.

`Attention.` Une variable de type énumération contient effectivement un entier, mais *ce dernier n'est pas accessible directement* en tant qu'entier. *Il est nécessaire* d'utiliser les noms définis lors de la déclaration du type *énumération*.
Ainsi, utiliser une syntaxe du type `Test := 1;` provoquera une erreur `Types incompatibles : t_test et Integer`. On ne pourra pas non plus afficher le contenu d'une variable de type énumération avec *WriteLn*.

4.2 Déclaration d'un type énumération

Lors de la déclaration d'un *type énumération*, il est nécessaire de définir les *noms* que l'on va associer aux entiers que pourra contenir une variable de ce *type*.
Les entiers sont fixés automatiquement *en commençant par 0* : ainsi, le *premier* nom donné correspondra à `0`, le *second* à `1`, etc.

Cela se fait dans un bloc **`type`** de la manière suivante :

```
type
  {nom} = ({noms séparés par ,});
  t_niveau = (facile, moyen, difficile);
```

Ici, on a déclaré le *type* `t_niveau` où

- `0` est appelé `facile` ;
- `1` est appelé `moyen` ;
- `2` est appelé `difficile`.

4.3 Utilisation

Une fois un *type* énumération déclaré, il est possible de déclarer des *variables* de ce type de la manière habituelle, dans un bloc **`var`**.

Quand utilise-t-on des variables de type énumération ? De manière générale, à chaque fois que l'on souhaite utiliser des variables caractérisant *une entité possédant un état parmi d'autres* ; et *un seul état* en même temps.
Par exemple :

- le temps qu'il fait (il ne fait pas *à la fois* beau et mauvais) ;
- l'état d'une espèce chimique (solide, liquide, gaz) ;
- le niveau d'un jeu (facile, moyen, difficile).

On notera que les variables de type simple *Boolean* (contenant soit **true**, soit **false**) sont d'une certaine manière des variables de type énumération, au sens où :

- la variable ne peut valoir à la fois **true** et **false** ;
- en réalité pour l'ordinateur, **false** correspond à `0` et **true** correspond à `1`.

4.3.1 a) Comparaisons

Nous avons précédemment vu qu'une variable de type énumération est en réalité *pour l'ordinateur* une variable de type *Integer*.

Cela signifie qu'il est possible d'effectuer des *opérations de comparaison* sur de telles variables, à l'aide des opérateurs =, <>, <, >, <= et >=.

Par exemple :

```
Program Exemple;
type
  t_niveau = (facile, moyen, difficile);

var
  jeu1, jeu2 : t_niveau;

begin
      jeu1 := difficile;
      jeu2 := facile;
      if jeu1 > jeu2 then
            WriteLn('jeu1 est plus difficile que jeu2.')
      else
            WriteLn('jeu1 est moins difficile que jeu2.');
end.
```

Ici, `jeu1 est plus difficile que jeu2.` s'affichera.

4.3.2 b) Boucle For

Par ailleurs, il est possible d'utiliser une variable de type énumération comme *variable compteur* pour une boucle *For*, de la manière suivante :

```
Program Exemple;
type
  t_niveau = (facile, moyen, difficile);

var
  jeu : t_niveau;

begin
      for jeu := facile to difficile do
            {bloc d'instructions répété ici 3 fois}
end.
```

Lors de la première répétition du bloc d'instructions, `jeu` vaudra `facile`. Lors de la seconde répétition, `jeu` vaudra `moyen` et lors de la troisième, `jeu` vaudra `difficile`.

Cela signifie qu'à chaque répétition, la boucle *For* arrive à faire passer `jeu` *à l'élément suivant*. Nous allons maintenant voir comment on peut réaliser ceci.

4.3.3 c) Incrémentation – décrémentation

Admettons que l'on ait déclaré une variable `Test` de type `t_jour`, et que `Test` contienne un jour de la semaine – on ne sait pas lequel. Comment faire passer `Test` au jour suivant ?

Il n'est pas possible d'utiliser une syntaxe du type :

```
Test := Test + 1;
```

En effet, il n'est pas possible d'utiliser une variable de type énumération comme une variable de type *Integer* – bien que ces variables contiennent des entiers, on ne peut effectuer des *opérations mathématiques* avec mais seulement des *comparaisons*. On dit qu'il y a *incompatibilité des types*.

Il sera alors nécessaire d'utiliser les fonctions :

- `succ(`{variable}`)` qui renvoie l'élément suivant ;
- `pred(`{variable}`)` qui renvoie l'élément précédent.

Remarquons que ces fonctions renvoient un élément *du même type* que la variable de type énumération. L'assignation est alors possible puisqu'il y a *compatibilité des types*.

Ainsi, pour faire passer `Test` au jour suivant, on utilisera :

```
Test := succ(Test);
```

`Attention.` Lorsque l'on utilise ces fonctions, l'ordinateur ne fait qu'incrémenter / décrémenter l'entier contenu dans la variable de type énumération *sans aucune vérification*. Ainsi, admettons que `Test` vaut `dimanche` – c'est-à-dire le nombre 6. Si l'on utilise `succ` pour faire passer `Test` au jour suivant, `Test` vaudra 7 ; or, aucun jour ne correspond à 7 et le contenu de la variable n'a alors plus aucun sens.

`Remarque.` L'utilisation de ces deux fonctions n'est nécessaire *que lorsque l'on a affaire à une variable de type énumération*.

5■ Les ensembles

Un ensemble est une variable très proche d'un tableau, à l'exception que :

- il n'est pas possible de placer deux fois la même valeur dans un ensemble ;
- *seuls des entiers* peuvent être placés dans un ensemble.

Avant d'utiliser un ensemble, il est tout d'abord nécessaire de déclarer un *type ensemble* :

```
type
  {nom} = set of {type d'objets};
  t_ens1 = set of char;
  t_ens2 = set of 1..24;
```

Comme on ne peut placer que des entiers dans un ensemble, *il n'est pas possible de définir un type d'ensemble contenant des réels*. On peut en revanche déclarer des types ensemble qui contiennent des entiers compris dans un certain intervalle, comme le montre l'exemple ci-dessus.

Il est ensuite possible de déclarer des variables de type ensemble, comme d'habitude dans un bloc **var** :

```
var
  ensemble : t_ens1;
```

Pour placer des éléments dans un ensemble, il suffit d'utiliser la syntaxe suivante :

```
{ensemble} := [{éléments séparés par ,}];
ensemble := [1, 3, 5, 7];
```

L'intérêt des ensembles – par ailleurs peu utilisés – réside dans les *opérations* qu'il est possible d'effectuer avec. Sur deux ensembles a et b, on peut ainsi réaliser :

- une *union* : a + b
- une *intersection* : a * b
- une *différence* : a - b

Il est également possible de *vérifier si un élément figure ou non dans un ensemble*, à l'aide de l'opérateur **in** qui peut par exemple être utilisé dans un test *If Then* :

```
if 10 in ensemble then
  {instructions à exécuter si 10 appartient à ensemble}
```

6■ Les enregistrements

6.1 Introduction

Les *enregistrements* constituent un concept extrêmement important de la programmation : *l'objet*, concept que l'on retrouvera sous des formes plus puissantes dans les langages modernes dits *orientés objet* tels que le *C++*, le *Java*.

Concrètement, les "variables de type enregistrement" sont des variables qui ne contiennent non pas une valeur, mais *plusieurs variables*, chacune possédant un nom distinctif.

Dans cet exemple, on a déclaré un *type enregistrement* `t_boite`.
Les *enregistrements* de type `t_boite` – par exemple l'enregistrement `Boite` – contiennent alors quatre variables, qui permettent de définir une boîte :

- sa longeur ;
- sa largeur ;
- sa hauteur ;
- sa couleur.

On retiendra qu'un enregistrement peut contenir des variables de types divers, contrairement à un tableau où toutes les cases sont de même type. Par ailleurs, ces variables sont nommées par un nom et non par un numéro comme c'est le cas pour les tableaux.

`Remarque.` Un enregistrement *n'est pas, à proprement parler*, une variable car un enregistrement ne contient pas directement de valeur. On parlera donc *d'enregistrement* et non de *variable* pour désigner ce que l'on pourrait appeler une "*variable de type enregistrement*".
On prendra garde à ne pas confondre :

- un *enregistrement*, qui contient des variables ; ces dernières contenant des valeurs ;
- le *type enregistrement* qui définit quelles variables contient un certain type d'enregistrement.

6.2 Déclaration du *type*

Un *type* enregistrement se définit dans un bloc **type** de la manière suivante :

```
{nom du type} = record
  {variable} : {type};
  {...}
  end;

t_boite = record
  Longueur, Largeur, Hauteur : integer;
  Couleur : string;
  end;
```

Nous ne saurions insister assez sur le fait qu'à ce niveau, nous n'avons pas encore déclaré d'*enregistrement*, mais simplement un *type* d'enregistrement, c'est-à-dire les spécifications d'un *type d'enregistrements*.

Les variables à l'intérieur de l'enregistrement peuvent être de n'importe quel type ; y compris des tableaux ou des variables comme les énumérations, voire même d'autres enregistrements.

`Remarque.` Les variables utilisées dans un enregistrement peuvent porter les mêmes noms que des variables globales ou locales, car on n'utilisera jamais ces variables en dehors de l'enregistrement.

6.3 Déclaration d'enregistrements

Une fois le *type enregistrement* déclaré, un enregistrement se déclare comme une variable standard, dans un bloc **var**. Dans notre exemple, nous aurions écrit :

```
var
  Boite : t_boite;
```

Il est possible de définir autant d'enregistrements qu'on le souhaite ; il est même possible de déclarer des *tableaux d'enregistrements* où chaque case est un enregistrement – bien que cela soit d'un intérêt limité.

Ainsi, il est possible de définir *plusieurs enregistrements* de type `t_boite`, représentant chacun un type de boite possédant des dimensions ou une couleur différentes.

6.4 Utilisation

Une fois déclaré un enregistrement, il va être nécessaire de manipuler les variables qui se trouvent à l'intérieur. Cela se fait très simplement à l'aide de la syntaxe :

```
{enregistrement}.{variable}
```

Il est alors possible de manipuler les variables à l'intérieur comme des variables standard. On peut par exemple effectuer une *affectation* :

```
Boite.Longueur := 12;
```

Exactement comme sur les tableaux, il est possible d'utiliser deux opérations importantes sur les enregistrements eux-mêmes, *et uniquement ces deux opérations* :

- il est possible d'utiliser l'opérateur d'affectation `:=` pour recopier directement un enregistrement dans un autre ; *à la condition expresse que les deux enregistrements soient de même type* ;
- il est possible d'utiliser un enregistrement comme paramètre d'une fonction, ou d'effectuer un renvoi – direct ou indirect – d'un enregistrement.

Ainsi, s'il est possible d'écrire :

```
Boite1 := Boite2;
```

pour recopier `Boite2` dans `Boite1` si ces enregistrements sont de même type, *il est absolument impossible* d'écrire `Boite1 = Boite2` pour faire une comparaison entre ces deux enregistrements.

Exercice d'application

On souhaite utiliser, dans un programme, des vecteurs. Plutôt que d'utiliser des tableaux de 3 cases, on préfère utiliser des enregistrements de type `t_vecteur` ; les enregistrements de ce type pouvant alors contenir trois réels x, y et z. Créer la déclaration de ce *type*, ainsi qu'une fonction permettant d'effectuer la somme de deux vecteurs (fonction renvoyant donc un vecteur).

Commençons par déclarer le type `t_vecteur` dans un bloc **`type`** :

```
type
  t_vecteur = record
    x, y, z : real;
  end;
```

Il est nécessaire de créer une fonction pour additionner deux enregistrements de type `t_vecteur` car, rappelons-le, il n'est pas possible d'utiliser un opérateur autre que `:=` ; il n'est donc pas possible d'utiliser +.
Lorsque l'on crée une fonction, commencer par réfléchir aux paramètres et aux renvois ; ici :

- la fonction prend deux paramètres, qui sont les deux vecteurs à additionner ; c'est-à-dire deux enregistrements de type `t_vecteur`. Nous appelerons ces paramètres a et b ;
- la fonction renvoie la somme sous forme d'un vecteur, donc un enregistrement de type `t_vecteur`.

Il faut alors bien se remémorer le concept de la variable spéciale qui est manipulable comme une variable ayant comme nom *le nom de la fonction*. Comme nous appellerons la fonction `Somme`, on manipulera la variable spéciale sous le nom de `Somme`.

On obtient alors :

```
function Somme(a:t_vecteur; b:t_vecteur):t_vecteur;
begin
      Somme.x := a.x + b.x;
      Somme.y := a.y + b.y;
      Somme.z := a.z + b.z;
end;
```

Dans le programme, si l'on souhaite placer la composante x de la somme de `vect1` et `vect2` dans la variable `composX` de type *Real*, on pourra utiliser la syntaxe :

```
composX := Somme(vect1, vect2).x;
```

car l'entité `Somme(vect1, vect2)` *est* la variable spéciale de `Somme`, donc *un enregistrement* de type `t_vecteur`.

CHAPITRE 4

Gestion des fichiers

1■ Fichiers texte

1.1 Notion de fichier texte

Le langage Pascal fournit un mécanisme *rudimentaire* permettant de lire et de modifier les *fichiers texte*, fichiers que l'on pourra créer et ouvrir avec le bloc-notes sous Windows, ou l'éditeur *Kate* sous Linux.

Retenons qu'un fichier texte est structuré en lignes :

```
Première ligne
Deuxième ligne
Troisième et dernière ligne
```

Ainsi, les programmes que nous serons amenés à créer ne pourront lire les fichiers texte que *ligne après ligne*.

1.2 Manipulation d'un fichier texte

Avant de pouvoir ouvrir un fichier texte, il est nécessaire de déclarer une variable de type *fichier texte*. Il s'agit d'une variable complexe dont le fonctionnement précis ne nous intéresse pas pour le moment.
Cette variable fichier texte – souvent appelée `F` – se déclare comme toute variable dans un bloc **`var`** :

```
var
  F : text;
```

Lorsque, durant le programme, on souhaite *ouvrir un fichier*, on *associe* cette variable au fichier à ouvrir. Cela se fait de la manière suivante, grâce à l'instruction `Assign` :

```
Assign({nom de la variable}, {nom du fichier});
Assign(F, 'C:\fichier.txt');
```

Il est alors nécessaire de préciser quel type d'opération on va effectuer sur le fichier : lecture ou écriture. *Il n'est en effet pas possible d'effectuer ces deux opérations en même temps sur un fichier texte*.
On utilisera ainsi l'une de ces procédures, au choix :

Ouvrir le fichier *en lecture*	`Reset({variable});`
Créer un nouveau fichier et l'ouvrir *en écriture*	`Rewrite({variable});`
Ouvrir un fichier déjà existant *en écriture*	`Append({variable});`

Lorsque l'on ouvre un fichier avec `Append`, les données que l'on écrira dedans seront toujours placées *à la fin du fichier, après ce qu'il contient déjà*. Si l'on souhaite effacer ces données pour en écrire de nouvelles, on utilisera `Rewrite`.

`Remarque.` La procédure `Rewrite` permet aussi de *créer un nouveau fichier*. Cela signifie que, lorsque l'on utilise `Assign` pour préciser le nom du fichier à ouvrir, le fichier indiqué *peut parfaitement ne pas exister*. Il faudra alors utiliser `Rewrite` pour le créer et l'ouvrir en écriture.

Une fois toutes ces opérations effectuées, le fichier associé à la *variable fichier texte est ouvert*, soit en lecture soit en écriture.
Comme nous l'avons déjà précisé, il est possible de lire ou d'écrire dans le fichier *ligne par ligne*. Pour ce faire, on utilise une variable de type *String* – qui doit avoir été déclarée précédemment dans un bloc **`var`** – qui va contenir la ligne lue ou à écrire.
On utilise alors les procédures suivantes, *suivant le type d'opération permis* :

- `ReadLn({variable fichier}, {variable de type String});`
- `WriteLn({variable fichier}, {variable de type String});`

`Attention.` Ne pas confondre les fonctions *ReadLn* et *WriteLn* prenant *deux* arguments et *servant pour les fichiers texte*, avec les fonctions *ReadLn* et *WriteLn* servant à *interagir avec l'utilisateur* et ne prenant *qu'un argument*.
En particulier, les fonctions *ReadLn* et *WriteLn* utilisées pour les fichiers *prennent deux arguments qui doivent être des variables*. Ainsi, l'écriture de `WriteLn(F, 'Bonjour.');` provoquera une erreur, car `'Bonjour.'` n'est pas une variable.

Une fois le fichier lu – ou écrit – il ne faut pas oublier de le fermer à l'aide de `Close`, sous peine de provoquer un plantage du programme une fois celui-ci terminé.

```
Close({variable fichier});
Close(F);
```

Admettons que l'on ait déclaré la variable `ligne` de type *String*, et qu'on ait associé `F` au fichier suivant :

```
Première ligne
Deuxième ligne
Troisième et dernière ligne
```

Exécutons alors le programme suivant :

```
ReadLn(F, ligne);
                ❶ ligne vaut Première ligne
ReadLn(F, ligne);
                ❷ ligne vaut Deuxième ligne
ReadLn(F, ligne);
                ❸ ligne vaut Troisième et dernière ligne
```

On remarque que le contenu de `ligne` change à chaque `ReadLn` : le fichier est lu *ligne après ligne*.
Notons *qu'il n'est pas possible de revenir en arrière dans le fichier* ; si l'on souhaite relire le fichier, il faut le fermer à l'aide de `Close`, puis le ré-ouvrir.

Dans notre exemple, le fichier texte comporte trois lignes ; ce qui signifie que *l'on ne peut effectuer plus de trois* `ReadLn` *sur ce fichier*.
Si l'on ne sait pas de combien de lignes est constitué le fichier que l'on est en train de lire, comment savoir si l'on se trouve à la fin du fichier ?
Il suffit d'utiliser la fonction `EOF(`{variable fichier}`)` – `EOF` signifiant *End Of File* – qui renvoie **true** lorsque l'on se trouve à la fin du fichier, ou **false** sinon.
Reprenons notre exemple :

```
ReadLn(F, ligne);
                ❶ EOF(F) vaut false
ReadLn(F, ligne);
                ❷ EOF(F) vaut false
ReadLn(F, ligne);
                ❸ EOF(F) vaut true, il ne faut alors plus effectuer de ReadLn
```

`Remarque.` Si l'on crée un fichier avec la procédure `Rewrite` ou bien si l'on ouvre un fichier en écriture avec `Append`, il est évident que l'on se trouve toujours à la fin du fichier et par conséquent, la fonction `EOF` renverra toujours **true**.
Par ailleurs, si l'on ouvre un fichier texte vide, alors `EOF` renverra **true** avant même la première lecture avec `ReadLn`.

1.3 Gestion des erreurs

Lors de l'ouverture d'un fichier, de nombreuses erreurs sont susceptibles de se produire. Par exemple, si le programme tente d'ouvrir en lecture un fichier qui n'existe pas, ou s'il tente d'ouvrir en écriture un fichier déjà ouvert par une autre application, etc.

Si de telles erreurs se produisent, l'ordinateur affiche généralement un message d'erreur et *le programme se termine brutalement*.
Afin d'éviter ceci et de permettre au programme d'afficher lui-même un message d'erreur, on *désactive l'interruption du programme en cas d'erreur* par la *directive* :

```
{$I-}
```

et on la réactive par :

```
{$I+}
```

Le `I` signifie ici *interruption*, le `+` ou le `-` indiquant l'activation ou la désactivation de cette interruption en cas d'erreur.

En général, on n'a besoin de désactiver cette interruption que dans une petite partie du programme, où se fait l'ouverture du fichier. Ainsi, on obtiendra un bloc du type :

```
{$I-}
Assign(F, 'C:\fichier.txt');
Reset(F);
{$I+}
```

`Attention.` Si une erreur se produit dans ce bloc, *rien de visible ne se produira* et le programme continuera à s'exécuter. Or, étant donné qu'une erreur s'est produite pendant l'ouverture du fichier, il est bien évident que le fichier n'a pas été ouvert ; ainsi, utiliser par la suite des instructions comme `WriteLn` ou `ReadLn` sur le fichier peut conduire à un plantage du programme.

Pour savoir si une erreur s'est produite dans ce bloc, on utilise la fonction `IOResult` qui ne prend aucun paramètre. S'il s'est produit une erreur, alors `IOResult` renvoie un chiffre entier différent de `0` – indiquant un *numéro d'erreur* – sinon elle renvoie `0`.
On pourra ainsi obtenir :

```
{$I-}
Assign(F, 'C:\fichier.txt');
Reset(F);
{$I+}
if IOResult = true then
      WriteLn('Erreur durant l''ouverture.')
else
    {lecture du fichier}
```

Exercice d'application

Créer un programme qui affiche le contenu d'un ficher texte ainsi que le nombre de lignes qu'il contient. Ne pas oublier la gestion des erreurs.

Le programme nécessite trois variables : la *variable fichier texte* `F`, une variable `ligne` de type *String* qui va contenir les lignes du fichier texte et une variable `i` de type *Integer* qui va servir à compter les lignes.

On obtient le programme :

```
Program Exercice;
var
  F : text;
  ligne : string;
  i : integer;

begin
      i := 0;
      {$I-}
      Assign(F, 'fichier.txt');
      Reset(F);
      {$I+}
      if IOResult <> 0 then
            WriteLn('Erreur durant l''ouverture.')
      else
      begin
            While EOF(F) = false do
            begin
                  ReadLn(F, ligne);
                  WriteLn(ligne);
                  i := i + 1;
            end;
            Close(F);
            WriteLn(i, 'lignes.');
      end;
end.
```

Ce programme peut être découpé en plusieurs parties :

- l'ouverture proprement dite du fichier avec une gestion des erreurs ;
- un grand bloc *If Then Else* permettant de n'exécuter les instructions de manipulation du fichier que s'il ne s'est pas produit d'erreur lors de son ouverture – le `Close` inclus ;
- dans ce bloc, une boucle *While* permettant d'afficher à l'écran les lignes successives du fichier. Attention à ne pas confondre les instructions *WriteLn* – ou *ReadLn* – différentes : l'une prenant deux paramètres permettant d'écrire dans un fichier, l'autre ne prenant qu'un paramètre permettant d'écrire à l'écran. Remarquons que l'on utilise une boucle *While* et non *Repeat Until* afin que l'instruction *ReadLn* ne soit pas exécutée si le fichier est vide, c'est-à-dire si `EOF` vaut **true** dès le départ.

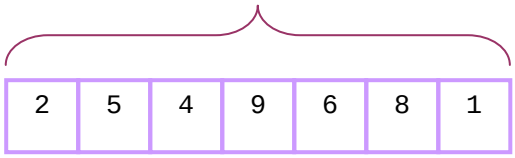
2■ Fichiers typés

2.1 Notion de fichier typé

Jusqu'à maintenant, nous n'avons pu sauvegarder dans des fichiers *que du texte*.
Or, il serait particulièrement utile de pouvoir sauvegarder le *contenu de variables* autres que *String*, en particulier des variables contenant des nombres telles les variables de type Integer ou Real, ainsi que le contenu des enregistrements en une seule fois.

Concrètement, l'idée du *fichier typé* est que l'on va pouvoir utiliser un fichier comme un tableau, et ainsi pouvoir sauvegarder de nombreuses données, *toutes du même type* dans le même fichier.

Fichier `Exemple` composé de 7 cases de type *Integer*



Cependant, contrairement au cas d'un tableau pour lequel le nombre de cases est défini à l'avance, dans le cas d'un fichier typé *il est possible d'ajouter des cases supplémentaires* lors de l'écriture du fichier.

2.2 Manipulation d'un fichier typé

Un fichier typé s'utilise presque de la même manière qu'un fichier texte, hormis quelques détails et surtout le fait que le fichier n'est pas lisible *ligne après ligne* mais *case après case*.

Comme pour un fichier texte, il est nécessaire de déclarer une variable *de type fichier*. Ici, il ne s'agit pas d'une variable de type *fichier texte*, mais d'une variable de type *fichier typé* ; il faut également préciser *le type de données que l'on va placer dans le fichier*. En pratique, on peut placer n'importe quel type de données dans les cases d'un fichier typé, des variables simples aux tableaux en passant par les enregistrements.

Cela se fait de la manière suivante :

```
var
  {nom de la variable} : file of {type de données};
  Exemple : file of integer;
```

Une fois la *variable fichier typé* déclarée, il faut ouvrir le fichier. Contrairement aux fichiers texte où il existe trois types d'ouverture (`Reset`, `Append` et `Rewrite`), il n'en existe ici que deux :

- `Reset(`{variable}`);` utilisé *dans la très grande majorité des cas* ; qui permet de lire et d'écrire dans un fichier si celui-ci existe déjà ;
- `Rewrite(`{variable}`);` si l'on souhaite créer un nouveau fichier puis l'ouvrir.

On remarquera qu'ici, on *ne distingue plus l'ouverture en lecture ou en écriture* ; le fichier est ouvert dans les deux modes à la fois.

La lecture et l'écriture dans un fichier typé se font toujours *case après case*, à l'aide des procédures suivantes :

- `Read(`{variable-fichier}`,` {variable}`)` si l'on souhaite lire le fichier et placer le contenu d'une case dans `variable` ;
- `Write(`{variable-fichier}`,` {variable}`)` si l'on souhaite écrire dans une case du fichier le contenu de `variable`.

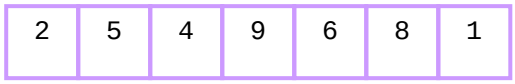
Bien entendu, `variable` doit être du même type que les données qui peuvent être placées dans le fichier typé ; *Integer* dans notre exemple.

De la même manière que pour les fichiers texte, on pourra utiliser la fonction `EOF` ; il ne faudra pas non plus oublier de fermer les fichiers ouverts avec `Close` avant la fin du programme.

2.3 Changement de position courante

Avec les fichiers texte, la lecture – ou l'écriture – se faisait ligne après ligne, *sans aucune possibilité de revenir en arrière* ou de sauter des lignes.
Pour les fichiers typés, le mécanisme est plus souple et bien que la lecture se fasse case après case, il est possible de changer ce que l'on appelle la *position courante*, c'est-à-dire *l'endroit où l'on se trouve dans le fichier* ; de manière à pouvoir revenir en arrière et sauter certaines cases.

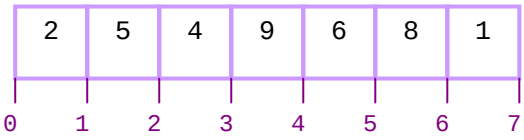
Reprenons notre fichier `Exemple` composé de 7 cases :



Admettons que `i` soit une variable de type *Integer*, et que l'on ait le programme :

```
                ❶ Position courante : 0
Read(F, i);                          // i vaut alors 2
                ❷ Position courante : 1
Read(F, i);                          // i vaut alors 5
                ❸ Position courante : 2
Read(F, i);                          // i vaut alors 4
                ❹ Position courante : 3
```

On voit que la *position* courante *avance automatiquement* en partant de 0. On pourra se représenter la position où l'on en est dans le fichier de la manière suivante :



On peut manipuler la *position courante* à l'aide de trois fonctions importantes :

- `FilePos`
- `Seek`
- `FileSize`

A tout moment, il est possible de connaître la *position courante* dans le fichier grâce à la fonction `FilePos(`{variable-fichier}`)` qui renvoie un entier correspondant à la position courante.

Il est également possible de *changer la position courante* grâce à la fonction `Seek`, afin de pouvoir se déplacer facilement dans le fichier et en particulier de revenir en arrière :

```
Seek(F, 5);
```

Il est souvent souhaitable d'établir la position courante *à la fin du fichier*, afin de pouvoir *rajouter* des cases. Pour cela, on est amené à utiliser la fonction `FileSize(F)` qui renvoie la dernière position du fichier. Dans notre exemple, il s'agit de la position 7.

Ainsi, pour pouvoir ajouter des cases à la fin du fichier, on utilisera :

```
Seek(F, FileSize(F));
{opérations d'écriture rajoutant des cases}
```

CHAPITRE 5

Programmation avancée

Ce chapitre traite de notions qui ne sont pas indispensables pour programmer en Pascal ; mais qui pourront constituer une passerelle vers des langages de programmation plus évolués que vous serez susceptible de rencontrer par la suite.

Nous traiterons dans ce chapitre :

- de la *programmation objet* ou *modulaire*;
- de l'utilisation des variables de type *String*.

1■ Programmation objet

1.1 Introduction

Contrairement à d'autres langages tels que le *C++*, le Pascal offre des mécanismes de programmation objet *rudimentaires*.
En pratique, *en Pascal*, la *programmation objet* consiste à *rassembler ensemble plusieurs fonctions, procédures et variables ayant trait à un même thème*.

Admettons que l'on désire réaliser un programme permettant d'effectuer plusieurs tâches complexes, comme par exemple :

- lire et décoder un fichier ;
- l'afficher d'une certaine manière à l'écran ;
- demander à l'utilisateur ce qu'il souhaite modifier ;
- enregistrer le fichier une fois modifié par l'utilisateur.

Ce programme pourrait être un traitement de texte comme *Word*, mais aussi un éditeur d'images, de pages Web, etc.
Ici, le programme est constitué de quatre tâches qui sont complexes, c'est-à-dire qui vont nécessiter plusieurs fonctions chacune, ainsi que des variables spécifiques à chaque tâche, qui ne concernent pas les autre tâches.

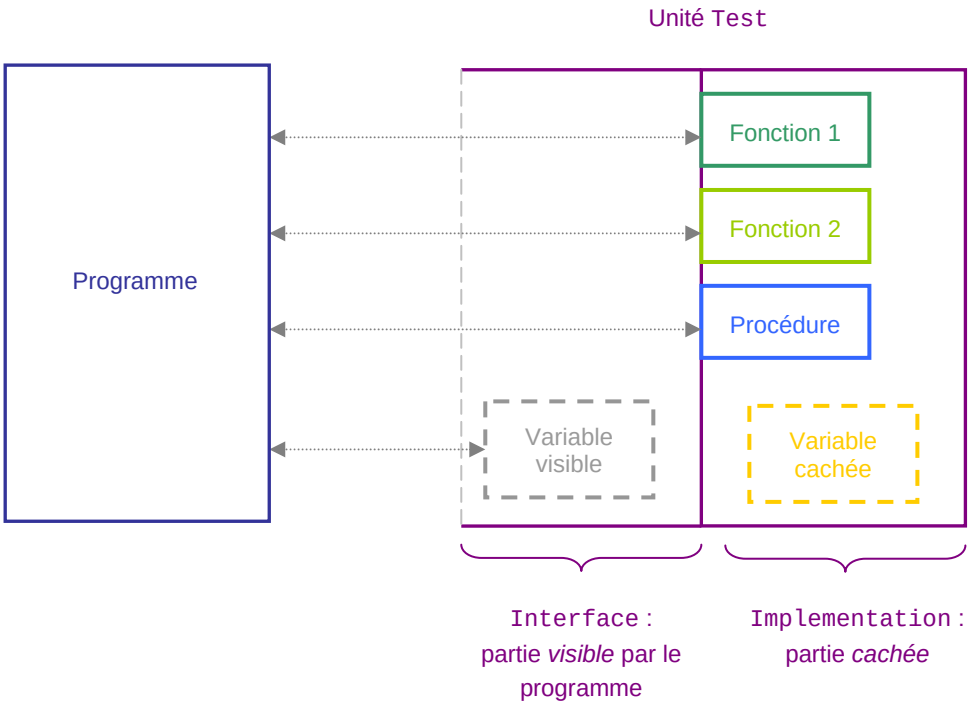
On pourrait écrire toutes ces fonctions et les utiliser telles quelles dans le programme. Cependant, on peut regrouper les fonctions et variables par thème ; il y aurait ici quatre thèmes. L'intérêt – difficilement perceptible à notre niveau – est de pouvoir réutiliser facilement l'une des tâches dans un autre programme, sans devoir recopier indépendamment les fonctions de cette tâche.
Par exemple, si les fonctions concernant la lecture d'un fichier ont été réalisées pour un traitement de texte, on pourra facilement les utiliser lors de la programmation d'un éditeur d'images ; sans avoir à tout réécrire.

1.2 Notion d'unité

Chaque regroupement de fonctions et de variables est placé dans un nouveau fichier et constitue ce que l'on appelle une *unité*.

Nous venons de voir qu'une unité était un regroupement de fonctions et de variables ayant trait à un même thème. Cependant, certaines variables peuvent ne servir *qu'aux fonctions rassemblées dans l'unité*, auquel cas il est possible de *cacher* ces variables au programme principal qui n'en n'a pas besoin.

Visuellement, on peut donc représenter une unité par une *boîte à deux compartiments*, l'un étant totalement caché du programme, l'autre étant au contraire accessible par ce dernier :



Il est entendu que les fonctions appartenant à l'unité peuvent accéder aux variables déclarées *dans* l'unité – ici la *Variable visible* et la *Variable cachée*. En revanche, ces fonctions *ne pourront pas accéder aux variables globales déclarées dans le programme en lui-même*.

Le schéma montre une subtilité concernant les fonctions de l'unité : elles sont contenues dans la partie cachée, mais elles dépassent légèrement dans la partie visible. En effet, si les fonctions étaient complètement dans la partie cachée, le programme ne pourrait y accéder. Cependant, on ne les place pas complètement dans la partie visible car *leur fonctionnement n'intéresse pas le programme* ; ce dernier doit seulement pouvoir *accéder* aux fonctions.

1.3 Mise en œuvre

Rappelons qu'une unité doit être placée *dans un nouveau fichier* ayant comme nom le nom de l'unité et une extension .pas. Dans le cas de notre unité `Test`, on aurait placé sa déclaration dans un nouveau fichier nommé `Test.pas`.

Une *unité* est toujours structurée de la même manière :

```
Unit {nom de l'unité};
interface
  {partie visible}

implementation
  {partie cachée}

begin
  {initialisation des variables}
end.
```

Dans les deux parties **`interface`** et **`implementation`**, on pourra placer – suivant si l'on souhaite qu'ils soient visibles ou non par le programme :

- des déclarations de *variables* ;
- des déclarations de *types* ;
- des déclarations de *constantes*.

Ceci se fait simplement à l'intérieur de blocs **`const`**, **`var`** ou **`type`** comme cela a été expliqué dans le premier chapitre.

En revanche, lorsque l'on souhaite placer des *fonctions* ou des *procédures* dans une unité, il faut procéder de la manière suivante :

1. Placer la fonction dans la partie **`implementation`** ;
2. Recopier la première ligne de cette fonction dans la partie **`interface`**.

`Attention.` Si l'on décide, pour une raison ou pour une autre, de changer la première ligne de la fonction – par exemple pour changer le nombre de paramètres – il ne faut pas oublier de la changer *dans les deux parties*. Lorsqu'une erreur se produit lors de la compilation d'une unité, il faut tout d'abord vérifier qu'il y a bien adéquation entre les deux parties.

A titre d'exemple, créons une unité `Test` regroupant :

- deux variables de type *Integer*, l'une visible et l'autre cachée ;
- une fonction renvoyant un nombre entier ;
- une procédure.

On obtiendrait alors le fichier `Test.pas` :

```
Unit Test;
interface
        var
            var_visible : integer;
        procedure Proc({paramètres});
        function Func({paramètres}):integer;

implementation
        var
            var_cachee : integer;

        procedure Proc({paramètres});
            begin
              {instructions}
            end;

        function Func({paramètres}):integer;
            begin
              {instructions}
            end;

begin

end.
```

Dans notre exemple, les variables `var_visible` et `var_cachee` ne sont pas *initialisées*, ce qui signifie que leur contenu est imprévisible. Il est ainsi conseillé d'initialiser les variables d'une unité ; cela se fait dans le bloc **`begin`** – **`end`** :

```
begin
      var_visible := 0;
      var_cachee := 0;
end.
```

Ainsi, une unité permet de rassembler :

- des variables, types, constantes – certaines cachées du programme ;
- des fonctions, procédures ;
- des instructions d'initialisation des variables.

Dans le programme principal, on pourra accéder à une ou plusieurs unités. Pour ce faire, il est nécessaire d'utiliser l'instruction **uses** qui se place juste après la ligne **Program**.
Par exemple, si l'on souhaite utiliser les unités `Unite1` et `Unite2` :

```
Program Exemple;
uses Unite1, Unite2;

  {blocs type et var}
  {fonctions}
begin
  {instructions}
end.
```

1.4 Problèmes de visibilité

Le découpage des unités *en deux parties* va nous faire rencontrer de nouveaux problèmes au niveau de la déclaration des variables.
En effet, jusqu'à maintenant, nous avons toujours déclaré les types *avant* les variables – rappelons en effet qu'un bloc **type** se trouve avant le bloc **var**.

Dans le cas d'une unité, et bien que le bloc **type** se trouve avant le bloc **var**, il est possible de déclarer une variable d'un type *avant le type en lui-même* ; comme dans l'exemple suivant :

```
interface
      var
        var_visible : t_tableau;

implementation
      type
        t_tableau : array[1..24] of integer;

      var
        var_cachee : t_tableau;
```

Il se produira une erreur de compilation, car lorsque l'ordinateur arrive au niveau de la déclaration de `var_visible` de type `t_tableau`, *il n'a pas encore vu la déclaration du type* `t_tableau`. Il n'y aura en revanche pas de problème `pour var_cachee`.

Dans ce cas, il n'y a pas d'autre solution que de placer la déclaration du type `t_tableau` dans la partie **interface**, de manière à ce que l'ordinateur puisse lire cette déclaration avant la déclaration de variables de type `t_tableau`.

Afin d'éviter tout problème supplémentaire, on placera toujours les blocs dans l'ordre suivant :

1. bloc **const** ;
2. bloc **type** ;
3. bloc **var** ;
4. fonctions et procédures.

2■ Manipulation de variables *String*

2.1 Le type *String*

Le type *String* est en réalité un type très complexe, car on peut effectuer de nombreuses opérations avec, comme :

- la *concaténation* de deux chaînes de caractères, c'est-à-dire la mise bout à bout de deux chaînes de caractères ;
- l'*extraction* d'un caractère donné d'une chaîne de caractères ;
- connaître la *longueur* d'une chaîne de caractères.

2.1.1 a) Concaténation

Concrètement, il s'agit de mettre bout à bout deux chaînes de caractères, à l'aide l'opérateur + :

Ainsi, si nous avons défini trois variables de type String `var1`, `var2` et `var3` ; il est possible d'utiliser la syntaxe suivante :

```
var1 := var2 + var3;
var1 := 'Bonjour ' + var2;
```

2.1.2 b) Extraction d'un caractère

Pour extraire un caractère précis d'une chaîne de caractères, il suffit d'utiliser la syntaxe :

```
{variable de type String}[{numéro du caractère}]
```

```
Program Exemple;
var
  var1, var2 : string;

begin
      var1 := 'Bonjour.';
      var2 := var1[3];        // var2 contient 'n'
end.
```

A la fin du programme, `var2` contient la chaîne de caractères n – composée d'un seul caractère.

2.1.3 c) Longueur d'une chaîne de caractères

La longueur d'une chaîne de caractères contenue dans une variable de type String peut être obtenue facilement grâce à l'instruction `Length({variable})` qui renvoie un entier correspondant au nombre de caractères.

2.2 Le code ASCII

Cette section traite d'une notion qui n'est pas à proprement parler de la programmation – il s'agit d'informatique pure – mais qui est présentée à titre de culture générale.

Il s'agit de comprendre comment l'ordinateur voit une chaîne de caractères. En effet, nous avons vu qu'un ordinateur ne peut lire qu'une séquence de chiffres, comment une chaîne de caractères peut-elle être représentée ? Il est nécessaire d'établir une *correspondance entre nombres et caractères* : c'est ce que l'on appelle le *code ASCII*.

Ainsi, à chaque caractère correspond un chiffre. Nous donnerons ci-dessous un court extrait de la *table ASCII* :

47	/
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
60	<
61	=
62	>
63	?
64	@
65	A
66	B
67	C
68	D
69	E
70	F
71	G

Seul un petit extrait est donné, et ici *tous les nombres correspondent à des caractères*. Cependant, on retiendra que certains nombres ASCII correspondent à des caractères *qui ne sont pas affichables à l'écran* – par exemple : le saut de ligne, le "supprimer" que l'on peut entrer au clavier mais qui ne sont pas directement affichables à l'écran. On dit que ces caractères sont des *caractères non affichables*.

Il peut parfois être utile de convertir un *caractère* en *nombre ASCII*, ou l'inverse.
En effet, on peut constater que, dans la table ASCII, les lettres respectent l'ordre de l'alphabet, et les chiffres – caractères – sont dans l'ordre croissant.

Une fois un caractère converti en nombre ASCII, il est facilement possible :

- de faire des *comparaisons* entre caractères – par exemple pour déterminer laquelle de deux lettres est la première dans l'alphabet ;
- faire des *opérations* arithmétiques – par exemple, à partir d'une lettre donnée, retrouver la lettre suivante ou précédente ;
- convertir un chiffre – caractère – en chiffre, c'est-à-dire par exemple convertir la chaîne de caractères `'12'` en nombre 12.

Il existe donc deux instructions Pascal permettant de faire ces conversions :

- `ord(`{caractère entre ' '}`)` qui renvoie un nombre ASCII ;
- `chr(`{nombre ASCII}`)` renvoyant un caractère, c'est-à-dire *une chaîne de caractères de type String*.

Par exemple, admettons que la variable `Texte` de type String contienne un chiffre (c'est-à-dire en réalité un caractère !). On peut alors obtenir le *chiffre au sens numérique* contenu dans cette variable de la manière suivante :

```
ord(Texte) - ord('0')
```

Si la variable `Texte` contient `'4'`, `ord(Texte)` renverra la valeur 52. En faisant la soustraction de 52 par 48 – qui correspond à `ord('0')` – on retrouve bien la *valeur* 4.

CHAPITRE 6

Conseils de méthode

1■ Méthode de réalisation d'un programme

La connaissance des outils et des mécanismes du langage Pascal est nécessaire mais pas suffisante pour programmer efficacement et rapidement.

Lorsque l'on conçoit un programme, il est préférable de suivre une méthode afin de ne pas perdre un temps précieux à essayer de nombreux programmes avant d'arriver à quelque chose de correct.

La réalisation efficace d'un programme peut être décomposée en quatre étapes :

1. savoir précisément ce que l'on veut faire ;
2. découper le travail en petites tâches ;
3. étudier *séparément*, pour chaque tâche, comment on va s'y prendre et réaliser la programmation des petites tâches ;
4. réaliser le programme principal qui va utiliser les petites tâches.

1.

Dans un premier temps, il faut savoir ce que l'on veut. Pour l'instant, il n'est pas question de savoir comment on va le faire, mais *il faut déterminer précisément* :

- ce que l'on va entrer comme données au programme ;
- ce qu'il va nous fournir comme résultat.

Il s'agit d'une étape *très importante* – voire fondamentale – de la conception d'un programme. Elle doit être réalisée *complètement et avec précision*, car il n'est raisonnablement pas possible de créer un programme si l'on ne sait pas précisément ce qu'il doit faire.

Exemple. On souhaite réaliser une calculatrice console ; c'est-à-dire que l'on va entrer une opération – par exemple `12 + 5 * 6` – et le programme doit fournir le résultat. Il s'agit d'un programme moins simple qu'il n'y parait !

Il faut d'abord déterminer précisément ce que l'on va pouvoir entrer, et ce que l'on ne pourra pas entrer. A ce stade, on sait que l'on va entrer *une chaîne de caractères* : le calcul à effectuer.

La conception du programme étant libre, on peut décider pour des raisons de simplicité :

- le programme ne gérera pas les parenthèses ;
- le programme ne gérera pas la priorité des opérations, c'est-à-dire qu'il effectuera le calcul de gauche à droite ;
- qu'il ne gérera que les opérateurs +, -, * et /.

2.

Dans un deuxième temps, il s'agit de savoir *comment* faire.
Pour cela, on va découper le travail à effectuer :

- d'une part par *grands thèmes* – demander quelque chose à l'utilisateur, enregistrer un fichier, etc. ;
- d'autre part, *pour chaque thème, en petites tâches* qui vont constituer des fonctions ou procédures.

Pour déterminer les grands thèmes, il est souvent plus facile de se mettre à la place de l'ordinateur et de se poser la question "*Comment peut-on faire pour résoudre ce problème par étapes ?*".

Une fois que l'on a déterminé les thèmes, le découpage en petites tâches doit se faire de manière à ce qu'un ordinateur puisse les réaliser ; c'est-à-dire qu'elles doivent être très basiques et systématiques. A ce stade, il faut garder un regard critique et ne pas réaliser un découpage sauvage de tout le programme ; il est évident que des petits programmes nécessiteront peu – ou pas – de découpage.

Exemple. Dans le cadre de notre calculatrice, essayons de voir comment un humain ferait pour résoudre le problème par étapes.
Le calcul se fait de gauche à droite. Admettons que l'on doive effectuer le calcul 12 + 2 * 6. Il faudrait d'abord effectuer 12 + 2, et remplacer cette partie par 14 ; on obtiendrait alors le calcul 14 * 6, qu'il faut à nouveau effectuer de la même manière. On s'arrête lorsqu'il ne reste plus d'opérateurs, c'est-à-dire lorsque l'on trouve 84.

Nous avons procédé ici par intuition, *rien ne garantit qu'il n'existe pas de méthode plus rapide ou plus adaptée pour réaliser le calcul*. Cependant, il n'est pas question à notre niveau de se poser des questions d'*optimisation* ; nous retiendrons ainsi ce mode de calcul.

Nous avons à présent trouvé les grands thèmes ou étapes :

- trouver le premier petit calcul à effectuer, celui le plus à gauche ;
- l'effectuer ;
- remplacer ce petit calcul par son résultat dans le calcul à effectuer ; puis recommencer jusqu'à ce qu'il ne reste plus d'opérateur dans le calcul.

Il s'agit alors de découper *chaque thème en tâches*. Généralement, il est souhaitable que chaque tâche corresponde à une fonction ou à une procédure.

En se mettant à la place de l'ordinateur, on peut de la même manière déterminer des tâches très basiques et systématiques qu'un ordinateur peut effectuer.

Dans le cas de notre calculatrice, on aurait par exemple pour la partie *Calcul* :

- extraire les deux nombres de chaque côté de l'opérateur *sans prendre en compte les espaces*, ce qui va nécessiter une autre sous-étape :
  - ✖ enlever les espaces à gauche et/ou à droite d'un nombre ;
- effectuer ce petit calcul, qui diffère selon l'opérateur.

3.

Ensuite, il faut passer à la *réalisation concrète* des fonctions associées aux différentes tâches que l'on a précédemment déterminé.

Cette étape ne peut être réalisée correctement que si l'étape précédente a été correctement effectuée. En effet, il devient nécessaire de mettre les mains dans le cambouis en réfléchissant aux *paramètres*, *renvois* et *variables* que les fonctions vont utiliser.

Il s'agit souvent de l'opération la plus difficile à réaliser ; elle est plus facile à réaliser si l'on a déjà réalisé de nombreux programmes. Il existe différentes approches, l'intuition fonctionne très bien avec de l'expérience ; mais à notre niveau nous préfèrerons une approche plus systématique qui donnera toujours de bons résultats, au prix parfois d'une certaine lourdeur dans le programme.

Pour chaque tâche à effectuer, il faut déterminer :

- de quelles données la fonction a besoin, c'est-à-dire à partir de quoi elle va pouvoir produire son résultat : ce seront les *paramètres* ;
- ce qu'elle a besoin de renvoyer.

A partir de là, il faut essayer de réaliser la fonction en elle-même en essayant de comprendre comment un humain ferait pour résoudre le problème.
Le formulaire – situé à la fin de ce mini-cours – sera alors très utile pour déterminer les outils dont l'on dispose ; *on retiendra simplement que les boucles sont souvent utilisées* et il faut toujours se poser la question de savoir si une boucle ne serait pas intéressante dans le problème considéré.

4. La dernière étape consiste à réaliser le programme en lui-même. A ce stade, il est conseillé de réaliser un petit récapitulatif des fonctions utilisées ; afin de ne pas être systématiquement obligé de revenir en arrière dans le programme pour voir quels paramètres prend telle fonction.

Cette étape est souvent facile à réaliser puisque le plus difficile a été fait ; il suffit d'appeler les fonctions dans le bon ordre et de déclarer les variables globales nécessaires.

2■ Et pour quelques dollars de plus…

Soyons réalistes : rares sont les programmes qui fonctionnent du premier coup. C'est pour cette raison qu'il ne faut pas attendre d'avoir terminé le programme pour essayer de compiler, car on se retrouverait à coup sûr avec quelques dizaines d'erreurs de compilation, erreurs qui corrigées à la va-vite rendraient le programme plus difficile encore à comprendre.

Ainsi, il est conseillé, au moment de débuter la réalisation d'un programme, d'écrire la structure de base présentée au *chapitre 1* ; afin que le programme soit fonctionnel et puisse être compilé. A chaque fois que l'on a réalisé une fonction, il est souhaitable de vérifier si le programme peut être compilé correctement.

Voici, en vrac, quelques conseils de méthode :

- *soyez clair et rigoureux* lors de la réalisation. Un programme dont l'on n'a pas compris le fonctionnement exact *ne fonctionnera pas* – ou alors par un hasard exceptionnel laissant présager le pire ;
- *utilisez la touche Tabulation* pour décaler vers la droite les *blocs* entre **`begin`** et **`end`**, comme cela est systématiquement fait dans ce mini-cours. On peut ainsi facilement voir la fin d'un *bloc*, ce qui peut être utile notamment avec les boucles ;
- *gardez un regard critique* sur les programmes que vous aurez l'occasion de rencontrer. Un programme avec des dizaines de constantes, de types définis à tors et à travers n'est pas plus efficace ni plus rapide : il est simplement plus difficile à lire et à comprendre ;
- *utilisez autant que possible* les conventions des préfixes suivis dans ce mini-cours, dont le simple emploi permet d'éviter bien des erreurs :
  - `m_` pour les variables *locales* et les *paramètres* ;
  - `p_` pour les *tuyaux* ;
  - `t_` pour les *types*.

Enfin, soyez conscient que comme la voile, les échecs ou la thermodynamique, la programmation ne s'apprend pas du jour au lendemain. Seule la réalisation de plusieurs programmes permettra d'être plus à l'aise et plus efficace, voir de faire les choses "à l'intuition".

Formulaire

Variables

Integer	Entiers relatifs
Real	Nombres réels
Boolean	**true** ou **false**
String	Chaîne de caractères

Affichage d'un *Real* avec *WriteLn* :

```
WriteLn(i:0:5);
```

signifie *5 chiffres après la virgule*

Tableaux :

- déclaration par `{nom} : array[{1er}..{dernier}] of {type};`
- accès à une *case* par `{nom}[{case}]`

Types

Intervalle :
```
{nom} = {1er}..{dernier};
```

Enumération :
```
{nom} = ({nom pour 0}, {nom pour 1}, {...});
```

Ensemble :
```
{nom} = set of {type des objets};
```

Enregistrement :
```
{nom} = record
{variables} : {type};
{...}
end;
```

Tests *conditionnels*

If Then Else :
```
if {condition} then
  {bloc entre begin et end}
else
  {bloc entre begin et end;}
```

pas de ; avant **else**
else est facultatif

Case of :
```
case {variable à tester} of
1 : {bloc}
2 : {bloc}
{...}
end;
```

Boucles

For : *repète n fois un bloc d'instructions*
```
for {variable-compteur} := {point de départ} to {arrivée} do
  {bloc entre begin et end;}
```

While : *répète un bloc d'instructions tant qu'une condition est vraie*
```
while {condition} do
  {bloc entre begin et end;}
```

Until : *répète un bloc d'instructions jusqu'à ce qu'une condition soit vraie*
```
repeat
  {bloc sans begin et end;}
until {condition};
```

`{condition}` doit pouvoir changer pour éviter une boucle infinie

Structures

```
Program {nom};
  uses  {unités}
  const {constantes}
  type  {types}
  var   {variables}

  {fonctions / procédures}

  begin
       {programme}
  end.
```

```
procedure {nom}({paramètres séparés par ;});
  var
    {variables locales}
  begin

  end;


function {nom}({paramètres séparés par ;}):{type};
  var
    {variables locales}
  begin

  end;
```

Renvoi *direct*

```
unit {nom};
  interface
    {partie visible}
  implementation
    {partie cachée}
  begin
    {initialisation des variables}
  end.
```

Renvoi *direct* : utilisation d'une *fonction* ; renvoi d'une seule donnée.

Renvoi *indirect* : utilisation avec une *fonction* ou une *procédure*, syntaxe du type :

```
procedure {nom}(var {tuyau}:{type});
```

Fichiers

Fichiers texte : structurés en lignes

```
Program Fichier;
var
  F : text;

begin
  {$I-}
  Assign(F, 'fichier.txt');
  {ouverture de F}
  {$I+}
  if IOResult <> 0 then
    WriteLn('Erreur.')
  else
    {manipulation de F}
    Close(F);
end.
```

Ouvrir en *lecture*	`Reset({variable});`
Créer un nouveau fichier en *écriture*	`Rewrite({variable});`
Ouvrir un fichier existant en *écriture*	`Append({variable});`

Lire une ligne	`ReadLn({variable-fichier}, {variable String});`
Écrire une ligne	`WriteLn({variable-fichier}, {variable String});`

Fichiers typés : structurés en cases

```
Program Fichier;
var
  F : file of {type};

begin
  {$I-}
  Assign(F, 'fichier.txt');
  {ouverture de F}
  {$I+}
  if IOResult <> 0 then
    WriteLn('Erreur.')
  else
    {manipulation de F}
    Close(F);
end.
```

Ouvrir un fichier	`Reset({variable-fichier});`
Créer un nouveau fichier	`Rewrite({variable-fichier});`

Lire une case	`Read({variable-fichier}, {variable});`
Ecrire une case	`Write({variable-fichier}, {variable});`
Changer la position	`Seek({variable-fichier}, {nouvelle position});`
Connaître la taille	`FileSize({variable-fichier});`
Connaître la position	`FilePos({variable-fichier});`