Cours/TD n°3 : les boucles

Où on se rendra compte qu'il est normal de rien comprendre…

Pour l'instant, on a vu beaucoup de choses. Les variables, les `Si Alors Sinon`, les tests avec les `ET`, les `OU` et les combinaisons de tout ça. C'est déjà un bon début, mais vous savez très bien que ça ne suffit pas pour écrire un programme complet : il manque les boucles.

Les boucles, c'est généralement le point **douloureux** de l'apprenti programmeur. C'est là que ça coince, car autant il est assez facile de comprendre comment fonctionnent les boucles, autant il est souvent long d'acquérir les **réflexes** qui permettent de les élaborer judicieusement pour traiter un problème donné.

On peut dire en fait que les boucles constituent la seule vraie structure logique **caractéristique de la programmation**. Si vous avez utilisé un tableur comme Excel, par exemple, vous avez sans doute pu manier des choses équivalentes aux variables (les cellules, les formules) et aux tests (le `SI`…). Mais les boucles, ça, ça n'a aucun **équivalent**. Cela n'existe que dans les langages de programmation proprement dits.

Donc pour la majorité d'entre vous, c'est quelque chose de **totalement neuf**, alors, à vos futures - *et inévitables* - difficultés sur le sujet, il y a trois remèdes : de la patience, de la rigueur, et encore de la patience!

A quoi ça sert ?

Prenons le cas d'une saisie au clavier, où par exemple, le programme pose une question à laquelle l'utilisateur doit répondre par O (Oui) ou N (Non). Mais tôt ou tard, l'utilisateur, facétieux ou maladroit, risque de taper **autre chose** que la réponse attendue. Dès lors, le programme peut planter soit par une erreur **d'exécution** (parce que le type de réponse ne correspond pas au type de la variable attendu) soit par une erreur **fonctionnelle** (il se déroule normalement jusqu'au bout, mais en produisant des résultats fantaisistes).

Alors, dans tout programme un tant soit peu sérieux, on met en place ce qu'on appelle un **contrôle de saisie**, afin de vérifier que les donnés entrées au clavier correspondent bien à celles attendues par l'algorithme. On pourrait essayer avec un `SI`. Voyons voir ce que ça donne :

```
Var Rep : Caractère
Début
Afficher "Voulez vous un café ? (O/N)"
Saisir Rep
Si Rep ≠ "O" ET Rep ≠ "N" Alors
  Afficher "Saisie erronée. Recommencez"
  Saisir Rep
FinSi
Fin
```

C'est impeccable. Du moins tant que l'utilisateur a le bon goût de ne se tromper **qu'une seule fois**, et d'entrer une valeur correcte à la deuxième demande. Si l'on veut également bétonner en cas de

deuxième erreur, il faudrait rajouter un SI. Et ainsi de suite, on peut rajouter des centaines de SI, mais on n'en sortira pas, il y aura toujours moyen qu'un acharné flanque le programme par terre.

La solution consistant à aligner des SI en pagaille est donc une impasse. La seule issue est donc de flanquer une structure de boucle, qui se présente ainsi :

```
TantQue booléen faire
  …
  Instructions
  …
FinTantQue
```

Le principe est simple : le programme arrive sur la ligne du TantQue. Il examine alors la valeur du booléen (qui, je le rappelle, peut être une **variable booléenne** ou, plus fréquemment, **une condition**). Si cette valeur est VRAI, le programme exécute les instructions qui suivent, jusqu'à ce qu'il rencontre la ligne FinTantQue. Il retourne ensuite sur la ligne du TantQue, procède au même examen, et ainsi de suite. Le manège enchanté ne s'arrête que lorsque le booléen prend la valeur FAUX.

Mise en pratique :

Donner le nombre d'exécution de la boucle des programmes suivants :

```
X<-3
V<-12
TantQue X>V faire
 X<-2*X
FinTantQue
```

```
X<-3
V<-12
TantQue X<V faire
 X<-2*X
FinTantQue
```

```
X<-3
V<-2
TantQue X<10 faire
 X<-X+V
FinTantQue
```

```
X<-3
V<-12
TantQue X<5 ET V>5 faire
 X<- V - 10
 V<- V + X - 6
FinTantQue
```

```
X<-1
V<-1
TantQue X<5 ET V<50 faire
 X<- X+1
 V<- V * X
FinTantQue
```

```
X<-1
V<-1
Rester<-FAUX
TantQue Rester faire
 Rester <- (X + 2) > V
 V<- V + 2
 X<- X + 1
FinTantQue
```

Où on trouve la solution.

Reprenons notre problème de contrôle de saisie. Une première approximation de la solution consiste à écrire :

```
Var Rep : Caractère
Début
Ecrire "Voulez vous un café ? (O/N)" Faire
TantQue Rep <> "O" et Rep <> "N"
  Lire Rep
FinTantQue
Fin
```

Là, on a le squelette de l'algorithme correct. Mais de même qu'un squelette ne suffit pas pour avoir un être vivant viable, il va nous falloir ajouter quelques muscles et organes sur cet algorithme pour qu'il fonctionne correctement.

Son principal défaut est de provoquer une **erreur** à chaque exécution. En effet, l'expression booléenne qui figure après le `TantQue` interroge la valeur de la variable `Rep`. Malheureusement, cette variable, si elle a été déclarée, n'a pas été **affectée** avant l'entrée dans la boucle. On teste donc une variable qui n'a pas de valeur, ce qui provoque une erreur et l'arrêt immédiat de l'exécution. Pour éviter ceci, on n'a pas le choix : il faut que la variable `Rep` ait déjà été affectée avant qu'on en arrive au premier tour de boucle. Pour cela, on peut faire une première lecture de `Rep` avant la boucle. Dans ce cas, celle-ci ne servira qu'en cas de mauvaise saisie lors de cette première lecture. L'algorithme devient alors :

```
Var Rep : Caractère
Début
Ecrire "Voulez vous un café ? (O/N)"
Lire Rep
TantQue Rep <> "O" et Rep <> "N" Faire
  Lire Rep
FinTantQue
Fin
```

Une autre possibilité, fréquemment employée, consiste à ne pas lire, mais à affecter arbitrairement la variable avant la boucle. Arbitrairement ? Pas tout à fait, puisque cette affectation doit avoir pour résultat de provoquer l'entrée **obligatoire** dans la boucle. L'affectation doit donc faire en sorte que le booléen soit mis à `VRAI` pour déclencher le premier tour de la boucle. Dans notre exemple, on peut donc affecter `Rep` avec n'importe quelle valeur, hormis « O » et « N » : car dans ce cas, l'exécution sauterait la boucle, et `Rep` ne serait pas du tout lue au clavier.

Pour terminer, remarquons que nous pourrions peaufiner nos solutions en ajoutant des affichages de libellés qui font encore un peu défaut. Ainsi, si l'on est un programmeur zélé, la première solution (celle qui inclut deux lectures de Rep, une en dehors de la boucle, l'autre à l'intérieur) pourrait devenir :

```
Var Rep : Caractère
Début
Ecrire "Voulez vous un café ? (O/N)"
Lire Rep
TantQue Rep <> "O" et Rep <> "N" Faire
  Ecrire "Vous devez répondre par O ou N. Recommencez"
  Lire Rep
FinTantQue
Ecrire "Saisie acceptée"
Fin
```

Remarque

Le danger lorsque l'on fait des boucles, c'est de mal les concevoir. C'est notamment le cas lorsque l'on créé une boucle dont la condition sera toujours **fausse**. Le programme ne rentre alors **jamais** dans la superbe boucle sur laquelle vous avez tant sué !

Mais la faute symétrique est au moins aussi désopilante.

Elle consiste à écrire une boucle dans laquelle le booléen **ne devient jamais** FAUX. L'ordinateur tourne alors dans la boucle comme un dératé et n'en sort plus. Seule solution, quitter le programme avec un démonte-pneu ou un bâton de dynamite. La **boucle infinie** est une des hantises les plus redoutées des programmeurs.

Mise en pratique :

Faites les algorithmes suivants :

1. Ecrire un algorithme qui demande à l'utilisateur un nombre jusqu'à ce que ce nombre soit compris entre 1 et 3.
2. Ecrire un algorithme qui permette de calculer X à la puissance Y. Pour rappel, X^2=X*X, $X^{3=}$X*X*X, X^4=X*X*X*X …
3. Ecrire un algorithme qui reproduit le fonctionnement d'une caisse enregistreuse. C'est-à-dire que l'ordinateur va demander à l'utilisateur de saisir le montant des articles tant que le prix est différent de zéro, puis qui affiche la somme du montant de tous les articles.

Boucler en comptant, ou compter en bouclant…

Imaginons qu'on nous demande d'écrire un algorithme qui demande à l'utilisateur de saisir un nombre et qui affiche les 10 nombres qui le suivent. La première question, **toujours la même**, est de savoir combien de variables vont être nécessaires. Une première variable pour le nombre que l'utilisateur va saisir, et une seconde **pour la boucle** que nous allons utiliser. La seconde question est de décider quand doit **s'arrêter** la boucle. La solution la plus simple est de décider que la boucle s'arrêtera quand la variable de la boucle sera supérieure à la valeur que l'utilisateur a rentrée + 10. Ainsi, nous aurons un algorithme du style :

```
Var val,boucle : Entier
Début
Ecrire "Entrez un nombre"
Lire val
boucle<-val
TantQue boucle < val+10 Faire
  Afficher boucle
  boucle <- boucle + 1
FinTantQue
Fin
```

Vous avez remarqué que la boucle était utilisée pour augmenter la valeur d'une variable. Cette utilisation des boucles est très fréquente, et dans ce cas, il arrive très souvent qu'on ait besoin d'effectuer un nombre **déterminé** de passages. Or, a priori, notre structure `TantQue` ne sait pas à l'avance combien de tours de boucle elle va effectuer (puisque le nombre de tours dépend de la valeur d'un booléen).

C'est pourquoi une autre structure de boucle est à notre disposition :

```
Var val,boucle : Entier
Début
Ecrire "Entrez un nombre"
Lire val
Pour boucle de val à val+10 Faire
  Afficher boucle
FinPour
Fin
```

Insistons : la structure « Pour … FinPour » n'est pas du tout indispensable ; on pourrait fort bien programmer toutes les situations de boucle uniquement avec un « Tant Que ». Le seul intérêt du « Pour » est d'épargner un peu de **fatigue** au programmeur, en lui évitant de gérer lui-même la progression de la variable qui lui sert de compteur (on parle **d'incrémentation**, encore un mot qui fera forte impression sur votre entourage).

Dit d'une autre manière, la structure « Pour … Suivant » est un cas particulier de `TantQue` : celui où le programmeur peut dénombrer **à l'avance** le nombre de tours de boucles nécessaires.

Il faut noter que dans une structure « Pour … Suivant », la progression du compteur est laissée à votre libre disposition. Dans la plupart des cas, on a besoin d'une variable qui augmente de 1 à

chaque tour de boucle. On ne précise alors rien à l'instruction « Pour » : celle-ci, par défaut, comprend qu'il va falloir procéder à cette incrémentation de 1 à chaque passage, en commençant par la première valeur et en terminant par la deuxième.

Mais si vous souhaitez une progression plus spéciale, de 2 en 2, ou de 3 en 3, ou en arrière, de -1 en -1, ou de -10 en -10, ce n'est pas un problème : il suffira de le préciser à votre instruction « Pour » en lui rajoutant le mot « **Pas de** » et la valeur de ce pas (Le « pas » dont nous parlons, c'est le « pas » du marcheur, « step » en anglais).

Naturellement, quand on stipule un pas négatif dans une boucle, la valeur initiale du compteur doit être **supérieure** à sa valeur finale si l'on veut que la boucle tourne ! Dans le cas contraire, on aura simplement écrit une boucle dans laquelle le programme ne rentrera jamais.

Nous pouvons donc maintenant donner la formulation générale d'une structure « Pour ». Sa syntaxe générale est :

```
Pour boucle de init à final Pas de ValeurDuPas Faire
  Instructions
FinPour
```

Les structures `TantQue` sont employées dans les situations où l'on doit procéder à un traitement systématique sur les éléments d'un ensemble dont on ne connaît pas d'avance la quantité, comme par exemple :

- le contrôle d'une saisie
- la gestion des tours d'un jeu (tant que la partie n'est pas finie, on recommence)
- la saisie d'un nombre inconnu de valeurs

Les structures `Pour` sont employées dans les situations où l'on doit procéder à un traitement systématique sur les éléments d'un ensemble dont le programmeur connaît d'avance la quantité.

Remarque.

Dans une structure `Pour`, il ne faut surtout pas essayer de **modifier** la valeur du compteur. En effet, la boucle pour change automatiquement sa valeur à chaque passage, et la modifier manuellement entraine un **plantage** quasi inévitable.

Mise en pratique :

- Ecrire un algorithme qui demande un nombre de départ, et qui calcule sa factorielle.
  NB : la factorielle de 8, notée 8 !, vaut 1 x 2 x 3 x 4 x 5 x 6 x 7 x 8
- Ecrire un algorithme qui demande successivement 20 nombres à l'utilisateur, et qui lui dise ensuite quel était le plus grand parmi ces 20 nombres.
- Modifiez l'algorithme précédent pour que le programme affiche de surcroît en quelle position avait été saisie ce nombre : « C'était le nombre n° 2 »
- Réécrire l'algorithme précédent, mais cette fois-ci on ne connaît pas d'avance combien l'utilisateur souhaite saisir de nombres. La saisie des nombres s'arrête lorsque l'utilisateur entre un zéro.