

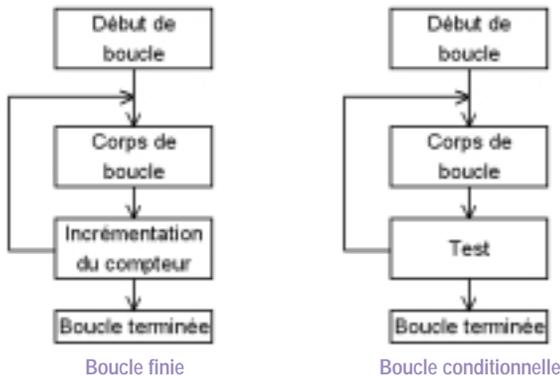
Rpl : leçon n°3 *Hp 49*

Après la gestion de la pile, le fonctionnement des variables et la structure des tests, voici le plus gros morceau à digérer : les boucles.

C'est la dernière étape avant de s'attaquer à des applications concrètes. Cette partie, sans doute la plus importante, est utilisée quasi constamment dans bon nombre d'applications (essentiellement pour des calculs itératifs). Pour ceux qui connaissent la programmation sur PC, les boucles sont gérées comme dans le langage C ou le Pascal, mais à la sauce polonaise bien évidemment.

• brèves généralités •

Une boucle va réaliser la même action soit un certain nombre de fois, soit en fonction d'une condition. Nous avons donc deux types de boucles : les boucles finies et les boucles conditionnelles ou infinies.



Pour la boucle finie on en sort en fonction de l'état du compteur de la boucle, et pour la boucle conditionnelle on en sort en fonction du résultat du test. Mais voyons tout ça en détail et séparément...

• boucles finies •

Une boucle finie utilise un compteur de boucle qui commence avec une valeur de départ et qui s'achève avec une valeur finale, le compteur s'incrémentant d'un pas variable (souvent égal à 1) à chaque boucle. Plusieurs cas de figures se présentent pour créer des boucles finies.

Commençons par la plus simple avec cette structure :

```
DEP ARR START (corps de boucle) NEXT
```

DEP est la valeur de départ et ARR la valeur d'arrivée. L'instruction START déclare le début de la boucle et NEXT en indique la fin avec un incrément de 1. Le corps de boucle représente un bloc d'instructions et de commandes qui sera exécuté autant de fois qu'il y a de boucles à faire. Voyons un petit exemple qui calcule la multiplication de deux nombres entiers (différents de zéro) par l'addition :

```
< -> A B < 0 1 A START B + NEXT > >
```

On utilise ce programme en ayant préalablement posé les deux entiers sur les niveaux 1 et 2 de la pile. Ici, la boucle va de 1 à A, et à chaque tour de boucle on ajoute la valeur B afin de réaliser une addition, la valeur de départ étant 0 pour l'initialisation. Prenons par exemple A=5 et B=7, voici ce que va réaliser la boucle :

$$A * B = 0 + 7 + 7 + 7 + 7 + 7 = 35$$

Voici une autre structure (sans doute la plus utilisée en Rpl) qui permet d'utiliser le compteur de boucle au sein du corps de boucle :

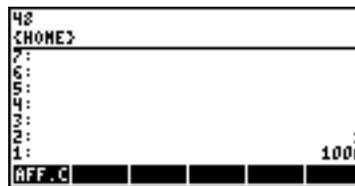
```
DEP ARR FOR COMPTEUR (corps de boucle) NEXT
```

Sur le même principe, nous avons une valeur de départ et une valeur finale représentées respectivement par DEP et ARR. Nouveauté par rapport à la structure précédente, nous pouvons connaître la valeur du compteur de boucle grâce à COMPTEUR, une variable qu'on définira comme locale à la boucle. C'est à dire qu'à l'intérieur de la boucle, nous pouvons utiliser la valeur du compteur. L'exemple le plus simple consiste à afficher la valeur du compteur sur la première ligne en haut de l'écran. Le programme devient tout aussi simple :

```
< -> A B < A B FOR C C 1 DISP 0.5 WAIT NEXT > >
```

Comme précédemment, les valeurs des bornes A et B sont à déposer sur les niveaux 1 et 2 de la pile avant de lancer le programme. Les valeurs vont ensuite s'afficher toutes les demies secondes en haut de l'écran grâce à la séquence 0.5 WAIT. On aurait pu simplifier ce programme par :

```
< FOR C C 1 DISP 0.5 WAIT NEXT >
```



La valeur de la boucle à l'instant T

Avec la dernière structure pour les boucles finies, seul le pas (ou la valeur de l'incrément) peut désormais évoluer grâce à la commande STEP :

```
DEP ARR FOR COMPTEUR (corps de boucle) PAS STEP
```

Les bornes DEP et ARR s'utilisent toujours de la même façon, ainsi que la variable COMPTEUR. Le pas (nommé ici PAS) peut véritablement évoluer : il peut être entier ou réel, positif ou négatif comme le montre cet exemple qui inverse les caractères d'une chaîne de caractères :

```
< DUP SIZE -> CHAINE TAILLE
```

```
< " " TAILLE 1 FOR I
```

```
CHAINE I I SUB + -1 STEP > >
```

Avant de lancer ce programme, il faut poser une chaîne de caractères sur le niveau 1 de la pile. Ici, comme on commence par extraire les caractères par la fin de la chaîne, le pas est négatif, d'où la séquence -1 STEP. Pour information, la commande SIZE permet de connaître la taille de la chaîne de caractères, et I I SUB permet d'extraire un caractère à la fois. Pour simplifier, on aurait pu écrire ce programme comme ceci :

```
< " " OVER SIZE 1 FOR I OVER I I SUB +  
-1 STEP SWAP DROP >
```



Team Palmtops : avant et après !

Avec les boucles finies, différents types de bornes peuvent être utilisés. Les voici classés par ordre de rapidité avec pour chacun un exemple :

- les entiers binaires :

```
#1d #50d FOR I ... NEXT
```

- les réels :

```
1. 2.5 FOR I ... .5 STEP
```

- les entiers longs :

```
1 50 FOR I ... NEXT
```

• boucles conditionnelles •

Pour être simpliste, une boucle conditionnelle va tourner infiniment jusqu'à ce que le test de sortie soit validé, c'est pourquoi on les appelle aussi boucles infinies. Deux types de structures sont proposées selon l'utilisation que l'on en fait.

La première structure est du type "tant que la condition est vraie, alors on exécute la boucle", d'où sa traduction en Rpl :

```
WHILE (condition) REPEAT (corps de boucle) END
```

La forme des conditions est identique à celle utilisée pour les tests, les opérateurs logiques sont donc eux aussi les mêmes. A titre d'exemple, réalisons un programme qui attend l'appui sur une touche :

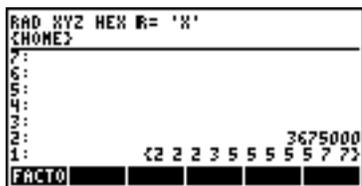
```
< CLLCD "J'attends..." 1 DISP
  WHILE KEY NOT REPEAT END DROP >
```

Le programme efface ici l'écran (commande CLLCD), affiche un message sur la première ligne et attend patiemment l'appui sur une touche. Un appui sur le clavier produit un code de la touche enfoncée, d'où le DROP qui justifie l'effacement du code placé sur le niveau 1 de la pile. La séquence KEY NOT veut simplement dire "pas d'appui sur une touche", et REPEAT END exprime l'exécution de la boucle à vide. Pour résumer, c'est une boucle d'attente...

Présentons maintenant une application pratique : essayons de trouver les diviseurs premiers d'un entier positif sans utiliser les commandes internes FACTOR ou FACTORS. Une simple boucle WHILE couplée à un test fait l'affaire :

```
< 2 -> a b
  < { } WHILE a 1 > REPEAT
    IF a b MOD 0 ==
      THEN b + a b / 'a' STO
      ELSE b 1 + 'b' STO
    END
  END >>
```

Dans ce programme, rien de vraiment neuf à part la commande STO qui permet de stocker de nouvelles valeurs dans les variables locales. L'algorithme est vraiment classique, chose que vous découvrirez par vous même... Voici néanmoins une illustration de cette application avec la décomposition de 3675000 qui devrait normalement donner $2^3 \cdot 3^3 \cdot 5^5 \cdot 7^2$ avec la commande interne :



Un entier en état de décomposition...

Le programme présenté est une solution, il est aussi possible d'imbriquer deux boucles WHILE pour aboutir exactement au même résultat :

```
< 2 -> a b
< { } WHILE a 1 > REPEAT
  WHILE a b MOD 0 ==
    REPEAT b + a b / 'a' STO END
  b 1 + 'b' STO
END >>
```

La seconde structure est du type "exécuter la boucle jusqu'à ce que la condition soit vraie", on en déduit aisément sa forme Rpl :

```
DO (corps de boucle) UNTIL (condition) END
```

L'exemple le plus simple consiste là aussi à attendre l'appui sur une touche, mais cette fois avec l'instruction DO :

```
< CLLCD "J'attends..." 1 DISP
  DO UNTIL KEY END DROP >
```

La première ligne ne change pas : effacement de l'écran et affichage du message. La commande DROP s'explique de la même manière que pour la boucle WHILE. Pour être concret, voici un petit programme qui calcule les six (bons) numéros du loto avec une boucle de type DO. Pour mémoire, le programme devra tirer six boules parmi quarante-neuf. Le programme devient alors :

```
< { } DO
  49 RAND * FLOOR 1 +
  DUP2 POS NOT
  { + } { DROP } IFTE
  UNTIL DUP SIZE 6 ==
  END SORT >
```

Le but du jeu est simple : on ajoute un nouveau nombre aléatoire compris entre 1 et 49 à une liste jusqu'à ce que la taille de cette liste soit égale à 6. Pour information, la commande RAND génère un nombre aléatoire compris entre 0 et 1 exclus, FLOOR arrondit un réel à l'entier qui lui est immédiatement inférieur, POS donne la position d'un objet dans une liste, SIZE donne la taille de la liste, et SORT trie la liste dans l'ordre croissant. Et pour changer, nous avons utilisé ici la structure de test IFTE.

Après assimilation des deux types de structures, on se rend compte que la boucle WHILE teste la condition avant d'exécuter le corps de boucle, alors qu'avec la boucle DO la condition est testée en fin de boucle. Ce qui veut dire que la boucle DO est exécutée au moins une fois avant de tester une éventuelle sortie.

• fin du cours •

C'est avec ce troisième volet que se termine l'initiation au langage Rpl. A présent, afin de prendre la machine complètement en main et programmer ce que bon vous semble (en utilisant les listes, les chaînes de caractères, les grobs, etc.), il faut absolument passer par une série d'exercices pratiques, sachant que l'essentiel étant de réussir ce qu'on désire concevoir. L'optimisation viendra avec l'expérience...

Philippe Pamart
[phpamart@nordnet.fr]