Cours 5

M.Adam – JF.Kamp – S.Letellier – F.Pouit

7 août 2016

Table des matières

1 Autres formes de boucles

1.1 Les deux autres formes de boucles

- `repeter ... jusqua`
- `pour`

1.2 La boucle répéter

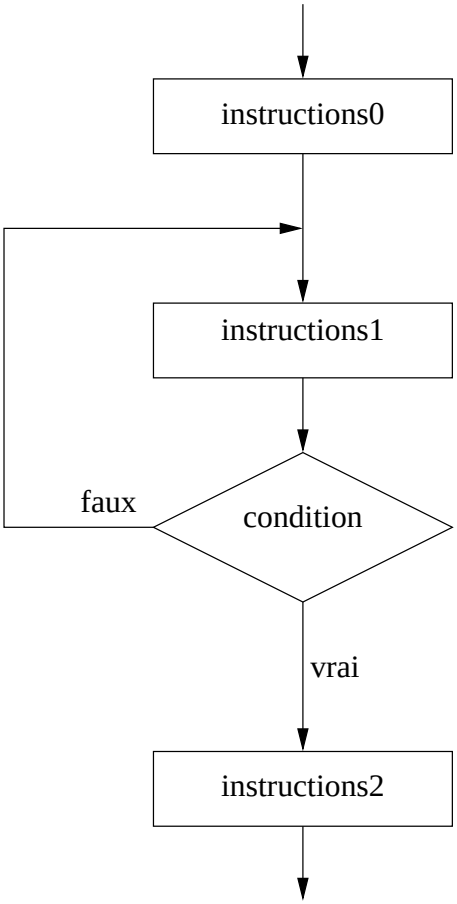
C'est une boucle qui :
- s'exécute au moins une fois,
- s'arrête sur condition d'arrêt.

1.2.1 Syntaxe

```
repeter
    instructions ;
jusqua (booleen condition)
finrepeter
```

1.2.2 Sémantique

```
instructions0;
repeter
     instructions1;
jusqua (condition)
finrepeter
instructions2;
```



1.2.3 Sémantique

```
instructions0;          | instructions0;
instructions1;          |
tantque (non condition) | repeter
    instructions1;      |     instructions1;
fintantque              | jusqua (condition)
                        | finrepeter
instructions2;          | instructions2;
```

1.2.4 Un exemple avec `tantque`

```
##
# Saisie d'un entier positif
# @author M.Adam
##
algo Tantque
principal
var
```

```
    entier val;
debut
    afficherln("Saisie d'un nombre positif");     #instructions0
    saisir("Entrez un entier positif", @val);     #instructions1
    tantque (val < 0)                             #tantque (non condition)
        saisir("Entrez un entier positif", @val); #instructions1
    fintantque
    afficherln("La valeur saisie est : "+val);    #instructions3
fin
```

1.2.5 Même exemple avec `repeter`

```
##
# Saisie d'un entier positif
# @author M.Adam
##
algo Repeter
principal
var
    entier val;
debut
    afficherln("Saisie d'un nombre positif");     #instructions0
    repeter
         saisir("Entrez un entier positif", @val);#instructions1
    jusqua (val >= 0)                              #jusqua (condition)
    finrepeter
    afficherln("La valeur saisie est : "+val);    #instructions3
fin
```

1.2.6 Utilisation

Une boucle `repeter` est à utiliser quand il faut exécuter au moins une fois la boucle avant de sortir.

1.3 La boucle pour

C'est une boucle qui :
- s'exécute un **nombre connu de tours**.

1.3.1 Syntaxe

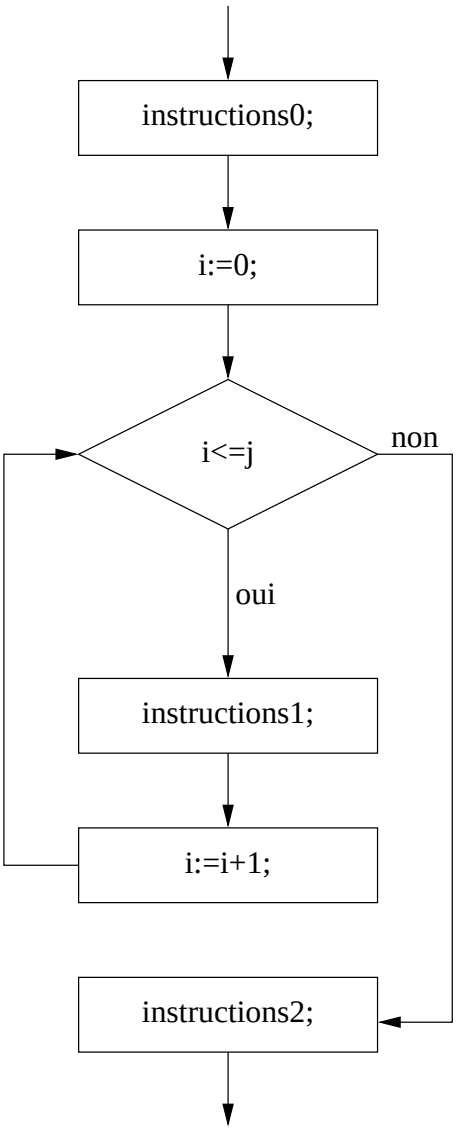
```
pour i := 0 à j pas 1
    instructions;
finpour
```

`j+1` tours de boucle.

i	`i==0`	`i==1`	...	`i==j-1`	`i==j`
tour	1	2		j	j+1

1.3.2 Sémantique

```
instructions0;
pour i := 0 à j pas 1
    instructions1;
finpour
instructions2;
```




```
instructions0;                | instructions0;
i := 0;                       |
tantque (i <= j) # i <> j+1   | pour i := 0 à j pas 1
    instructions1;            |     instructions1;
```

```
    i:= i + 1;                |
fintantque                    | finpour
instructions2;                | instructions2;
```

1.3.3 Un exemple avec tantque

```
##
# Affiche 10 nombres aléatoirement
# @author M.Adam
##
algo Tantque
principal
var
    entier i;
debut
    afficherln("Affiche 10 nombres aléatoirement"); #instructions0
    i := 0;
    tantque (i <= 10)  #i <> 11
        afficherln(i+" "+arrondi(aleatoire()*100)); #instructions1
        i := i + 1;
    fintantque
    afficherln("fin de l'algorithme");              #instructions3
fin
```

1.3.4 Même exemple avec pour

```
##
# Affiche 10 nombres aléatoirement
# @author M.Adam
##
algo Pour
principal
var
    entier i;
debut
    afficherln("Affiche 10 nombres aléatoirement"); #instructions0

    pour i := 0 à 10 pas 1
        afficherln(i+" "+arrondi(aleatoire()*100)); #instructions1
    finpour
    afficherln("fin de l'algorithme");              #instructions3
fin
```

1.3.5 Utilisation

Une boucle pour est à utiliser quand le nombre de tours de boucle est connu.

1.3.6 Limites dans l'utilisation du pour

Le programmeur se doit de :
- ne pas modifier l'indice dans la boucle, par exemple, `i := i + 1;`
- ne pas modifier la borne de fin dans la boucle, par exemple, `j := j - 1;`

1.3.7 Limites du pour dans TestAlgo

La borne supérieure doit toujours être supérieure ou égale à l'initialisation.

```
##
# Exemple qui ne fonctionne pas en TestAlgo
# La limite supérieure est inférieure à l'initialisation
# @author M.Adam
##
algo Pour
principal
var
    entier i;
debut
    afficherln("Limite de la boucle pour en TestAlgo");

    pour i := 0 à -1 pas 1
        afficherln(i+" "+arrondi(aleatoire()*100));
    finpour
    afficherln("fin de l'algorithme");
fin

TestAlgo - Interprétation engagée.
Limite de la boucle pour en TestAlgo
...
Erreur a la ligne 13 : La boucle "pour" est infinie
```

1.4 Programmer ces deux formes de boucles

La boucle `tantque` est suffisante pour programmer tous les algorithmes.

Les autres formes de boucles permettent
- parfois de simplifier l'écriture des boucles,
- souvent de simplifier leur lecture.

1.4.1 La boucle `repeter`

Deux solutions :
- adapter la méthode pour la boucle `tantque`,
- utiliser la méthode pour la boucle `tantque` et reconnaitre le schéma de traduction pour la transformer en boucle `repeter`.

1.4.2 La boucle `pour`

Deux solutions :
- comme le nombre de tours est connu, programmer dès le départ une boucle pour,
- utiliser la méthode pour la boucle `tantque` et reconnaitre le schéma de traduction pour la transformer en boucle `pour`.

2 Plus loin avec les procédures et les fonctions

2.1 Les tests unitaires

A chaque procédure `proc()`, à chaque fonction `fonc()` est associée une procédure `testProc()`, et, respectivement, `testFonc()`.

Cette procédure permet le test unitaire de la procédure ou de la fonction :
- tester tous les cas de **bon fonctionnement**,
- tester un maximum de cas de bon fonctionnement si l'exhaustivité n'est pas possible,
- ne pas traiter les cas d'erreurs car nous faisons l'hypothèse que les paramètres sont corrects,
- n'utiliser que des variables locales,
- être "bavarde".

2.1.1 Exemple de Test Unitaire

Soit la fonction `palindrome()` dont la description est donnée par :

```
##
# Teste si une chaîne est un palindrome
# @param chaîne à tester
# @return var si la chaîne est un palindrome, faux sinon
##
fonction palindrome(chaine mot) : booleen
```

Tests à effectuer

-
-
-
-
-

Code de testPalindrome()

```
##
# Teste la fonction palindrome"
##
procedure testPalindrome()
debut
    alaligne();
    afficherln("*** testPalindrome()");

    afficher("test sur un palindrome, palindrome(radar) : ");
    afficherln(palindrome("radar"));

    afficher("test sur un palindrome, palindrome(abba) : ");
    afficherln(palindrome("abba"));

    afficher("test sur un non palindrome, palindrome(rider) : ");
    afficherln(palindrome("rider"));

    afficher("test sur un non palindrome, palindrome(raor) : ");
    afficherln(palindrome("raor"));

    afficher("test sur un mot vide, palindrome( ) : ");
    afficherln(palindrome(""));

    afficher("test sur un mot d'un seul caractère, palindrome(a) : ");
    afficherln(palindrome("a"));

    alaligne();
fin
```

Exécution

```
*** testPalindrome()
test sur un palindrome, palindrome(radar) : VRAI
test sur un palindrome, palindrome(abba) : VRAI
test sur un non palindrome, palindrome(rider) : FAUX
test sur un non palindrome, palindrome(raor) : FAUX
test sur un mot vide, palindrome( ) : VRAI
test sur un mot d'un seul caractère, palindrome(a) : VRAI
```

2.1.2 Avantages

- Chaque procédure ou fonction est testée au fur et à mesure.
- Les procédures de test peuvent être codées par une autre personne.
- Les procédures de test peuvent être codées avant la procédure ou fonction.
- La recherche d'erreurs est facilitée.
- La construction des autres procédures ou fonctions qui utilisent du code déjà écrit et testé est plus solide.

- Il est possible ultérieurement de d'utiliser à nouveau une procédure de test.

2.1.3 Inconvénients

- L'écriture initiale du code est plus longue.
- Il faut avoir bien réfléchi aux tests et être le plus exhaustif possible.
- Il faut savoir remettre en cause une procédure ou une fonction déjà testée.

2.2 Problèmes complexes

Parfois un problème à résoudre est très complexe à programmer.

Divide and conquer

L'idée est de décomposer un gros problème en de plus petits problèmes plus simples à résoudre.

2.2.1 L'exemple du tri dit par sélection

Voir la vidéo :
`https://www.youtube.com/watch?v=Ns4TPTC8whw`

2.2.2 La procédure `afficheTab()`

```
##
# affiche les n premiers valeurs d'un tableau d'entier
# @param tableau à afficher
# @param nombre de valeurs à afficher
##
procedure afficheTab(par_ref tableau_de entier tab, entier n)
```

Le test de afficheTab()

```
*** testAfficheTab()
afficheTab(@tab,10) :
10 20 30 40 50 60 70 80 90 100
afficheTab(@tab,5) :
10 20 30 40 50
afficheTab(@tab,0) :
```

2.2.3 La procédure `echangeValTab()`

```
##
# echange deux valeurs dans un tableau d'entiers
# @param tableaux d'entiers
# @param premier indice pour l'échange
# @param deuxième indice pour l'échange
##
procedure echangeValTab(par_ref tableau_de entier tab, entier i, entier j)
```

```
*** testEchangeValTab()
echangeValTab(@tab, 0, 1) :
avant :
10 20 30 40 50 60 70 80 90 100
après :
20 10 30 40 50 60 70 80 90 100

echangeValTab(@tab, 9, 9) :
avant :
20 10 30 40 50 60 70 80 90 100
après :
10 20 30 40 50 60 70 80 90 100
```

2.2.4 Procédure placerMin()

l'idée est de placer au rang i la plus petite des valeurs se trouvant dans la suite du tableau.

```
##
# place au rang i la plus petite valeur de la suite du  tableau
# @param tableau d'entiers
# @param rang de départ où sera placée la valeur la plus petite
# @param nombre de valeurs dans le tableau
##
procedure placerMin(par_ref tableau_de entier tab, entier i, entier lg)
```

Toutes les valeurs des paramètres sont supposées être correctes.

Principe Parcourir le tableau à partir du rang i. Si la valeur courante est plus petit que celle à l'indice i, les deux cases sont permutées.

Corps de boucle L'indice `j` est utilisé pour le parcours du tableau.

```
si (tab[i] > tab[j]) alors
    echangeValTab(@tab, i, j);
```

```
finsi
j := j + 1;
```

Conditions de sortie

- `j >= lg`, l'indice est sortie du tableau.

Condition de continuation

- `non(j >= lg)`
- qui se réécrit `j < lg`

Initialisation

```
j := i + 1;
```

Il est inutile de comparer `i` avec `i`.

Terminaison rien

Code complet

```
##
# place au rang i la plus petite valeur de la suite du  tableau
# @param tableau d'entiers
# @param rang de départ où sera placée la valeur la plus petite
# @param nombre de valeurs dans le tableau
##
procedure placerMin(par_ref tableau_de entier tab, entier i, entier lg)
var
    entier j;
debut
    j := i + 1;
    tantque(j < lg)
        si (tab[i] < tab[j]) alors
            echangeValTab(@tab, i, j);
        finsi
        j := j + 1;
    fintantque
fin
```

Évidemment, certains auront reconnu une boucle `pour`.

2.2.5 Procédure `testPlacerMin()`

```
*** testPlacerMin()
placerMin(@tab, 0, 10) :
avant :
100 30 20 40 60 70 90 80 50 40
après :
20 100 30 40 60 70 90 80 50 40

placerMin(@tab, 0, 10) :
avant :
20 100 30 40 60 70 90 80 50 40
après :
20 100 30 40 60 70 90 80 50 40

placerMin(@tab, 10, 10) :
avant :
20 100 30 40 60 70 90 80 50 40
après :
20 100 30 40 60 70 90 80 50 40

placerMin(@tab, 5, 10) :
avant :
20 100 30 40 60 70 90 80 50 40
après :
20 100 30 40 60 40 90 80 70 50
```

2.2.6 Procédure `triSelection()`

```
##
# tri d'un tableau d'entiers selon la méthode dite de sélection
# @param tableau d'entiers à trier
# @param nombre de valeurs du tableau
##
procedure triSelection(par_ref tableau_de entier tab, entier lg)
```

Principe Le tableau est parcouru entièrement par un indice, `i`. À chaque tour, la plus petite valeur se trouvant dans le tableau à partir du rang `i` est placée en `i`.

Corps de boucle

```
placerMin(@tab, i, lg);
i := i + 1;
```

Conditions de sortie

- `i >= lg` en dehors du tableau

Condition de continuation

- `non(i >= lg)`
- qui se réécrit `i < lg`

Initialisation

```
i := 0;
```

Terminaison rien

Code complet

```
##
# tri d'un tableau d'entiers selon la méthode dite de sélection
# @param tableau d'entiers à trier
# @param nombre de valeurs du tableau
##
procedure triSelection(par_ref tableau_de entier tab, entier lg)
var
    entier i;
debut
    i:=0;
    tantque(i < lg)
        placerMin(@tab, i, lg);
        i := i + 1;
    fintantque
fin
```

Évidemment, là encore, il s'agit d'une boucle `pour`.

2.2.7 Procédure `testTriSelection()`

```
*** testTriSelection()
triSelection(@tab, 10) :
avant :
100 30 20 40 60 70 90 80 50 40
après :
20 30 40 40 50 60 70 80 90 100
```

3 Conclusion

3.1 A retenir

- Il existe d'autres formes de boucles : `pour` et `repeter`.
- La boucle `pour` sert quand le nombre de tours est connu.
- La boucle `repeter` sert quand le corps de boucle doit toujours être exécuté au moins une fois.
- Chaque procédure ou fonction doit être testée individuellement par une procédure `test..()`.
- Quand un problème est complexe, il faut le décomposer en problèmes moins complexes à programmer.
- La décomposition d'un problème en sous-problème rend la mise au point plus simple.