

CHAP4 NOTION D'ALGORITHME ET DE PROGRAMME

STRUCTURES CODITIONNELLES REPETITIVES

Une **structure répétitive** (ou **structure itérative**) répète l'exécution d'un traitement, dans un ordre précis, un nombre déterminé ou indéterminé de fois. Une structure itérative est aussi appelée boucle.

Deux cas sont cependant à envisager, selon que :

- le nombre de répétitions est connu à l'avance : c'est le cas des **boucles itératives**
- le nombre de répétitions n'est pas connu ou est variable : c'est le cas des **boucles conditionnelles**

1- Boucle Pour (For)

La structure de contrôle répétitive **Pour** (**For** en langage PASCAL) utilise un indice entier qui varie (avec un incrément = 1) d'une valeur initiale jusqu'à une valeur finale. À la fin de chaque itération, l'indice est incrémenté de 1 d'une manière automatique (implicite).

La syntaxe de la boucle pour est comme suit :

<u>pour</u> <indice> ← <vi> à <vf> <u>faire</u>	for (indice = vi; indice <= vf; indice+1)
<instruction(s)>	{
	instruction(s)
	}
<u>finPour;</u>	

<indice> : variable entière

<vi> : valeur initiale <vf> : valeur finale

d'une manière générale la structure de la boucle for en langage C est :

for (initialisation; condition; modification) { instructions; }

La boucle **pour** contient un bloc d'instructions (les instructions à répéter). Si le bloc contient une seule instruction, le begin et end sont facultatifs.

Le bloc sera répété un nombre de fois = ($\text{<vf>} - \text{<vi>} + 1$) si la valeur finale est supérieure ou égale à la valeur initiale. Le bloc sera exécuté pour $\text{<indice>} = \text{<vi>}$, pour $\text{<indice>} = \text{<vi>}+1$, pour $\text{<indice>} = \text{<vi>}+2$, ..., pour $\text{<indice>} = \text{<vf>}$.

La sortie de la boucle s'effectue lorsque le nombre souhaité d'itérations est atteint, c'est-à-dire lorsque <indice> prend la valeur <vf>.

Il ne faut jamais mettre de point-virgule après le mot clé do. (erreur logique).

Exemple Faire la somme des 10 premiers entiers naturels.

```
Algorithme utilisation_du_compteur1;  
  Var somme,i : integer;  
debut  
  Somme ← 0 ;  
  Pour i ← 1 to 9 Faire somme ← somme + i ;  
fin.
```

2- Boucle Tant-que (While)

La structure de contrôle répétitive **tant-que** (**while** en langage PASCAL) utilise une expression logique ou booléenne comme condition d'accès à la boucle : si la condition est vérifiée (elle donne un résultat vrai : TRUE) donc on entre à la boucle, sinon on la quitte.

La syntaxe de la boucle **tant-que** est comme suit :

```
tant-que <condition> faire  
debut  
<instruction(s)> finTant-que;
```

```
while <condition>  
{  
<instruction(s)>  
>;  
}
```

<condition> : expression logique qui peut être vraie ou fausse.

On exécute le bloc d'instructions tant que la condition est vraie. Une fois la condition est fausse, on arrête la boucle, et on continue l'exécution de l'instruction qui vient après **finTant-que** (après }).

Toute boucle **Pour** peut être remplacée par une boucle **tant-que**, cependant l'inverse n'est pas toujours possible.

Exemple : Faire la somme des 10 premiers entiers naturels.

```
Algorithme utilisation_compteur2 ;  
  Var i,somme :entier ;  
debut  
  I ← 1 ; Somme ← 0;  
  tant-que i < 10 faire  
  debut  
    somme ← somme + i ;  
    i ← i + 1 ;  
  fin;  
fin.
```

3- Boucle Répéter (Repeat)

La structure de contrôle répétitive **répéter** utilise une expression logique ou booléenne comme condition de sortie de la boucle : si la condition est vérifiée (elle donne un résultat vrai : TRUE) on sort de la boucle, sinon on y accède (on répète l'exécution du bloc).

La syntaxe de la boucle répéter est comme suit :

répéter	do
<instruction(s)> ;	{
jusqu'à <condition>;	instructions;
	}
	while (condition);

<condition> : expression logique qui peut être vraie ou fausse.

On exécute le bloc d'instructions jusqu'à avoir la condition correcte. Une fois la condition est vérifiée, on arrête la boucle, et on continue l'exécution de l'instruction qui vient après jusqu'à (while (condition)).

La différence entre la boucle **répéter** et la boucle **tant-que** est :

– La condition de **répéter** est toujours l'inverse de la condition **tant-que** : pour **répéter** c'est la condition de sortie de la boucle, et pour **tant-que** c'est la condition d'entrer.

– Le test de la condition est à la fin de la boucle (la fin de l'itération) pour **répéter**. Par contre, il est au début de l'itération pour la boucle **tant-que**. C'est-à-dire, dans **tant-que** on teste la condition avant d'entrer à l'itération, et dans **répéter** on fait l'itération après on teste la condition.

Exemple : Faire la somme des 10 premiers entiers naturels.

```

Algorithme utilisation_compteur3 ;
Var somme,i : entier ;
debut
    Somme ← 0 ; i ← 1 ;
    Répéter
        somme ← somme + i ; i ← i+1 ;
    jusqu'à i = 10 ;
End.

```

En langage C :

```

#include <stdio.h>

int main()
{
    int somme, i;

    somme = 0;
    i = 1;

    do
    {
        somme = somme + i;
        i = i + 1;
    }
    while (i != 10);

    return 0;
}

```

Choix de la boucle

La règle est simple :

Si le nombre d'itérations est connu a priori, alors on utilise un for.

Sinon : on utilise le repeat (quand il y a toujours au moins une itération), ou le while (quand le nombre d'itérations peut être nul).

4-Représentation en organigramme

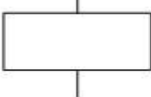
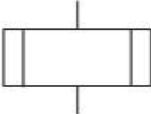
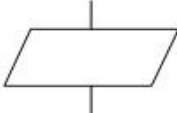
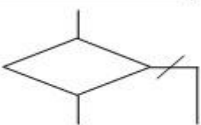
Un algorithme peut aussi être représenté de façon graphique. On parle ici d'*organigramme*. Un organigramme est une représentation normalisée de l'enchaînement des opérations et des décisions à effectuer par un programme d'ordinateur. Chaque type d'action dans l'algorithme possède une représentation dans l'organigramme.

Il est préférable d'utiliser la représentation algorithmique que la représentation par organigramme notamment lorsque le problème est complexe.

Les inconvénients qu'on peut rencontrer lors de l'utilisation des organigrammes sont :

- Quand l'organigramme est long et tient sur plus d'une page,
- problème de chevauchement des flèches,
- plus difficile à lire et à comprendre qu'un algorithme.

Les principaux symboles rencontrés dans un organigramme sont représentés dans le tableau ci-dessous :

SYMBOLE	DESIGNATION	SYMBOLE	DESIGNATION
Symboles de traitement		Symboles auxiliaires	
	Symbole général Opération sur des données, instructions, ...		Renvoi Connecteur utilisé à la fin et en début de ligne pour assurer la continuité
	Sous-programme Portion de programme		Début, fin ou interruption d'un algorithme
	Entrée-Sortie Mise à disposition ou enregistrement d'une information		Liaison Les différents symboles sont reliés entre eux par des lignes de liaison. Le cheminement va de haut en bas et de gauche à droite. Un cheminement différent est indiqué à l'aide d'une flèche.
Symbole de test			
	Branchement Décision d'un choix parmi d'autres en fonction des conditions		

2. Représentation des primitives algorithmiques

2.1. L'enchaînement

L'enchaînement permet d'exécuter une série d'actions dans l'ordre de leur apparition.

Soit $A_1, A_2, \dots, A_n$ une série d'actions, leur enchaînement est représenté comme suit :



$A_1, A_2, \dots, A_n$: peuvent être des actions élémentaires ou complexes.

2.2. La structure alternative simple

Représentation algorithmique	Représentation sous forme d'organigramme
<pre> <u>si</u> <Condition> <u>alors</u> <action(s)>; <u>finsi</u>; </pre> Si la condition est vérifiée, le bloc <action(s)> sera exécuté, sinon rien, et on continue l'exécution de l'instruction après fin si.	<pre> graph TD Entry(()) --> Cond{Conditions?} Cond -- oui --> Act[Action(s)] Act --> Join(()) Cond -- non --> Join Join --> Exit[Suite de l'organigramme] </pre>

Les conditions utilisées pour les tests (simple ou double) sont des expressions logiques ou booléennes, ça veut dire des expressions dont leur évaluation donne soit TRUE (Vrai) ou FALSE (faux). Toute comparaison entre deux nombres représente une expression logique. On peut former des expressions logiques à partir d'autres expressions logiques en utilisant les opérateurs suivants : Not, Or et And.

2.3. La structure alternative double

Représentation algorithmique	Représentation sous forme d'organigramme
<pre> <u>si</u> <Condition> <u>alors</u> <action1(s)>; <u>sinon</u> <action2(s)>; <u>finsi</u>; </pre> Si la condition est vérifiée, le bloc <action1(s)> sera exécuté, sinon (si elle est fautive) on exécute <action2(s)>.	<pre> graph TD Entry(()) --> Cond{Conditions?} Cond -- oui --> Act1[Action1(s)] Cond -- non --> Act2[Action2(s)] Act1 --> Join(()) Act2 --> Join Join --> Exit[Suite de l'organigramme] </pre>

2.4. La structure itérative POUR (Boucle POUR)

Représentation algorithmique	Représentation sous forme d'organigramme
<pre> pour <cpt> ← <vi> à <vf> faire <action(s)>; finpour; </pre>	<pre> graph TD Entry(()) --> Decision{<cpt> <= <vi>} Decision -- oui --> Action[Action(s)] Action --> Increment[<cpt> ← <cpt> + 1] Increment --> Entry Decision -- non --> Exit[Suite de l'organigramme] </pre>

2.5. La structure itérative Tant-Que (Boucle Tant-Que)

Représentation algorithmique	Représentation sous forme d'organigramme
<pre> Tant-que <condition> faire <action(s)>; finpour; </pre>	<pre> graph TD Entry(()) --> Decision{Condition ?} Decision -- oui --> Action[<Action(s)>] Action --> Entry Decision -- non --> Exit[Suite de l'organigramme] </pre>

On exécute le bloc d'instructions <actions> tant que la <condition> est vérifiée (c'est-à-dire elle est vraie). Le déroulement de la boucle est comme suit :

- On évalue la condition : si la condition est fausse on sort de la boucle
- Si la condition est vraie, on exécute le bloc <actions> ; et on réévalue la condition (on revient à 1).

2.1. La structure itérative Répéter (Boucle Répéter)

Représentation algorithmique	Représentation sous forme d'organigramme
<pre> Répéter <action(s)>; Jusqu'à <condition>; </pre>	<pre> graph TD Entry(()) --> Action[<Action(s)>] Action --> Decision{Condition ?} Decision -- non --> Entry Decision -- oui --> Exit[Suite de l'organigramme] </pre>

On répète l'exécution du bloc *<action(s)>* jusqu'à avoir la condition correcte. Le déroulement est comment suit :

- 1 – On exécute le bloc *<action(s)>* ;
- 2 – On évalue la condition : si la condition est vérifiée (elle est vraie) on sort de la boucle (on continue la suite de l'algorithme);
- 3- si la condition n'est pas vérifiée (elle est fausse) on revient à 1.

Remarques :

✓ N'importe quelle boucle *POUR* peut être remplacée par une boucle *Tant-Que*, cependant l'inverse n'est pas toujours correcte, c'est-à-dire, il y a des cas où la boucle *Tant-Que* ne peut pas être remplacée par une boucle *POUR*.

✓ On transforme une boucle pour à une boucle Tant-Que comme suit :

Boucle POUR	Boucle Tant-Que
<pre> pour <cpt> ← <vi> à <vf> faire <action(s)>; finpour; </pre>	<pre> <cpt> ← <vi>; Tant-que <cpt> <= <vf> faire <action(s)>; <cpt> ← <cpt> + 1; finTant-Que; </pre>

✓ La boucle Répéter possède une condition de sortie (c'est-à-dire si elle est vraie on sort de la boucle), alors que la boucle Tant-que possède une condition d'entrée (c'est-à-dire si elle est vraie on entre dans la boucle).

✓ La boucle Répéter exécute le bloc *<action(s)>* au moins une fois, le teste vient après

Exemples

Exemple1

Ecrire un algorithme qui effectue la division de deux entiers par des soustractions successives.

Solution

```

Algorithme division ;
Var dividende, diviseur, quotient :entier ;
Début
    quotient←0 ;
    ecrire('entrer le dividende') ;
    lire(dividende) ;
    ecrire('entrer diviseur') ;
    lire(diviseur) ;
    tantque dividende ≥ diviseur faire
    début
        dividende←dividende-diviseur ;
        quotient←quotient+1 ;
    fin ;
    ecrire('le quotient est ',quotient) ;
    ecrire('le reste est ',dividende) ;
fin.

```

```

#include <stdio.h>

int main()
{
    int dividende, diviseur, quotient;

    quotient = 0;

    printf("Entrer le dividende : ");
    scanf("%d", &dividende);

    printf("Entrer le diviseur : ");
    scanf("%d", &diviseur);

    while (dividende >= diviseur)
    {
        dividende = dividende - diviseur;
        quotient = quotient + 1;
    }

    printf("Le quotient est : %d\n", quotient);
    printf("Le reste est : %d\n", dividende);

    return 0;
}

```

Exemple2

Faire une variante de ce programme en utilisant la boucle repeat...until

Solution

```

Algorithme division ;
Var dividende, diviseur, quotient :entier ;
Début
    quotient←0 ;
    ecrire('entrer le dividende') ;
    lire(dividende) ;
    ecrire('entrer diviseur') ;
    lire(diviseur) ;
    si dividende ≥ diviseur alors
    début
        répéter
            dividende←dividende-diviseur ;
            quotient←quotient+1 ;
        jusqu'à dividende < diviseur ;
    fin ;
    ecrire('le quotient est ',quotient) ;
    ecrire('le reste est ',dividende) ;
fin.

```

```

#include <stdio.h>

int main()
{
    int dividende, diviseur, quotient;

    quotient = 0;

    printf("Entrer le dividende : ");
    scanf("%d", &dividende);

    printf("Entrer le diviseur : ");
    scanf("%d", &diviseur);

    if (dividende >= diviseur)
    {
        do
        {
            dividende = dividende - diviseur;
            quotient = quotient + 1;
        }
        while (dividende >= diviseur);
    }

    printf("Le quotient est : %d\n", quotient);
    printf("Le reste est : %d\n", dividende);

    return 0;
}

```

Exercices d'application

Exercice 1 Ecrire un algorithme qui demande à l'utilisateur un nombre compris entre 1 et 3 jusqu'à ce que la réponse convienne.

Exercice 2 Ecrire un algorithme qui demande un nombre compris entre 10 et 20, jusqu'à ce que la réponse convienne. En cas de réponse supérieure à 20, on fera apparaître un message : « Plus petit ! », et inversement, « Plus grand ! » si le nombre est inférieur à 10.

Exercice 3 Ecrire un algorithme qui demande un nombre de départ, et qui ensuite affiche les dix nombres suivants. Par exemple, si l'utilisateur entre le nombre 17, le programme affichera les nombres de 18 à 27.

Exercice 4 Ecrire un algorithme qui demande un nombre de départ, et qui ensuite écrit la table de multiplication de ce nombre, présentée comme suit (cas où l'utilisateur entre le nombre 7) : Table de 7 : $7 \times 1 = 7$

$$7 \times 2 = 14$$

$$7 \times 3 = 21 \dots$$

$$7 \times 10 = 70.$$

Exercice 5 Ecrire un algorithme qui demande un nombre positif de départ, et qui calcule la somme des entiers jusqu'à ce nombre. Par exemple, si l'on entre 5, le programme doit calculer : $1 + 2 + 3 + 4 + 5 = 15$ NB : on souhaite afficher uniquement le résultat, pas la décomposition du calcul.

Exercice 6 Ecrire un algorithme qui demande un nombre de départ, et qui calcule sa factorielle. NB : la factorielle de 8, notée $8!$, vaut $1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7 \times 8$

Exercice 7 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 :

Entrez le nombre numéro 1 : 12

Entrez le nombre numéro 2 : 14 etc.

Entrez le nombre numéro 20 : 6

Le plus grand de ces nombres est : 14

Modifiez ensuite l'algorithme pour que le programme affiche de surcroît en quelle position avait été saisie ce nombre : C'était le nombre numéro 2.