CHAPITRE N° 3

STRUCTURES RÉPÉTITIVES : FOR, WHILE DO-WHILE

Boucle **for**

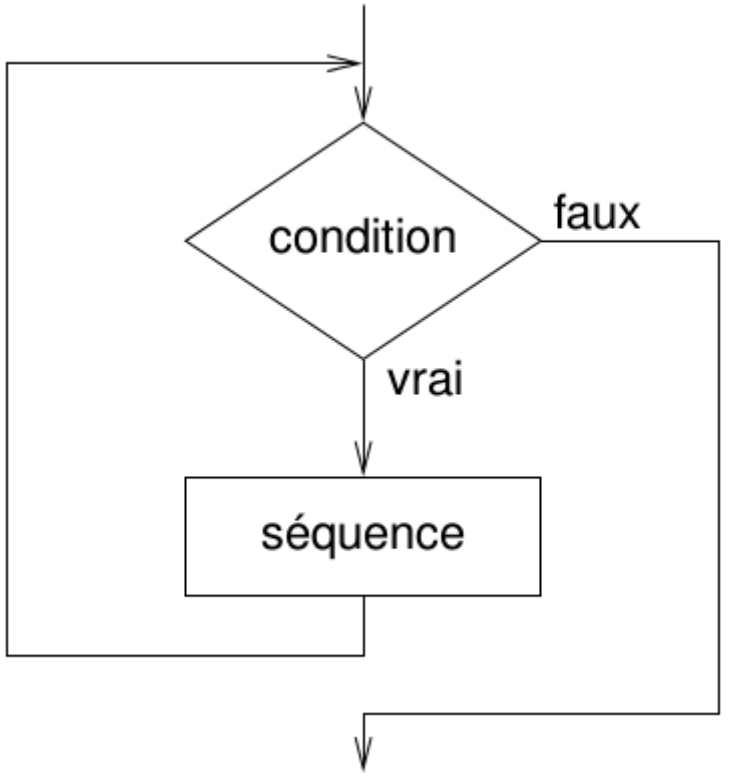
- Exécute un bloc de code un certain nombre de fois.
- Le bloc de code peut n'avoir aucune instruction, une instruction ou plus.
- L'instruction **for** entraîne l'exécution de la boucle **for** dans un nombre fixe de fois.
- Ce qui suit est la syntaxe de la boucle **for** :

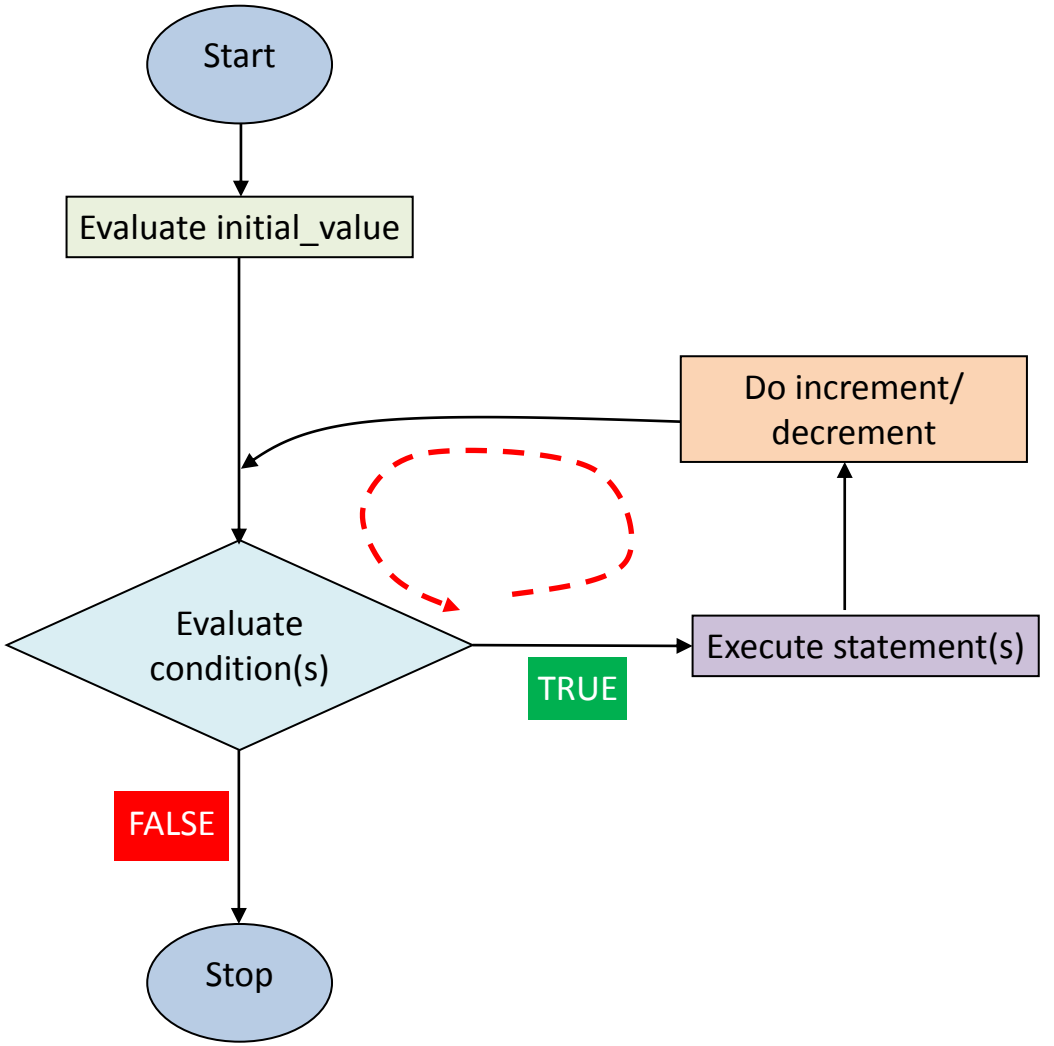
```
for(initial_value;condition(s);increment/decrement)
      statement(s);
next_statement;
```

- **initial_value**, **condition(s)** et **incrémentation / décrémentation** sont toutes les expressions valides en C.
- **statement(s)** peuvent être une instruction C unique ou composée (un bloc de code) **{}**.
- Pendant l'exécution du programme, l'instruction **for** est rencontrée les événements suivants :
  1. **initial_value** est évaluée par ex. `intNum = 1`.
  2. Ensuite, la ou les conditions sont évaluées, généralement une expression relationnelle.
  3. Si **condition(s)** sont évaluées à FALSE , l'instruction **for** se termine et l'exécution passe à l'instruction suivante.
  4. Si **condition(s)** sont évaluées comme TRUE, **statement(s)** sont exécutées.
  5. Ensuite, **l'incrémentation / décrémentation** est exécutée et l'exécution revient à l'étape **n °. 2** jusqu'à ce que **condition(s)** deviennent FAUX.

Boucle **for**

- L'organigramme de la boucle **for** devrait ressembler à ce qui suit





Programme d'exemple : imprimant les entiers de 1 à 10.

dixPremiersEntiers.c

```c
#include <stdio.h>
void main()
{
    int nCount;
    // Affficher les entiers de 1 à 10
    for(nCount = 1; nCount <= 10; nCount++)
        printf("%d ", nCount);
    printf("\n");
}
```

Sortie :

```
1 2 3 4 5 6 7 8 9 10
```

Boucle `for` : est très flexible

- La boucle **for** est une construction très flexible,
- Peut utiliser le compteur décrémentant au lieu d'incrémenter. Par exemple,

```
for(nCount = 100; nCount > 0; nCount--)
```

- Peut utiliser un compteur autre que 1, par exemple 3,

```
for(nCount = 0; nCount < 1000; nCount += 3)
```

- La valeur initiale peut être omise si la variable de test a été initialisée au préalable.
- Cependant, le point-virgule doit toujours être là. Par exemple,

```
nCount=1;
for(   ;nCount < 1000; nCount ++)
```

- La valeur initiale peut être n'importe quelle expression valide en C, l'expression est exécutée une fois lorsque l'instruction **for** est atteinte pour la première fois. Par exemple,

```
nCount =1;
for(printf("Début de la boucle"); nCount < 1000; nCount ++)
```

Boucle **for** : est très flexible

- L'expression d'incrémentation / décrémentation peut être omise tant que la variable de compteur est mise à jour dans le corps de l'instruction **for**.
- Le point-virgule doit toujours être inclus. Par exemple,

```
for(nCount =0; nCount < 100; )
    printf("%d", nCount++);
```

- `condition(s)` qui termine la boucle peut être n'importe quelle expression valide en C.
- Tant qu'elle est évaluée comme TRUE , l'instruction **for** continue de s'exécuter.
- Les opérateurs logiques peuvent être utilisés pour construire des expressions de `condition(s)` plus complexes. Par exemple,

```
for(nCount =0; nCount < 1000 && name[nCount] != 0; nCount ++)
    printf("%d", name[nCount]);
for(nCount = 0; nCount < 1000 && list[nCount];)
    printf("%d", list[nCount++]);
```

Remarque: *les instructions* **for** *et les tableaux sont étroitement liés, il est donc difficile de définir l'un sans expliquer l'autre (ce sera discuté dans un autre chapitre).*

Boucle `for :` est très flexible

- La ou les instructions **`for`** peuvent être suivies d'une instruction **`NULL`** (vide), de sorte que la tâche est effectuée dans la boucle **`for`** elle-même,
- L'instruction **`NULL`** se compose d'un point-virgule seul sur une ligne. Par exemple,

```
for(count = 0; count < 20000; count++)
    ;
```

- Cette instruction fournit une pause (délai) de 20 000 millisecondes.

Boucle **for** : est très flexible

- Une expression peut être créée en séparant deux sous-expressions avec *l'opérateur virgule*, et sont évaluées (dans l'ordre de gauche à droite), et l'expression entière est évaluée à la valeur de la sous-expression.

*"Nous avons deux tableaux de **1000** éléments chacun, nommés **a[]** et **b[]**. Ensuite, nous voulons copier le contenu de **a[]** vers **b[]** dans l'ordre inverse, donc, après l'opération de copie, le contenu du tableau doit être…"*

b[0], b[1], b[2],… et a[999], a[998], a[997],… et ainsi de suite.

- Le code doit être comme suit :

```
for(iRow = 0, jColumn = 999; iRow < 1000; iRow++, jColumn--)
    b[jColumn] = a[iRow];
```

Boucle **for** : est très flexible

- Le bloc de programme ci-dessus calculera et affichera la somme des 20 premiers nombres naturels,
- L'exemple ci-dessus peut être réécrit comme suit:

```
for(iNum = 1, nSom = 0; iNum <= 20; iNum++)
    nSom = nSom + iNum;
printf("Somme des 20 premiers nombres naturels = %d", nSom);
```

Notez que la partie d'initialisation a deux instructions séparées par une virgule (,).

- Autre exemple

sommesDixPremiersPairsEtImpairs.c

```
for(iNum = 1, nSom1=0, nSom2 = 0; iNum <= 20; iNum = iNum + 1)
{
     if(iNum%2==0)
    nSom1 = nSom1 + iNum;
     else
    nSom2 = nSom2 + iNum;
}
printf("Somme des 10 premiers nombres naturels pairs = %d\n", nSom1);
printf("Somme des 10 premiers nombres naturels impairs = %d\n", nSom2);
```

- Dans cet exemple, l'instruction **for** est une instruction composée ou bloc.

Boucle **for :** est très flexible

- Nous pouvons également créer une boucle infinie ou sans fin en mettant toutes les expressions ou en utilisant une constante non **NULLE** pour `condition(s)` , comme indiqué dans les deux extraits de code suivants :

```
for( ; ; )
    printf("Ceci est une boucle infinie\n");
```

- or

```
for( ; 1 ; )
    printf("Ceci est une boucle infinie\n");
```

- Dans les deux cas, le message "Ceci est une boucle infinie" sera imprimé à plusieurs reprises, indéfiniment.
- Toutes les constructions de répétition discutées jusqu'à présent peuvent être imbriquées à n'importe quel degré.

Boucle **while**

- Exécute un bloc d'instructions tant qu'une condition spécifiée est TRUE.
- La construction générale de la boucle

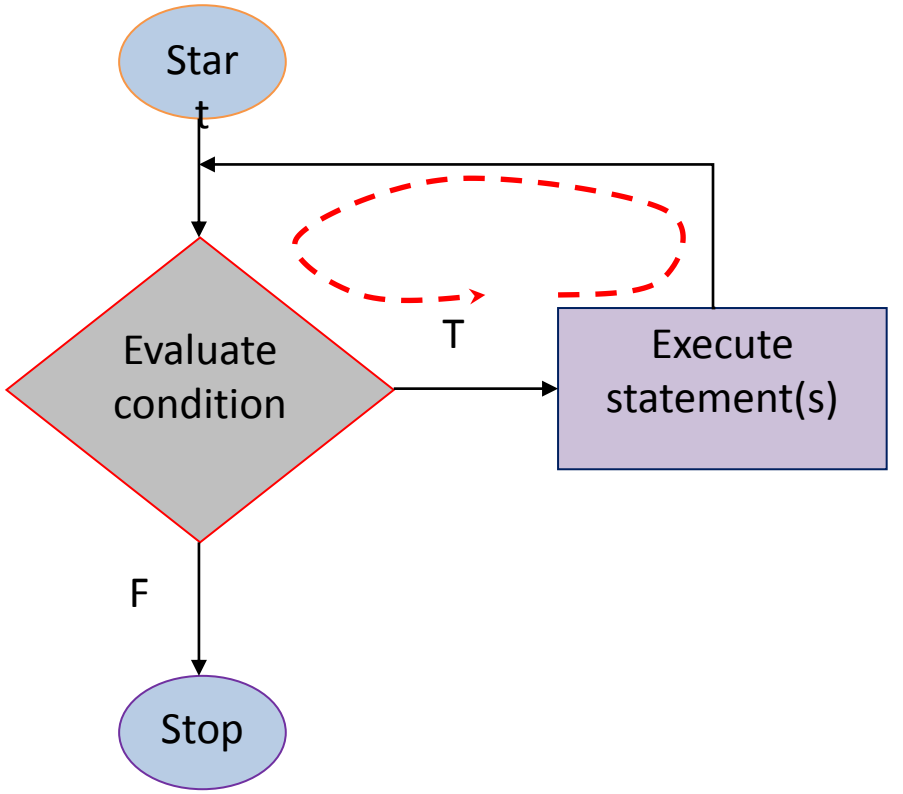
```
while( condition(s) )
      statement(s);
next_statement;
```

- **condition(s)** peut être n'importe quelle expression valide en C.
- **statement(s);** peuvent être une instruction unique ou composée en C(un bloc de code).
- Lorsqu'une instruction **while** est rencontrée, les événements suivants se produisent:

1. **condition(s)** est évaluée.
2. Si **condition(s)** prend la valeur **FALSE** , la boucle **while** se termine et l'exécution passe à **next_statement**.
3. Si **condition(s)** est évaluée comme **TRUE** , la ou les **next_statement** en C sont exécutées.
4. Ensuite, l'exécution revient à l'étape **numéro 1** jusqu'à ce que la condition devienne **FALSE**.

Boucle **`while`**

- L'organigramme de la boucle **`while`** est illustré ci-dessous.

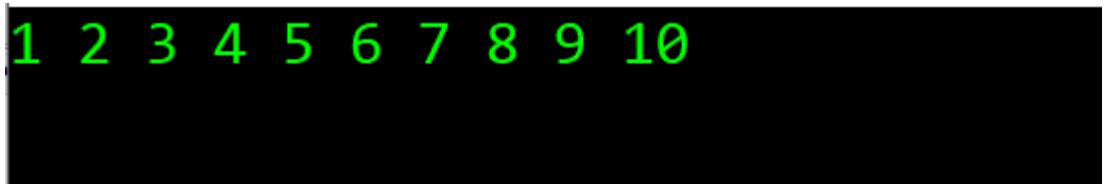


Programme d'exemple : imprimant les entiers de 1 à 10.

dixPremiersEntiers.c

```c
#include <stdio.h>
void main()
{
  int  nCalculate = 1;
  // définit la condition while
  while(nCalculate <= 10)
  {
      // Afficher
      printf("%d ", nCalculate);
      // incrément de 1, se répète
      nCalculate++;
   }
   // Retour à la ligne
   printf("\n");}
```

Sortie :



Boucle **while**

- La même tâche qui peut être effectuée à l'aide de l'instruction **for**.
- Mais, alors que la boucle **while** ne contient pas de section d'initialisation, le programme doit initialiser explicitement toutes les variables au préalable.
- En conclusion, la boucle **while** est essentiellement une instruction **for** sans les composants *d'initialisation* et *d'incrémentation*.
- La comparaison syntaxique entre for et while,

```
for( ; condition; )   vs   while(condition)
```

- Tout comme les instructions for et if, les instructions **while** peuvent également être imbriquées.

Boucle **do-while**

- Exécute un bloc d'instructions tant qu'une condition spécifiée est VRAIE au moins une fois.
- Testez la condition à la fin de la boucle plutôt qu'au début, comme le démontrent les boucles **for** et **while**.
- La construction de la boucle **do-while** est :

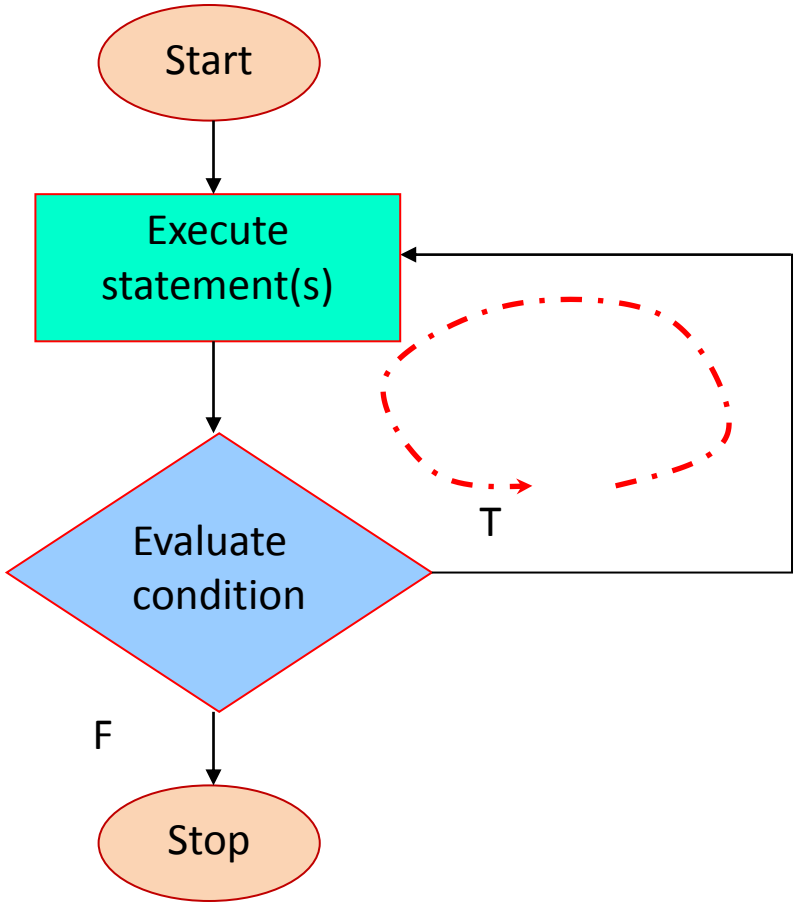
```
do
      statement(s);
while ( condition(s) ) ;
next_statement;
```

- **condition(s)** peut être n'importe quelle expression valide en C.
- **statement(s)** peut être une instruction unique ou composée en C(un bloc de code).
- Lorsque le programme rencontre la boucle **do-while**, les événements suivants se produisent:

1. **statement(s)** sont exécutées.
2. **condition(s)** est évaluée. Si elle est **TRUE**, l'exécution retourne à l'étape numéro 1. Si elle est **FALSE**, la boucle se termine et l'instruction **next_statement** est exécutée.
3. Cela signifie que les instructions de la boucle **do-while** seront exécutées au moins une fois.

Boucle **do-while**

- Un organigramme pour la boucle **do-while**



- Les instructions sont toujours exécutées au moins une fois.
- Les boucles **for** et **while** évaluent la condition au début de la boucle, de sorte que les instructions associées ne sont pas exécutées si la condition est initialement **FALSE**.

Exemple : Programme de calcul de la racine carrée

racineCarree.c

```c
#include <stdio.h>
void main()
{
        float N;
        do
        {
            printf("Entrer un nombre (>= 0) : ");
            scanf("%f", &N);
        }
        while (N < 0);
        printf("La racine carrée de %.2f est %.2f\n", N,
sqrt(N));
}
```

Sortie :

```
Entrer un nombre (>= 0) : -5
Entrer un nombre (>= 0) : -2
Entrer un nombre (>= 0) : -7
Entrer un nombre (>= 0) : -8.25
Entrer un nombre (>= 0) : 18
La racine carrÚe de 18.00 est 4.24
```