



# **Les Structures de Contrôle**

Les structures de contrôle définissent la suite dans laquelle les instructions sont effectuées. Nous allons apprendre à contrôler les flux des instructions. En effet un programme n'est pas qu'une simple suite d'instructions se déroulant linéairement une fois et une seule.

## **I. Structures de test**

Ces instructions permettent de conditionner l'exécution d'instructions à la valeur de vérité d'une expression.

### **1. Structure conditionnelle simple**

L'instruction la plus usitée est le SI qui a besoin d'une expression et d'un bloc d'instructions. Cette expression sera évaluée en contexte scalaire et servira de condition ; si elle est vérifiée, le bloc d'instructions sera exécuté.

La construction conditionnelle permet de construire une instruction complexe de la forme :

```
if(expression_bool)
{
    instruction1;
    instruction2;
    ...
}
```

- L'expression booléenne est tout d'abord calculée,
- Puis les instructions comprises entre les accolades ne sont exécutées que si le résultat du calcul de l'expression booléenne donne vrai.
- Puis la suite, s'il y a une séquence d'instructions après, elle sera exécutée.

### **Remarque :**

N'oubliez pas que toute instruction simple est toujours terminée par un point- virgule. Ainsi, ce bloc :

```
{ i=5;k=3}
```

est incorrect car il manque un point-virgule à la fin de la seconde instruction.

D'autre part, un bloc joue le même rôle syntaxique qu'une instruction simple (point-virgule compris). Évitez donc d'ajouter des points-virgules intempestifs à la suite d'un bloc.

### **2. Construction alternative**

Il est possible d'exécuter d'autres instructions dans le cas où la condition est fausse. On utilise pour cela l'opérateur ELSE qui, lui aussi, est suivi d'un bloc d'instructions.

La construction alternative permet de construire une instruction complexe de la forme :

```

If (expression_bool)
{
    instruction11;
    instruction12;
    ...
}
else
{
    instruction21;
    instruction22;
    ...
}

```

- L'expression booléenne est tout d'abord calculée,
- Si le résultat du calcul donne vrai, les instructions 11, 12 ...sont exécutées,
- Si le résultat du calcul donne faux, les instructions 21, 22... sont exécutées,
- Puis la suite, s'il y en a une séquence d'instruction après, est sera exécutée dans les 2 cas.

### 3. Imbrication de if

Pour résoudre un problème complexe, un algorithme peut devenir complexe. Chacune des instructions le composant - y compris celles figurant dans une construction conditionnelle ou alternative - peut être remplacée par une construction conditionnelle ou alternative.

On peut obtenir alors un algorithme en forme de "poupée russe", où les constructions sont emboîtées les unes dans les autres.

```

IF                                (expr_bool1)
{
    ...
}
ELSE
    IF                                (expr_bool2)
    {
        ...
    }
    ELSE ...
...

```

### 4. Choix selon des valeurs énumérées

Cette construction diffère des précédentes, car elle ne fait pas usage d'expression booléenne.

On l'utilise quand, en fonction d'une seule donnée, on souhaite effectuer un et un seul traitement parmi une série de traitements possibles.

```

switch(nom_de_la_variable)
{
    case                                valeur_1:
        Instructions à exécuter dans le cas où la variable vaut valeur_1
        break;
    case                                valeur_2:
        Instructions à exécuter dans le cas où la variable vaut valeur_2
        break;
    default:
        Instructions à exécuter dans le cas où la variable vaut une valeur autre que
        valeur_1                                et                                valeur_2
        break;
}

```

- L'expression est tout d'abord calculée, elle doit avoir un résultat de type "énumérable", c'est-à-dire dont on peut énumérer les valeurs : en pratique de type entier ou caractère,
- Si le résultat du calcul donne une des valeurs prévues, alors les instructions correspondantes sont exécutées ;
- Si le résultat du calcul donne une valeur qui n'était pas prévue, les instructions associées au cas autrement sont exécutées. Le cas autrement est facultatif.

Cette construction est pratique pour la partie d'algorithme suivante, dans laquelle on suppose que la variable jour contient un entier - compris entre 1 et 7 - représentant le jour de la semaine (du lundi au dimanche). L'algorithme affiche en toutes lettres le nom du jour correspondant.

```

...
switch (jour)
{
    1 : printf("Lundi"); break;
    2 : printf("Mardi"); break;
    3 : printf("Mercredi"); break;
    4 : printf("Jeudi"); break;
    5 : printf("Vendredi"); break;
    6 : printf("Samedi"); break;
    7 : printf("Dimanche"); break;
    default : printf("Erreur"); break;
}

```

Il est possible d'utiliser le mot-clé **default** comme étiquette à laquelle le programme se branchera dans le cas où aucune valeur satisfaisante n'aura été rencontrée auparavant.

Il est aussi possible d'écrire un algorithme équivalent en utilisant des constructions alternatives, mais ceci est quelque peu fastidieux.

```

...
if (jour=1) printf('Lundi')
else if (jour=2) printf('Mardi')
    else if (jour=3) printf('Mercredi')
        else if (jour=4) printf('Jeudi')
            else if (jour=5) printf('Vendredi')
                else if (jour=6) printf('Samedi')

```

```
    else if (jour=7) printf('Dimanche')
    else printf('Erreur')
...

```

Il est d'ailleurs toujours possible de remplacer une construction **switch** par une série de constructions **if** imbriquées.

On conviendra cependant facilement à la vue de cet exemple, que la construction **switch**, permet un raccourci évident d'écriture, quand cette construction peut être utilisée; elle évite d'utiliser une trop grande suite de SI.

**Remarque :**

Un **else** se rapporte toujours au dernier **if** rencontré auquel un **else** n'a pas encore été attribué.

## II. Structures répétitives

L'objectif est de prévoir par programme la répétition de certains traitements, sans avoir à les répéter (recopier) dans l'écriture du programme. Recopier ne serait d'ailleurs pas possible si l'on souhaite répéter un calcul autant de fois que nécessaire pour que le résultat satisfasse une propriété donnée.

### 1. La boucle WHILE

```
while ( <expression> )
    <bloc d'instructions>

```

1. l'expression booléenne est tout d'abord calculée; si le calcul de l'expression booléenne donne vrai, les instructions comprises dans le bloc d'instructions sont exécutées;
2. puis l'exécution recommence au point 1;
3. la suite, s'il y en a une après le bloc, est exécutée dès que le calcul de l'expression booléenne, effectué au point 1, donne faux.

**Remarques :**

- L'expression booléenne d'une construction **while** doit pouvoir être calculée dès le début de la construction : les variables qui apparaissent dans l'expression doivent donc toutes avoir une valeur avant.
- Les instructions contenues dans la construction **while** peuvent ne jamais être exécutées, si dès la première fois, l'expression booléenne a la valeur faux.
- Les instructions contenues dans la construction **while** doivent contribuer à modifier le calcul de l'expression booléenne, pour que ce calcul finisse par donner la valeur faux. L'expression booléenne d'une construction **while** est aussi appelée « condition de continuation ».

*Exemple 1*

```
/* Afficher les nombres de 0 à 9 */

```

```

int I = 0;
while (I<10)
{
    printf("%i \n", I);
    I++;
}

```

#### Exemple 2

```

int I;
/* Afficher les nombres de 0 à 9 */
I = 0;
while (I<10)
    printf("%i \n", I++);
/* Afficher les nombres de 1 à 10 */
I = 0;
while (I<10)
    printf("%i \n", ++I);

```

#### Exemple 3

```

/* Afficher les nombres de 10 à 1 */
int I=10;
while (I)
    printf("%i \n", I--);

```

## 2. la boucle DO ... WHILE

Dans le cas où l'on souhaite exécuter le bloc d'instruction une fois avant d'effectuer le test, on peut utiliser la boucle DO .. WHILE.

```

do
    <bloc d'instructions>

while ( <expression> );

```

1. les instructions comprises entre **do** et **while** sont exécutées
2. l'expression booléenne est alors calculée ; si le calcul de l'expression booléenne donne faux, l'exécution recommence au point 1 ; si le calcul de l'expression booléenne donne vrai, l'exécution continue avec la suite, s'il y en a une après **while**.

### Remarques :

L'expression booléenne d'une construction **do ... while** n'a besoin d'être calculée, pour la première fois, qu'après une première exécution des instructions contenues dans la construction **do ... while**.

Les instructions contenues dans la construction **do ... while** sont toujours exécutées au moins une fois.

Les instructions contenues dans la construction **do ... while** doivent contribuer à modifier le calcul

de l'expression booléenne, pour que ce calcul finisse par donner la valeur vrai. L'expression booléenne d'une construction **do ..while** est aussi appelée "condition d'arrêt".

*Exemple 1*

```
float N;
do
{
    printf("Introduisez un nombre entre 1 et 10 :");
    scanf("%f", &N);
}
while (N<1 // N>10);
```

*Exemple 2*

```
int n, div;
printf("Entrez le nombre à diviser : ");
scanf("%i", &n);
do
{
    printf("Entrez le diviseur ( 0) : ");
    scanf("%i", &div);
}
while (!div);
printf("%i / %i = %f\n", n, div, (float)n/div);
```

**do - while** est comparable à la structure répéter du langage algorithmique (**repeat until** en Pascal) si la condition finale est inversée logiquement.

### 3. la boucle FOR

A la différence des deux constructions précédentes, la construction FOR est destinée au cas particulier où l'on connaît par avance le nombre de fois où l'on veut répéter certains traitements.

La construction for permet de répéter *n* fois un traitement ; elle permet de plus de faire évoluer une variable (appelée "compteur"), en lui affectant pour chacune des exécutions une nouvelle valeur (incrémentée ou décrémentée – par défaut, ce sera incrémenté de 1).

```
for ( <expr1> ; <expr2> ; <expr3> )
    <bloc d'instructions>
```

est équivalent à :

```
<expr1>;
while ( <expr2> )
{
    <bloc d'instructions>
    <expr3>;
}
```

**<expr1>** est évaluée une fois avant le passage de la boucle. Elle est utilisée pour initialiser les données de la boucle.

<expr2> est évaluée avant chaque passage de la boucle. Elle est utilisée pour décider si la boucle est répétée ou non.

<expr3> est évaluée à la fin de chaque passage de la boucle. Elle est utilisée pour réinitialiser les données de la boucle.

Le plus souvent, **for** est utilisé comme boucle de comptage :

```
for ( <init.> ; <cond. répétition> ; <compteur> )  
    <bloc d'instructions>
```

*Exemple 1*

```
int I;  
for (I=0 ; I<=20 ; I++)  
    printf("Le carré de %d est %d \n", I, I*I);
```

En pratique, les parties <expr1> et <expr2> contiennent souvent plusieurs initialisations ou réinitialisations, *séparées par des virgules*.

*Exemple 2*

```
int n, tot;  
for (tot=0, n=1 ; n<101 ; n++)  
    tot+=n;  
printf("La somme des nombres de 1 à 100 est %d\n", tot);
```

#### 4. Choix de la structure répétitive

Nous avons vu trois façons différentes de programmer des boucles (**while**, **do - while**, **for**). Utilisez la structure qui reflète le mieux l'idée du programme que vous voulez réaliser, en respectant toutefois les directives suivantes :

- \* Si le bloc d'instructions ne doit pas être exécuté si la condition est fausse, alors utilisez **while** ou **for**.

- \* Si le bloc d'instructions doit être exécuté au moins une fois, alors utilisez **do - while**.

- \* Si le nombre d'exécutions du bloc d'instructions dépend d'une ou de plusieurs variables qui sont modifiées à la fin de chaque répétition, alors utilisez **for**.

- \* Si le bloc d'instructions doit être exécuté aussi longtemps qu'une condition extérieure est vraie (p.ex aussi longtemps qu'il y a des données dans le fichier d'entrée), alors utilisez **while**.

Le choix entre **for** et **while** n'est souvent qu'une question de préférence ou d'habitudes:

- \* **for** nous permet de réunir avantageusement les instructions qui influencent le nombre de répétitions au début de la structure.

- \* **while** a l'avantage de correspondre plus exactement aux structures d'autres langages de programmation (**while**, **tant que**).


- \* **for** a le désavantage de favoriser la programmation de structures surchargées et par la suite illisibles.

- \* **while** a le désavantage de mener parfois à de longues structures, dans lesquelles il faut chercher pour trouver les instructions qui influencent la condition de répétition.



### III. Les instructions de branchement inconditionnel


#### 1. L'instruction **break**

 Nous avons déjà vu le rôle de `break` au sein du bloc régi par une instruction `switch`.

Le langage C autorise également l'emploi de cette instruction dans une boucle. Dans ce cas, elle sert à interrompre le déroulement de la boucle, en passant à l'instruction qui suit cette boucle. Bien que si son exécution est conditionnée par un choix ; dans le cas contraire, en effet, elle serait exécutée dès le premier tour de boucle, ce qui rendrait la boucle inutile.

Exemple :

```
main() {  
  
    int i ;  
    for ( i=1 ; i<=10 ; i++ )  
        { printf ("début tour %d\n", i) ;  
          printf ("bonjour\n")  
          if ( i==3 ) break ;  
          printf ("fin tour %d\n", i) ;  
        }  
    printf ("après la boucle") ;  
}
```

```
début tour 1  
bonjour  
fin tour 1  
début tour 2  
bonjour  
fin tour 2  
début tour 3  
bonjour  
après la boucle
```

#### Remarque :

En cas de boucles imbriquées, `break` fait sortir de la boucle la plus interne. De même si `break` apparaît dans un `switch` imbriqué dans une boucle, elle ne fait sortir que du `switch`.

#### 2. L'instruction **continue**

L'instruction `continue`, quant à elle, permet de passer prématurément au tour de boucle suivant. En voici un premier exemple avec `for` :

```

main()
{
    int i ;
    for ( i=1 ; i<=5 ; i++ )
        { printf ("début tour %d\n", i) ;

        if (i<4) continue ;
        printf ("bonjour\n") ;
    }
}

```

```

début tour 1
début tour 2
début tour 3
début tour 4
bonjour
début tour 5
bonjour

```

## Remarques :

- Lorsqu'elle est utilisée dans une boucle for, cette instruction « continue » effectue bien un branchement sur l'évaluation de l'expression de fin de parcours de boucle (nommée `expression_2` dans la présentation de sa syntaxe), et non après.
- En cas de boucles imbriquées, l'instruction *continue* ne concerne que la boucle la plus interne.

### 3. L'instruction goto

Elle permet classiquement le branchement en un emplacement quelconque du programme. Voyez cet exemple qui simule, dans une boucle for, l'instruction `break` à l'aide de l'instruction `goto` (ce programme fournit les mêmes résultats que celui présenté comme exemple de l'instruction `break`).

```

main() {

    int i ;
    for ( i=1 ; i<=10 ; i++ )
        { printf ("début tour %d\n", i) ;
          printf ("bonjour\n") ;
          if ( i==3 ) goto sortie ;
          printf ("fin tour %d\n", i) ;
        }
    sortie : printf ("après la boucle") ;
}

```

```

début tour 1

```

```
bonjour  
fin tour 1  
début tour 2  
bonjour  
fin tour 2  
début tour 3  
bonjour  
après la boucle
```