Chapitre 24 - Les boucles et les tests conditionnels

Les boucles

Les boucles for et l'incrémentation

Dans un programme, une boucle est utilisée pour répéter plusieurs fois les mêmes instructions. Rappelez-vous la méthode proposée pour changer une roue d'un véhicule ; elle est composée d'une suite d'instructions élémentaires à faire dans un certain ordre :

- Retirer l'enjoliveur
- Débloquer les quatre boulons
- Positionner le cric à l'endroit repéré au plus près de la roue
- Monter le cric jusqu'à ce que la roue décolle du sol
- Retirer complètement les quatre boulons
- Dégager la roue
- Positionner la nouvelle roue à pneus spéciaux
- Remettre les quatre boulons
- Descendre le cric complètement jusqu'à ce que la voiture soit sur le sol
- Dégager le cric de la voiture
- Serrer à fond les quatre boulons
- Remettre l'enjoliveur

Si nous devons changer toutes les roues de notre véhicule, nous pouvons recopier cette série d'instructions quatre fois, mais il est bien plus simple de ne la donner qu'une seule fois et de dire de l'appliquer à chaque roue. L'opérateur peut alors utiliser un compteur : chaque fois qu'il a terminé la série d'instructions, il rajoute une croix sur un bout de papier pour se rappeler combien de roues ont été faites. Le nombre de croix est le **compteur** et chaque fois que la série d'instructions est réalisée, ce compteur augmente d'une unité : on dit qu'il est **incrémenté**. Lorsque le compteur atteint quatre, toutes les roues de la voiture ont été changées et on peut arrêter le travail. Pour un camion, il faudrait arrêter le travail lorsque le compteur devient égal au nombre de roues de celui-ci.

On peut aussi partir d'un compteur égal au nombre de roues, par exemple quatre sur une voiture, et chaque fois que la série d'instructions est réalisée, on diminue ce compteur d'une unité pour se rappeler de combien de roues il reste à faire. Lorsque celui-ci arrive à zéro, il ne reste plus aucune roue à traiter et on arrête le travail ; dans ce cas, le compteur est **décrémenté**.

Il est donc très facile de créer une boucle dans un programme informatique ; heureusement, tous les langages informatiques ont prévu cette possibilité, ce qui simplifie le travail du programmeur.

Boucle for

La boucle for est utilisée pour répéter plusieurs fois les mêmes instructions dans un programme. Elle utilise un compteur incrémenté (ou décrémenté) à chaque passage dans la boucle et permettant de sortir de la boucle lorsque ce compteur arrive à une certaine valeur.

La syntaxe à respecter pour une boucle for est :

```
for (initialisation ; condition ; incrément) {

// Ce qu'il faut répéter

}
```

Entre les deux accolades { }, on place les instructions à exécuter. Autrement, on a des parenthèses ( ) et des points-virgules ; (**attention à cela !**).

Voyons maintenant ce qui se trouve dans les parenthèses.

Initialisation

Ceci est réalisé en premier et une seule fois. C'est là qu'on déclare le compteur (en général int, ou byte si la boucle est parcourue moins de 256 fois) et qu'on l'initialise (en général 0 ou 1, mais on peut faire différemment).

Condition

À chaque itération, la condition est testée. Si la condition est vraie, les instructions sont exécutées et le compteur est incrémenté (ou décrémenté). Lorsque la condition devient fausse, la boucle prend fin, c'est-à-dire que le programme sort de la boucle et exécute les instructions qui la suivent.

Incrément

On augmente le compteur de la boucle à chaque itération (cas général) mais on peut aussi le décrémenter (dans ce cas, on fait un compte à rebours en quelque sorte).

Pour préciser ces trois derniers paragraphes, voyons un exemple.

Cas général d'une boucle à parcourir 5 fois

Voici la syntaxe à écrire :

```
for (int compteur = 0 ; compteur < 5 ; compteur++) {

// Mettre ici les instructions à exécuter 5 fois

}
```

Initialisation : on utilise un entier appelé compteur comme compteur de boucle et on l'initialise à 0 (premier chiffre disponible).

Condition : la boucle est répétée tant que le compteur est strictement inférieur à 5 (donc pour 0, 1, 2, 3, 4, ce qui fait bien parcourue 5 fois). Lorsque compteur est égal à 5, on sort immédiatement de la boucle.

Incrément : à chaque itération (un parcours de boucle), le compteur augmente d'une unité.

compteur++ est équivalent à compteur = compteur + 1

On peut bien entendu initialiser le compteur à 1 (pour ceux qui ont du mal avec le chiffre 0) et dans ce cas, pour parcourir la boucle 5 fois, la condition devient :

compteur <= 5

Nous pouvons donc obtenir le même résultat en écrivant :

```
for (int compteur = 1 ; compteur <= 5 ; compteur++) {

// Mettre ici les instructions à exécuter 5 fois

}
```

La boucle est bien parcourue tant que compteur est égal à 1, 2, 3, 4, 5, donc elle est parcourue 5 fois.

Dans les boucles for, le compteur s'appelle i, j, k, etc. : il n'est pas nécessaire de lui donner un nom plus élaboré car on comprend qu'il s'agit bien du compteur de boucle. Si on initialise à 0 (comme font tous les informaticiens), pour avoir 5 itérations, il suffit de mettre < 5 et si on préfère initialiser à 1, il faut mettre <= 5.

Compte à rebours

La boucle suivant exécutera un compte à rebours :

for (int i = 10 ; i > 0 ; i--) {

// Mettre ici les intructions à exécuter

}

Le compteur i va de 10 jusqu'à 1.

i--est équivalent à i = i - 1

Puissance de la boucle for en langage C

La boucle for peut être parcourue en faisant croitre puis décroitre son compteur comme le montre l'exemple suivant :

```
void loop()
{
  int x = 1;
  for (int i = 0; i > -1; i = i + x){
    Serial.println (i) ;
    if (i == 255) x = -1;            // inverse l'incrémentation, ce qui donne une décrémentation
    delay(10);
  }
}
```

Dans cet exemple, i est incrémenté de 0 à 255, puis décrémenté de 255 à 0. La boucle s'arrête lorsque i devient égal à -1. Remarquez que dans le test if, on a omis les accolades { } car si le test est vrai, il n'y a qu'une seule instruction à exécuter ; j'ai laissé cet exemple tel que je l'ai trouvé, mais il est préférable d'avoir le réflexe de mettre les accolades.

Si on utilise une multiplication dans l'incrément, on obtient une progression logarithmique comme le montre l'exemple suivant :

```
for(int x = 2; x < 100; x = x * 1.5){
Serial.println(x);
}
```

Ce programme affiche sur le moniteur :

2, 3, 4, 6, 9, 13, 19, 28, 42, 63, 94 (seule la partie entière est conservée puisque x est déclaré comme un entier).

Les deux derniers exemples sont tirés du site officiel d'Arduino ; il faut bien entendu initialiser les communications série dans le setup.

Les boucles while et do…while

Maintenant que nous avons étudié les boucles for, voyons les boucles while et do…while.

Boucle while

La syntaxe est la suivante :

```
while (expression) {

// Instruction à exécuter

}
```

Cette boucle est parcourue **indéfiniment** jusqu'à ce que l'expression entre parenthèses devienne fausse. On l'utilise quand on ne sait pas à l'avance combien d'itérations il faut faire, par exemple lorsqu'on attend qu'un bouton poussoir soit enfoncé (combien de lectures faudra-t-il faire avant que le B/P soit enfoncé ?).

On peut aussi utiliser une boucle while pour répéter un certain nombre de fois une série d'instruction ; dans ce cas, on utilise un compteur, comme le montre l'exemple ci-dessous :

```
int compteur = 0 ;

while (compteur < 100) {

// Répète 100 fois

compteur++ ;

}
```

L'expression de la parenthèse reste vraie tant que le compteur évolue de 0 à 99, lorsque compteur = 100, l'expression n'est plus vraie (car il n'est plus inférieur à 100, il est égal à 100).

Dans une boucle while, l'expression est **testée en début de boucle** et la boucle est parcourue tant que l'expression est vraie. Dès que l'expression est fausse, on sort de la boucle.

Boucle do…while

La boucle do… while a une autre philosophie puisque l'expression est **testée en fin de boucle**. Dans ces conditions, la boucle do…while est **parcourue (et exécutée) au moins une fois**.

La syntaxe à respecter est la suivante :

```
do {

// Instructions à répéter

} while (expression) ;
```

Attention à ne pas oublier le point-virgule à la fin !

Boucle sans fin

Dans les boucles while ou do…while, l'expression qui est testée est soit vraie, soit fausse ; le langage C la considère donc comme une variable booléenne dont la valeur est **0 si l'expression est fausse** et un **nombre non nul si l'expression est vraie** (par exemple 1 mais aussi 2, 3 etc.).

Ceci nous permet d'écrire une boucle qui sera indéfiniment parcourue :

```
while (1) { }
```

Dans ce cas, l'expression qui vaut 1 n'a pas besoin d'être testée puisque le programme a déjà sa valeur qui est non nulle : l'expression est donc **considérée comme vraie** et la boucle (qui ne fait rien) est exécutée indéfiniment. En rajoutant simplement cette ligne à la fin de votre programme, vous le ferez boucler indéfiniment, ce qui veut dire que cela empêche la fonction loop d'être exécutée plusieurs fois en boucle.

À retenir sur les boucles :

- Une boucle permet de d'exécuter plusieurs fois la même série d'instructions.
- La boucle for.
- L'incrémentation et la décrémentation.
- La boucle while.
- La boucle do … while.
- La boucle sans fin bloquant l'exécution du programme.

Les tests conditionnels

Les tests conditionnels permettent d'introduire une certaine forme d'intelligence artificielle dans nos programmes puisque ceux-ci ne réaliseront pas les mêmes choses en fonctions de conditions extérieures. Par exemple, si un ILS situé en amont d'un passage à niveau est fermé par l'aimant d'une locomotive, alors il faut fermer les barrières du passage à niveau. La condition extérieure est l'approche d'un train, le résultat du test est la fermeture des barrières.

En « langage » d'Arduino, cela se traduit pas **if** qui signifie si en anglais :

```
if (condition)
        {
                // Ce qu'il faut faire
        }
```

Si la condition est vraie, alors faire ce qui est entre accolades. La condition peut tester l'état d'un interrupteur (poussoir, ILS, etc.) ou bien le résultat d'un calcul. On utilise pour cela des **opérateurs de comparaison** comme :

== (si égal à), != (si non égal à, ou encore si différent de), < (si inférieur), > (si supérieur), <=, >=.

Si la condition n'est pas vraie, les instructions entre accolades ne sont pas exécutées.

Notez bien le **double signe égal** pour tester une égalité :

```
if (x == 12)
        {
                // Faire ceci
        }
```

Si vous aviez mis if (x = 12), le programme aurait affecté la valeur 12 à la variable x ; comme celle-ci aurait été non-nulle, elle aurait été considérée comme VRAIE et les instructions entre accolades auraient été systématiquement exécutées (voir les variables booléennes au chapitre 9).

On peut aussi préciser ce qu'il faut faire si la condition n'est pas vraie et dans ce cas on utilise la structure de contrôle **if … else** (si … autrement).

```
if (condition)
        {
                // Action A à faire si condition est vraie
        }
else
        {
                // Action B à faire si condition est fausse
        }
```

Le else (autrement) peut d'ailleurs conduire à exécuter un autre test ; on obtient ainsi une structure du type **if … else if … else** ; le « else if » est sur une même ligne et est suivi d'une deuxième condition à tester.

```
if (condition 1)
        {
                // Faire ceci
        }
else if (condition 2)
        {
                // Faire cela
        }
else
        {
                // Faire autrement
        }
```

En termes de programmation, on parle de **branchement** car en fonction du résultat du test de notre condition (VRAI ou FAUX), le programme se branche à un endroit ou à un autre pour exécuter les instructions qui suivent.

Une autre façon de programmer un branchement est d'utiliser la structure **switch case**. Dans ce cas, on compare la valeur d'une variable (généralement un entier naturel) aux valeurs des différents cas à traiter.

```
switch (variable) {
        case 1 :
                // Faire ceci si variable vaut 1
                break ;
        case 2 :
                // Faire ceci si variable vaut 2
                break ;
        default :
                // Faire cela si variable ne vaut ni 1, ni 2 (le cas default est optionnel)
        break ;
}
```

L'instruction break est suivie de point-virgule alors que case ou default sont suivis de deux points. L'instruction break est **obligatoire pour sortir du switch** et ne pas exécuter les instructions situées au-dessous.

<u>À retenir sur les tests conditionnels</u> :

- Les tests conditionnels permettent de faire prendre des décisions à notre programme en fonctions de conditions.
- Les conditions peuvent être l'état d'un interrupteur ou d'un capteur, ou bien le résultat d'un calcul.
- Les opérateurs de comparaison tels que ==, !=, <, <=, etc.
- La structure if.
- La structure if … else.
- La structure switch case.
- L'instruction break à ne pas omettre.