Le langage Arduino (2/2)

diy arduino

Dernière mise à jour : lundi 27 juin 2016 à 16h32

Auteurs :  Eskimon  olyte

Catégorie : Matériel et électronique

Le langage Arduino (1/2) Notre premier programme !

> J'ai une question. Si je veux faire que le code que j'ai écrit se répète, je suis obligé de le recopier autant de fois que je veux ? Ou bien il existe une solution ?

Voilà une excellente question qui introduit le chapitre que vous allez commencer à lire car c'est justement l'objet de ce chapitre. Nous allons voir comment faire pour qu'un bout de code se répète. Puis nous verrons, ensuite, comment organiser notre code pour que celui-ci devienne plus lisible et facile à débugger. Enfin, nous apprendrons à utiliser les tableaux qui nous seront très utiles. Voilà le programme qui vous attend !

- Les boucles
- Les fonctions
- Les tableaux

Les boucles

> Qu'est-ce qu'une boucle ?

En programmation, une **boucle** est une instruction qui permet de répéter un bout de code. Cela va nous permettre de faire se répéter un bout de programme ou un programme entier. Il existe deux types principaux de boucles :

- La **boucle conditionnelle**, qui teste une condition et qui exécute les instructions qu'elle contient tant que la condition testée est vraie.
- La **boucle de répétition**, qui exécute les instructions qu'elle contient, un nombre de fois prédéterminé.

La boucle while

Problème : **Je veux que le volet électrique de ma fenêtre se ferme automatiquement quand la nuit tombe. Nous ne nous occuperons pas de faire le système qui ferme le volet à l'arrivée de la nuit. La carte Arduino dispose d'un capteur qui indique la position du volet (ouvert ou fermé). Ce que nous cherchons à faire : c'est créer un bout de code qui fait descendre le volet** tant qu'il n'est pas fermé. Pour résoudre le problème posé, il va falloir que l'on utilise une boucle.

```
1  /* ICI, un bout de programme permet de faire les choses suivantes :
2  1. un capteur détecte la tombée de la nuit et la levée du jour
3       - Si c'est la nuit, alors on doit fermer le volet
4       - Sinon, si c'est le jour, on doit ouvrir le volet
5  2. le programme lit l'état du capteur qui indique si le volet est ouvert ou fermé
6  3. enregistrement de cet état dans la variable de type String : position_volet
7       - Si le volet est ouvert, alors : position_volet = "ouvert";
8       - Sinon, si le volet est fermé : position_volet = "ferme";
9  */
10
11 while(position_volet == "ouvert")
12 {
13     // instructions qui font descendre le volet
14 }
```

La boucle while

Comment lire ce code ?

En anglais, le mot **while** signifie "tant que". Donc si on lit la ligne :

```
1  while(position_volet == "ouvert") {/* instructions */}
```

Il faut la lire : "TANT QUE la position du volet est **ouvert**", on boucle/répète les instructions de la boucle (entre les accolades).

Construction d'une boucle while

Voilà donc la syntaxe de cette boucle qu'il faut retenir :

```
while(/* condition à tester */)
{
    // les instructions entre ces accolades sont répétées
    // tant que la condition est vraie
}
```

Syntaxe de la boucle while

Un exemple

Prenons un exemple simple, réalisons un compteur !

```
// variable compteur qui va stocker le nombre de fois que la boucle
int compteur = 0;
// aura été exécutée

// tant que compteur est différent de 5, on boucle
while(compteur != 5)
{
    compteur++; // on incrémente la variable compteur à chaque tour de boucle
}
```

Un petit compteur

Si on teste ce code (dans la réalité rien ne s'affiche, c'est juste un exemple pour vous montrer), cela donne :

```
compteur = 0
compteur = 1
compteur = 2
compteur = 3
compteur = 4
compteur = 5
```

Résultat de notre compteur

Donc au départ, la variable **compteur** vaut 0, on exécute la boucle et on incrémente **compteur**. Mais

compteur ne vaut pour l'instant que 1, donc on ré-exécute la boucle. Maintenant **compteur** vaut 2. On répète la boucle, ... jusqu'à 5. Si **compteur** vaut 5, la boucle n'est pas ré-exécutée et on continu le programme. Dans notre cas, le programme se termine.

La boucle do...while

Cette boucle est similaire à la précédente. Mais il y a une différence qui a son importance ! En effet, si on prête attention à la place la condition dans la boucle **while**, on s'aperçoit qu'elle est testée avant de rentrer dans la boucle. Tandis que dans une boucle do...while, la condition est testée seulement lorsque le programme est rentré dans la boucle :

```
do
{
    // les instructions entre ces accolades sont répétées
    // TANT QUE la condition est vrai
}while(/* condition à tester */);
```

Syntaxe de la boucle do...while

> Le mot **do** vient de l'anglais et se traduis par **faire**. Donc la boucle do...while signifie "faire les instructions, tant que la condition testée est fausse". Tandis que dans une boucle while on pourrait dire : "tant que la condition est fausse, fais ce qui suit".

> Qu'est-ce que ça change ?

Et bien, dans une **while**, si la condition est fausse dès le départ, on entrera jamais dans cette boucle. A l'inverse, avec une boucle **do...while**, on entre dans la boucle puis on test la condition. Reprenons notre compteur :

```
// variable compteur = 5
int compteur = 5;

do
{
    compteur++;  // on incrémente la variable compteur à chaque tour de boucle
}while(compteur < 5);  // tant que compteur est inférieur à 5, on boucle
```

un compteur avec do...while

Dans ce code, on définit dès le départ la valeur de **compteur** à 5. Or, le programme va rentrer dans la boucle alors que la condition est fausse. Donc **la boucle est au moins exécutée une fois** ! Et ce quelle que soit la véracité de la condition. En test cela donne :

```
compteur = 6
```

Résultat de la boucle do...while

Concaténation

Une boucle est une instruction qui a été répartie sur plusieurs lignes. Mais on peut l'écrire sur une seule ligne :

```
// variable compteur = 5
int compteur = 5;

do{compteur++;}while(compteur < 5);
```

La boucle sur une seule ligne

> C'est pourquoi il ne faut pas oublier le point virgule à la fin (après le while). Alors que dans une simple boucle **while** le point virgule **ne doit pas** être mis !

La boucle for

Voilà une boucle bien particulière. Ce qu'elle va nous permettre de faire est assez simple. Cette boucle est exécutée X fois. Contrairement aux deux boucles précédentes, on doit lui donner trois paramètres.

```
for(int compteur = 0; compteur < 5; compteur++)
{
    // code à exécuter
}
```

la boucle for

Fonctionnement

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

D'abord, on crée la boucle avec le terme **for** (signifie "pour que"). Ensuite, entre les parenthèses, on doit donner trois **paramètres** qui sont :

- la création et l'assignation de la variable à une valeur de départ
- suivit de la définition de la condition à tester
- suivit de l'instruction à exécuter

Donc, si on lit cette ligne : "POUR compteur allant de 0 jusque 5, on incrémente compteur". De façon plus concise, la boucle est exécutée autant de fois qu'il sera nécessaire à **compteur** pour arriver à 5. Donc ici, le code qui se trouve à l'intérieur de la boucle sera exécuté 5 fois.

A retenir

La structure de la boucle :

```
for(/*initialisation de la variable*/ ; /*condition à laquelle la boucle s'arrête
```

syntaxe de la boucle for

La boucle infinie

La boucle infinie est très simple à réaliser, d'autant plus qu'elle est parfois très utile. Il suffit simplement d'utiliser une **while** et de lui assigner comme condition une valeur qui ne change jamais. En l'occurrence, on met souvent le chiffre 1.

```
while(1)
{
    // instructions à répéter jusqu'à l'infinie
}
```

La boucle infinie

On peut lire : "TANT QUE la condition est égale à 1, on exécute la boucle". Et cette condition sera toujours remplie puisque "1" n'est pas une variable mais bien un chiffre. Également, il est possible de mettre tout autre chiffre entier, ou bien le booléen "TRUE" :

```
while(TRUE)
{
    // instructions à répéter jusqu'à l'infinie
}
```

La boucle infinie avec un booléen

Cela ne fonctionnera pas avec la valeur **0**. En effet, 0 signifie "condition fausse" donc la boucle s'arrêtera aussitôt...

La fonction loop() se comporte comme une boucle infinie, puisqu'elle se répète après avoir fini d'exécuter ses tâches.

Les fonctions

Dans un programme, les lignes sont souvent très nombreuses. Il devient alors impératif de séparer le programme en petits bouts afin d'améliorer la lisibilité de celui-ci, en plus d'améliorer le fonctionnement et de faciliter le débogage. Nous allons voir ensemble ce qu'est une fonction, puis nous apprendrons à les créer et les appeler.

Qu'est-ce qu'une fonction ?

Une **fonction** est un "conteneur" mais différent des variables. En effet, une variable ne peut contenir qu'un nombre, tandis qu'une fonction peut contenir un programme entier ! Par exemple ce code est une fonction :

```
1 void setup()
2 {
3     // instructions
4 }
```

un exemple de fonction

En fait, lorsque l'on va programmer notre carte Arduino, on va écrire notre programme dans des fonctions. Pour l'instant nous n'en connaissons que 2 : `setup()` et `loop()`. Dans l'exemple précédent, à la place du commentaire, on peut mettre des instructions (conditions, boucles, variables, ...). Ce sont ces instructions qui vont constituer le programme en lui même. Pour être plus concret, une fonction est un bout de programme qui permet de réaliser une tâche bien précise. Par exemple, pour mettre en forme un texte, on peut colorier un mot en bleu, mettre le mot en gras ou encore grossir ce mot. A chaque fois, on a utilisé une fonction :

- **gras**, pour mettre le mot en gras
- **colorier**, pour mettre le mot en bleu
- **grossir**, pour augmenter la taille du mot

En programmation, on va utiliser des fonctions. Alors ces fonctions sont "réparties dans deux grandes

familles". Ce que j'entends par là, c'est qu'il existe des fonctions toutes prêtes dans le langage Arduino et d'autres que l'on va devoir créer nous même. C'est ce dernier point qui va nous intéresser.

> On ne peut pas écrire un programme sans mettre de fonctions à l'intérieur ! On est obligé d'utiliser la fonction **setup**() et **loop**() (même si on ne met rien dedans). Si vous écrivez des instructions en dehors d'une fonction, le logiciel Arduino refusera systématiquement de compiler votre programme. Il n'y a que les variables globales que vous pourrez déclarer en dehors des fonctions.

> J'ai pas trop compris à quoi ça sert ?

L'utilité d'une fonction réside dans sa capacité à simplifier le code et à le séparer en "petits bouts" que l'on assemblera ensemble pour créer le programme final Si vous voulez, c'est un peu comme les jeux de construction en plastique : chaque pièce à son propre mécanisme et réalise une fonction. Par exemple une roue permet de rouler ; un bloc permet de réunir plusieurs autres blocs entre eux ; un moteur va faire avancer l'objet créé... Et bien tous ces éléments seront assemblés entre eux pour former un objet (voiture, maison, ...). Tout comme, les fonctions seront assemblées entre elles pour former un programme. On aura par exemple la fonction : "mettre au carré un nombre" ; la fonction : "additionner a + b" ; etc. Qui au final donnera le résultat souhaité.

Fabriquer une fonction

Pour fabriquer une fonction, nous avons besoin de savoir trois choses :

- Quel est le **type** de la fonction que je souhaite créer ?
- Quel sera son **nom** ?
- Quel(s) **paramètre(s)** prendra-t-elle ?

Nom de la fonction

Pour commencer, nous allons, en premier lieu, choisir le nom de la fonction. Par exemple, si votre fonction doit récupérer la température d'une pièce fournie par un capteur de température : vous appellerez la fonction **lireTemperaturePiece**, ou bien **lire_temperature_piece**, ou encore **lecture_temp_piece**. Bon, des noms on peut lui en donner plein, mais soyez logique quant au choix de ce dernier. Ce sera plus facile pour comprendre le code que si vous l'appelez **tmp** (pour température).

> Un nom de fonction explicite garantit une lecture rapide et une compréhension aisée du code. Un lecteur doit savoir ce que fait la fonction juste grâce à son nom, sans lire le contenu !

Les types et les paramètres

Les fonctions ont pour but de découper votre programme en différentes unités logiques. Idéalement, le programme principal ne devrait utiliser que des appels de fonctions, en faisant un minimum de traitement. Afin de pouvoir fonctionner, elles utilisent, la plupart du temps, des "choses" en **entrées** et renvoient "quelque chose" en **sortie**. Les entrées seront appelées des **paramètres de la fonction** et la sortie sera appelée **valeur de retour**.

> Notez qu'une fonction ne peut renvoyer qu'un seul résultat à la fois. Notez également qu'une fonction ne renvoie pas obligatoirement un résultat. Elle n'est pas non plus obligée d'utiliser des paramètres.

Les paramètres

Les paramètres servent à nourrir votre fonction. Ils servent à donner des informations au traitement qu'elle doit effectuer. Prenons un exemple concret. Pour changer l'état d'une sortie du microcontrôleur, Arduino nous propose la fonction suivante: [digitalWrite(pin, value)]http://arduino.cc/en/Reference/DigitalWrite). Ainsi, la référence nous explique que la fonction a les caractéristiques suivantes:

- paramètre **pin**: le numéro de la broche à changer
- paramètre **value**: l'état dans lequel mettre la broche (HIGH, (haut, +5V) ou LOW (bas, masse))
- retour: pas de retour de résultat

Comme vous pouvez le constater, l'exemple est explicite sans lire le code de la fonction. Son nom, digitalWrite ("écriture numérique" pour les anglophobes), signifie qu'on va changer l'état d'une broche **numérique** (donc pas analogique). Ses paramètres ont eux aussi des noms explicites, **pin** pour la broche à changer et **value** pour l'état à lui donner. Lorsque vous aller créer des fonctions, c'est à vous de voir si elles ont besoin de paramètres ou non. Par exemple, vous voulez faire une fonction qui met en pause votre programme, vous pouvez faire une fonction `Pause()` et déterminera la durée pendant laquelle le programme sera en pause. On obtiendra donc, par exemple, la syntaxe suivante : `void Pause(char duree)`. Pour résumer un peu, on a le choix de créer des **fonctions vides**, donc sans paramètres, ou bien des **fonctions "typées"** qui acceptent un ou plusieurs paramètres.

> Mais c'est quoi ça "void" ?

J'y arrive ! Souvenez vous, un peu plus haut je vous expliquais qu'une fonction pouvait retourner une valeur, la fameuse valeur de sortie, je vais maintenant vous expliquer son fonctionnement.

Le type void

On vient de voir qu'une fonction pouvait accepter des paramètres et éventuellement renvoyer quelque chose. Mais ce n'est pas obligatoire. En effet, si l'on reprend notre fonction "Pause", elle ne renvoie rien car ce n'est pas nécessaire de signaler quoi que ce soit. Dans ce cas, on préfixera le nom de notre

fonction avec le mot-clé "void". La syntaxe utilisée est la suivante :

```
void nom_de_la_fonction()
{
    // instructions
}
```

une fonction sans valeur de retour

On utilise donc le type `void` pour dire que la fonction n'aura pas de retour. Une fonction de type void ne peut donc pas retourner de valeur. Par exemple :

```
void fonction()
{
    int var = 24;
    return var; // ne fonctionnera pas car la fonction est de type void
}
```

Impossible pour une fonction void de retourner un entier

Ce code ne fonctionnera pas, parce que la fonction `int`. Ce qui est impossible ! Le compilateur le refusera et votre code final ne sera pas généré. Vous connaissez d'ailleurs déjà au moins deux fonctions qui n'ont pas de retour... Et oui, la fonction "setup" et la fonction "loop"  . Il n'y en a pas plus à savoir. 

Les fonctions "typées"

Là, cela devient légèrement plus intéressant. En effet, si on veut créer une fonction qui calcule le résultat d'une addition de deux nombres (ou un calcul plus complexe), il serait bien de pouvoir renvoyer directement le résultat plutôt que de le stocker dans une variable qui a une portée globale et d'accéder à cette variable dans une autre fonction. En clair, l'appel de la fonction nous donne directement le résultat. On peut alors faire "ce que l'on veut" avec ce résultat (le stocker dans une variable, l'utiliser dans une fonction, lui faire subir une opération, ...)

Comment créer une fonction typée ?

En soit, cela n'a rien de compliqué, il faut simplement remplacer `long`, ...) Voilà un exemple :

```
int maFonction()
{
```

```
    int resultat = 44;  // déclaration de ma variable résultat
    return resultat;
}

```

Une fonction typée "entier"

Notez que je n'ai pas mis les deux fonctions principales, à savoir `loop()`, mais elles sont obligatoires ! Lorsqu'elle sera appelée, la fonction `resultat`. Voyez cet exemple :

```
int calcul = 0;

void loop()
{
    calcul = 10 * maFonction();
}

int maFonction()
{
    int resultat = 44;  // déclaration de ma variable résultat
    return resultat;
}
```

Appel d'une fonction typée

Dans la fonction `calcul = 10 * 44;` Ce qui nous donne : calcul = 440. Bon ce n'est qu'un exemple très simple pour vous montrer le fonctionnement. Plus tard, lorsque vous serez au point, vous utiliserez certainement cette combinaison de façon plus complexe.

> Comme cet exemple est très simple, je n'ai pas inscrit la valeur retournée par la fonction `maFonction()` dans une variable, mais il est préférable de le faire. Du moins, lorsque c'est utile, ce qui n'est pas le cas ici.

Les fonctions avec paramètres

C'est bien gentil tout ça, mais maintenant vous allez voir quelque chose de bien plus intéressant. Voilà un code, nous verrons ce qu'il fait après :

```
int x = 64;
int y = 192;
```

```
3
4  void loop()
5  {
6      maFonction(x, y);
7  }
8
9  int maFonction(int param1, int param2)
10 {
11     int somme = 0;
12     somme = param1 + param2;
13     // somme = 64 + 192 = 255
14
15     return somme;
16 }
```

Une fonction avec deux paramètres

Que se passe-t-il ?

J'ai défini trois variables : `maFonction()` est "typée" et accepte des **paramètres**. Lisons le code du début :

- On déclare nos variables
- La fonction `maFonction()` que l'on a créée

C'est sur ce dernier point que l'on va se pencher. En effet, on a donné à la fonction des paramètres. Ces paramètres servent à "nourrir" la fonction. Pour faire simple, on dit à la fonction : "**Voilà deux paramètres, je veux que tu t'en serves pour faire le calcul que je veux**" Ensuite arrive le prototype de la fonction.

Le prototype... de quoi tu parles ?

Le prototype c'est le "titre complet" de la fonction. Grâce à elle on connait le **nom** de la fonction, le **type** de la valeur retourné, et le type des différents **paramètres**.

```
1  int maFonction(int param1, int param2)
```

le prototype de la fonction.

La fonction récupère dans des variables les paramètres que l'on lui a envoyés. Autrement dit, dans la variable `y`. Soit : `param2 = y = 192`. Pour finir, on utilise ces deux variables créées "à la volée" dans le prototype de la fonction pour réaliser le calcul souhaité (une somme dans notre cas).

> On parle aussi parfois de signature, pour désigner le nom et les paramètres de la fonction.

> A quoi ça sert de faire tout ça ? Pourquoi on utilise pas simplement les variables x et y dans la fonction ?

Cela va nous servir à simplifier notre code. Mais pas seulement ! Par exemple, vous voulez faire plusieurs opérations différentes (addition, soustraction, etc.) et bien au lieu de créer plusieurs fonctions, on ne va en créer qu'une qui les fait toutes ! Mais, afin de lui dire quelle opération faire, vous lui donnerez un paramètre lui disant : "**Multiplie ces deux nombres**" ou bien "**additionne ces deux nombres**". Ce que cela donnerait :

```
1  unsigned char operation = 0;
2  int x = 5;
3  int y = 10;
4
5  void loop()
6  {
7      // le paramètre "opération" donne le type d'opération à faire
8      maFonction(x, y, operation);
9  }
10
11 int maFonction(int param1, int param2, int param3)
12 {
13     int resultat = 0;
14     switch(param3)
15     {
16     case 0 : // addition, resultat = 15
17         resultat = param1 + param2;
18         break;
19     case 1 : // soustraction, resultat = -5
20         resultat = param1 - param2;
21         break;
22     case 2 : // multiplication, resultat = 50
23         resultat = param1 * param2;
24         break;
25     case 3 : // division, resultat = 0 (car nombre entier)
```

```
        resultat = param1 / param2;
        break;
    default :
        resultat = 0;
        break;
    }

    return resultat;
}

```

Une fonction générique

Les tableaux

Comme son nom l'indique, cette partie va parler des tableaux.

Quel est l'intérêt de parler de cette surface ennuyeuse qu'utilisent nos chers enseignants ?

Eh bien détrompez-vous, en informatique un tableau ça n'a rien à voir ! Si on devait (beaucoup) résumer, un tableau est une grosse variable. Son but est de **stocker des éléments de mêmes types en les mettant dans des cases**. Par exemple, un prof qui stocke les notes de ses élèves. Il utilisera un tableau de `float` (nombre à virgule), avec une case par élèves. Nous allons utiliser cet exemple tout au long de cette partie. Voici quelques précisions pour bien tout comprendre :

- chaque élève sera identifié par un numéro allant de 0 (le premier élève) à 19 (le vingtième élève de la classe)
- on part de 0 car en informatique la première valeur dans un tableau est 0 !

Un tableau en programmation

Un tableau, tout comme sous Excel, c'est un ensemble constitué de cases, lesquels vont contenir des informations. En programmation, ces informations seront des **nombres**. Chaque case d'un tableau contiendra une valeur. En reprenant l'exemple des notes des élèves, le tableau répertoriant les notes de chaque élève ressemblerait à ceci :

élève 0	élève 1	élève 2	[...]	élève n-1	élève n	
10	15,5	8	[...]	18	7	

Table: Un tableau en informatique

A quoi ça sert ?

On va principalement utiliser des tableaux lorsque l'on aura besoin de stocker des informations sans pour autant créer une variable pour chaque information. Toujours avec le même exemple, au lieu de créer une variable `eleve2` et ainsi de suite pour chaque élève, on inscrit les notes des élèves dans un tableau.

> Mais, concrètement c'est quoi un tableau : une variable ? une fonction ?

Ni l'un, ni l'autre. En fait, on pourrait comparer cela avec un index qui pointe vers les valeurs de variables qui sont contenus dans chaque case du tableau. Un petit schéma pour simplifier :

élève 0	**élève 1**
variable dont on ne connaît pas le nom mais qui stocke une valeur	idem, mais variable différente de la case précédente

Par exemple, cela donnerait :

élève 0	**élève 1**
variable `note_eleve0`	variable `note_eleve1`

Avec notre exemple :

élève 0	**élève 1**
10	15,5

Soit, lorsque l'on demandera la valeur de la case 1 (correspondant à la note de l'élève 1), le tableau nous renverra le nombre : 15,5. Alors, dans un premier temps, on va voir comment déclarer un tableau et l'initialiser. Vous verrez qu'il y a différentes manières de procéder. Après, on finira par apprendre comment utiliser un tableau et aller chercher des valeurs dans celui-ci. Et pour finir, on terminera ce chapitre par un exemple. Il y a encore du boulot ! 

Déclarer un tableau

Comme expliqué plus tôt, un tableau contient des éléments de même type. On le déclare donc avec un type semblable, et une taille représentant le nombre d'éléments qu'il contiendra. Par exemple, pour notre classe de 20 étudiants :

```
1 float notes[20];
```

déclarer un tableau de 20 cases

On veut stocker des notes, donc des valeurs décimales entre 0 et 20. On va donc créer un tableau de float (car c'est le type de variable qui accepte les nombres à virgule, souvenez-vous !). Dans cette classe, il y a 20 élèves (de 0 à 19) donc le tableau contiendra 20 éléments. Si on voulait faire un tableau de 100 étudiants dans lesquels on recense leurs nombres d'absence, on ferait le tableau suivant:

```
1 char absenteisme[100];
```

un tableau de `char`

Accéder et modifier une case du tableau

Pour accéder à une case d'un tableau, il suffit de connaître **l'indice** de la case à laquelle on veut accéder. L'indice c'est le numéro de la case qu'on veut lire/écrire. Par exemple, pour lire la valeur de la case 10 (donc indice 9 car on commence à 0):

```
1 float notes[20]; // notre tableau
2 float valeur; // une variable qui contiendra une note
3
4 // valeur contient désormais la note du dixième élève
5 valeur = notes[9];
```

Accéder à une valeur

Ce code se traduit par l'enregistrement de la valeur contenue dans la dixième case du tableau, dans une variable nommée `valeur`. A présent, si on veut aller modifier cette même valeur, on fait comme avec une variable normale, il suffit d'utiliser l'opérateur ' = ' :

```
1 notes[9] = 10,5;   // on change la note du dixième élève
```

Modifier une valeur

En fait, on procède de la même manière que pour changer la valeur d'une variable, car, je vous l'ai dit, chaque case d'un tableau est une variable qui contient une valeur ou non.

> Faites attention aux indices utilisés. Si vous essayez de lire/écrire dans une case de tableau trop loin (indice trop grand, par exemple 987362598412), le comportement pourrait devenir imprévisible.

Car en pratique vous modifierez des valeurs qui seront peut-être utilisées par le système pour autre chose. Ce qui pourrait avoir de graves conséquences !

> Vous avez sûrement rencontré des crashs de programme sur votre ordinateur, ils sont souvent dû à la modification de variable qui n'appartiennent pas au programme, donc l'OS "tue" ce programme qui essai de manipuler des trucs qui ne lui appartiennent pas.

Initialiser un tableau

Au départ, notre tableau était vide :

```
1 float notes[20];
2 // on créer un tableau dont le contenu est vide, on sait simplement qu'il contien
```

Ce que l'on va faire, c'est **initialiser** notre tableau. On a la possibilité de remplir chaque case une par une ou bien utiliser une boucle qui remplira le tableau à notre place. Dans le premier cas, on peut mettre la valeur que l'on veut dans chaque case du tableau, tandis qu'avec la deuxième solution, on remplira les cases du tableau avec la même valeur, bien que l'on puisse le remplir avec des valeur différentes mais c'est un peu plus compliqué. Dans notre exemple des notes, on part du principe que 6l'examen n'est pas passé, donc tout le monde à 0. Pour cela, on parcourt toutes les cases en leur mettant la valeur 0 :

```
1  char i=0; // une variable que l'on va incrémenter
2  float notes[20];  // notre tableau
3
4  void setup()
5  {
6      // boucle for qui remplira le tableau pour nous
7      for(i = 0; i < 20; i++)
8      {
9          notes[i] = 0; // chaque case du tableau vaudra 0
10     }
11 }
```

Initialisation d'un tableau

L'initialisation d'un tableau peut se faire directement lors de sa création, comme ceci :

```
float note[] = {0,0,0,0 /*, etc.*/ };
// Le tableau aura alors autant de case que de nombre passé en paramètres
```

Initialisation à la déclaration

Exemple de traitement

Bon c'est bien beau tout ça, on a des notes coincées dans un tableau, on en fait quoi ? :roll:

Excellente question, et ça dépendra de l'usage que vous en aurez ! Voyons des cas d'utilisations pour notre tableau de notes (en utilisant des fonctions).

La note maximale

Comme le titre l'indique, on va rechercher la note maximale (le meilleur élève de la classe). La fonction recevra en paramètre le tableau de float, le nombre d'éléments dans ce tableau et renverra la meilleure note.

```
float meilleurNote(float tableau[], int nombreEleve)
{
    int i = 0;
    int max = 0; // variable contenant la future meilleure note

    for(i=0; i<nombreEleve, i++)
    {
        // si la note lue est meilleure que la meilleure actuelle
        if(tableau[i] > max)
        {
            // alors on l'enregistre
            max = tableau[i];
        }
    }
    // on retourne la meilleure note
    return max;
}
```

Recherche de la note maximale

Ce que l'on fait, pour lire un tableau, est exactement la même chose que lorsqu'on l'initialise avec une boucle `for`.

Il est tout à fait possible de mettre la valeur de la case recherché dans une variable :

```
1 int valeur = tableau[5]; // on enregistre la valeur de la case 6 du tableau dans
```

Voila, ce n'était pas si dur, vous pouvez faire pareil pour chercher la valeur minimale afin vous entrainer !

Calcul de moyenne

Ici, on va chercher la moyenne des notes. La signature de la fonction sera exactement la même que celle de la fonction précédente, à la différence du nom ! Je vous laisse réfléchir, voici la signature de la fonction, le code est plus bas mais essayez de le trouver vous-même avant :

```
1 float moyenneNote(float tableau[], int nombreEleve)
```

Une solution :

Afficher/Masquer le contenu masqué

On en termine avec les tableaux, on verra peut être plus de choses en pratique. 

Maintenant vous pouvez pleurer, de joie bien sûr, car vous venez de terminer la première partie ! A présent, faisons place à la pratique...

Le langage Arduino (1/2) Notre premier programme !

mmaire