Ce site utilise Google Analytics. En poursuivant votre navigation sur ce site, vous nous autorisez à déposer des cookies à des fins de mesure d'audience. Pour s'opposer à ce dépôt vous pouvez cliquer ici.

En savoir plus OK

Le langage Arduino (1/2)

Licence CC BY-NC-SA

diy arduino

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

Auteurs : Eskimon olyte

Catégorie : Matériel et électronique

Le matériel Le langage Arduino (2/2)

A présent que vous avez une vision globale sur le fonctionnement de la carte Arduino, nous allons pouvoir apprendre à programmer avant de nous lancer dans la réalisation de programmes très simples pour débuter ! Pour pouvoir programmer notre carte, il nous faut trois choses :

- Un ordinateur
- Une carte Arduino
- Et connaitre le langage Arduino

C'est ce dernier point qu'il nous faut acquérir. Le but même de ce chapitre est de vous apprendre à programmer avec le langage Arduino. Cependant, ce n'est qu'un support de cours que vous pourrez parcourir lorsque vous devrez programmer tout seul votre carte. En effet, c'est en manipulant que l'on apprend, ce qui implique que votre apprentissage en programmation sera plus conséquent dans les prochains chapitres que dans ce cours même.

> Le langage Arduino est très proche du C et du C++. Pour ceux dont la connaissance de ces langages est fondée, ne vous sentez pas obligé de lire les deux chapitres sur le langage Arduino. Bien qu'il y ait des points quelques peu importants.

- La syntaxe du langage
- Les variables
- Les conditions

La syntaxe du langage

La syntaxe d'un langage de programmation est l'ensemble des règles d'écriture liées à ce langage. On va donc voir dans ce sous-chapitre les règles qui régissent l'écriture du langage Arduino.

Le code minimal

Avec Arduino, nous devons utiliser un code minimal lorsque l'on crée un programme. Ce code permet de diviser le programme que nous allons créer en deux grosses parties.

```
1  // fonction d'initialisation de la carte
2  void setup()
3  {
4      // contenu de l'initialisation
5  }
6
7  // fonction principale, elle se répète (s'exécute) à l'infini
8  void loop()
9  {
10     // contenu de votre programme
11 }
```

Le code minimal d'un programme Arduino

Vous avez donc devant vous le code minimal qu'il faut insérer dans votre programme. Mais que peut-il bien signifier pour quelqu'un qui n'a jamais programmé ?

La fonction `setup`

Dans ce code se trouvent deux fonctions. Les fonctions sont en fait des portions de code.

```
1  // fonction d'initialisation de la carte
2  void setup()
3  {
4      // contenu de l'initialisation
5      // on écrit le code à l'intérieur
```

```
}
```

Zoom sur la fonction setup

Cette fonction **setup()** est appelée une seule fois lorsque le programme commence. C'est pourquoi c'est dans cette fonction que l'on va écrire le code qui n'a besoin d'être exécuté une seule fois. On appelle cette fonction : **fonction d'initialisation**. On y retrouvera la mise en place des différentes sorties et quelques autres réglages. C'est un peu le check-up de démarrage. Imaginez un pilote d'avion dans sa cabine qui fait l'inventaire : - patte 2 en sortie, état haut ? - OK - timer 3 à 15 millisecondes ? - OK ...

Une fois que l'on a initialisé le programme il faut ensuite créer son "cœur", autrement dit le programme en lui même.

```
// fonction principale, elle se répète (s'exécute) à l'infini
void loop()
{
    // contenu de votre programme
}
```

Zoom sur la fonction principale

C'est donc dans cette fonction **loop()** où l'on va écrire le contenu du programme. Il faut savoir que cette fonction est appelée en permanence, c'est-à-dire qu'elle est exécutée une fois, puis lorsque son exécution est terminée, on la ré-exécute et encore et encore. On parle de **boucle infinie**.

> A titre informatif, on n'est pas obligé d'écrire quelque chose dans ces deux fonctions. En revanche, il est **obligatoire** de les écrire, même si elles ne contiennent aucun code !

Les instructions

> Dans ces fonctions, on écrit quoi ?

C'est justement l'objet de ce paragraphe. Dans votre liste pour le diner de ce soir, vous écrivez les tâches importantes qui vous attendent. Ce sont des **instructions**. Les instructions sont des lignes de code qui disent au programme : "fait ceci, fait cela, ..." C'est tout bête mais très puissant car c'est ce qui va orchestrer notre programme.

Les points virgules

Les points virgules terminent les instructions. Si par exemple je dis dans mon programme : "appelle la fonction couperDuSaucisson" je dois mettre un point virgule après l'appel de cette fonction.

> Les points virgules (;) sont synonymes d'erreurs car il arrive très souvent de les oublier à la fin des instructions. Par conséquent le code ne marche pas et la recherche de l'erreur peut nous prendre un temps conséquent ! Donc faites bien attention.

Les accolades

Les accolades sont les "conteneurs" du code du programme. Elles sont propres aux fonctions, aux conditions et aux boucles. Les instructions du programme sont écrites à l'intérieur de ces accolades.

Parfois elles ne sont pas obligatoires dans les conditions (nous allons voir plus bas ce que c'est), mais je recommande de les **mettre tout le temps** ! Cela rendra plus lisible votre programme.

Les commentaires

Pour finir, on va voir ce qu'est un commentaire. J'en ai déjà mis dans les exemples de codes. Ce sont des lignes de codes qui seront ignorées par le programme. Elles ne servent en rien lors de l'exécution du programme.

> Mais alors c'est inutile ?

Non car cela va nous permettre à nous et aux programmeurs qui lirons votre code (s'il y en a) de savoir ce que signifie la ligne de code que vous avez écrite. C'est très important de mettre des commentaires et cela permet aussi de reprendre un programme laissé dans l'oubli plus facilement ! Si par exemple vous connaissez mal une instruction que vous avez écrite dans votre programme, vous mettez une ligne de commentaire pour vous rappeler la prochaine fois que vous lirez votre programme ce que la ligne signifie.

```
1 // cette ligne est un commentaire sur UNE SEULE ligne
```

Ligne unique de commentaire

```
1 /*cette ligne est un commentaire, sur PLUSIEURS lignes
2 qui sera ignoré par le programme, mais pas par celui qui lit le code */
```

Commentaire sur plusieurs lignes

Les accents

> Il est formellement interdit de mettre des accents en programmation. Sauf dans les commentaires.

Les variables

Nous l'avons vu, dans un microcontrôleur, il y a plusieurs types de mémoire. Nous nous occuperons seulement de la mémoire "vive" (RAM) et de la mémoire "morte" (EEPROM). Je vais vous poser une énigme. Imaginons que vous avez connecté un bouton poussoir sur une broche de votre carte Arduino. Comment allez-vous stocker l'état du bouton (appuyé ou éteint) ?

Une variable, qu'est ce que c'est ?

Une **variable est un nombre**. Ce nombre est stocké dans un espace de la mémoire vive (RAM) du microcontrôleur. La manière qui permet de les stocker est semblable à celle utilisée pour ranger des chaussures : dans un casier numéroté.

Chaussures rangées dans des cases numérotées									
1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60

Table: Un tableau est un ensemble de case

> Une variable est un nombre, c'est tout ?

Ce nombre a la particularité de changer de valeur. Étrange n'est-ce pas ? Et bien pas tant que ça, car une variable est en fait le **conteneur** du nombre en question. Et ce conteneur va être stocké dans une case de la mémoire. Si on matérialise cette explication par un schéma, cela donnerait :

nombre => variable => mémoire

- le symbole "=>" signifiant : "est contenu dans..."

Le nom d'une variable

Le nom de variable accepte quasiment tous les caractères sauf :

- `.` (le point)
- `,` (la virgule)
- `é`, `à`, `ç`, `è` (les accents)

Bon je vais pas tous les donner, il n'accepte que l'alphabet alphanumérique ([a-z], [A-Z], [0-9]) et _ (underscore). Il ne doit **pas** commencer par un chiffre.

Définir une variable

Si on donne un nombre à notre programme, il ne sait pas si c'est une variable ou pas. Il faut le lui indiquer. Pour cela, on donne un **type** aux variables. Oui, car il existe plusieurs types de variables ! Par exemple la variable "x" vaut 4 :

```
1  x = 4;
```

Assignation à une variable

Et bien ce code ne fonctionnerait pas car il ne suffit pas ! En effet, il existe une multitude de nombres : les nombres entiers, les nombres décimaux, ... C'est pour cela qu'il faut assigner une variable à un type. Voilà les types de variables les plus répandus :

Type	Quel nombre il stocke ?	Valeurs maximales du nombre stocké	Nombre sur X bits	Nombre d'octets
int	entier	-32 768 **à** +32 767	16 bits	2 octets
long	entier	-2 147 483 648 **à** +2 147 483 647	32 bits	4 octets
char	entier	-128 **à** +127	8 bits	1 octets
float	décimale	-3.4×10^{38} **à** $+3.4 \times 10^{38}$	32 bits	4 octets
double	décimale	-3.4×10^{38} **à** $+3.4 \times 10^{38}$	32 bits	4 octets

Table: Les types de variables

Par exemple, si notre variable "x" ne prend que des valeurs entières, on utilisera les types **int**, **long**, ou **char**. Si maintenant la variable "x" ne dépasse pas la valeur 64 ou 87, alors on utilisera le type **char**.

```
1  char x = 0;
```

Assignation avec un type

> Si en revanche x = 260, alors on utilisera le type supérieur (qui accepte une plus grande quantité de nombre) à **char**, autrement dit **int** ou **long**.

> Mais t'es pas malin, pour éviter les dépassements de valeur ont met tout dans des double ou long !

Oui, mais NON. Un microcontrôleur, ce n'est pas un ordinateur 2GHz multicore, 4Go de RAM ! Ici on parle d'un système qui fonctionne avec un CPU à 16MHz (soit 0,016 GHz) et 2 Ko de SRAM pour la mémoire vive. Donc deux raisons font qu'il faut choisir ses variables de manière judicieuse :

- La RAM n'est pas extensible, quand y en a plus, il y en a plus !
- Le processeur est de type 8 bits (sur Arduino UNO), donc il est optimisé pour faire des traitements sur des variables de taille 8 bits, un traitement sur une variable 32 bits prendra donc (beaucoup) plus de temps !

Si à présent notre variable "x" ne prend jamais une valeur négative (-20, -78, ...), alors on utilisera un type **non-signé**. C'est à dire, dans notre cas, un **char** dont la valeur n'est plus de -128 à +127, mais de 0 à 255. Voici le tableau des types non signés, on repère ces types par le mot **unsigned** (de l'anglais : non-signé) qui les précède :

Type	Quel nombre il stocke ?	Valeurs maximales du nombre stocké	Nombre sur X bits	Nombre d'octets
unsigned char	entier non négatif	0 à 255	8 bits	1 octet
unsigned int	entier non négatif	0 à 65 535	16 bits	2 octets
unsigned long	entier non négatif	0 à 4 294 967 295	32 bits	4 octets

Table: Les types non signés

Une des particularités du langage Arduino est qu'il accepte un nombre plus important de types de variables. Je vous les liste dans ce tableau :

Type	Quel nombre il stocke ?	Valeurs maximales du nombre stocké	Nombre sur X bits	Nombre d'octets
byte	entier non négatif	0 à 255	8 bits	1 octet
word	entier non négatif	0 à 65535	16 bits	2 octets
boolean	entier non négatif	0 à 1	1 bit	1 octet

Table: Les types propres à Arduino

Pour votre information, vous pouvez retrouver ces tableaux sur cette page.

Les variables booléennes

Les variables **booléennes** sont des variables qui ne peuvent prendre que deux valeurs : ou VRAI ou FAUX. Elles sont utilisées notamment dans les boucles et les conditions. Nous verrons pourquoi. Une variable booléenne peut être définie de plusieurs manières :

```
1 // variable est fausse car elle vaut FALSE, du terme anglais "faux"
2 boolean variable = FALSE;
3 // variable est vraie car elle vaut TRUE, du terme anglais "vrai"
4 boolean variable = TRUE;
```

Les variables booléennes

Quand une variable vaut "0", on peut considérer cette variable comme une variable booléenne, elle est donc fausse. En revanche, lorsqu'elle vaut "1" ou n'importe quelle valeurs différente de zéro, on peut aussi la considérer comme une variable booléenne, elle est donc vraie. Voilà un exemple :

```
1 // variable est fausse car elle vaut 0
2 int variable = 0;
3 // variable est vraie car elle vaut 1
4 int variable = 1;
5 // variable est vraie car sa valeur est différente de 0
6 int variable = 42;
```

Un booléen codé avec des entiers

Le langage Arduino accepte aussi une troisième forme d'écriture (qui lui sert pour utiliser les broches de sorties du microcontrôleur) :

```
1 // variable est à l'état logique bas (= traduction de "low"), donc 0
2 int variable = LOW;
3 // variable est à l'état logique haut (= traduction de "high"), donc 1
4 int variable = HIGH;
```

l'utilisation des constantes LOW et HIGH

Nous nous servirons de cette troisième écriture pour allumer et éteindre des lumières...

Les opérations "simples"

On va voir à présent les opérations qui sont possibles avec le langage Arduino (addition, multiplication, ...).

Je vous vois tout de suite dire : "Mais pourquoi on fait ça, on l'a fait en primaire ! " Et bien parce que c'est quelque chose d'essentiel, car on pourra ensuite faire des opérations avec des variables. Vous verrez, vous changerez d'avis après avoir lu la suite !

L'addition

Vous savez ce que c'est, pas besoin d'explications. Voyons comment on fait cette opération avec le langage Arduino. Prenons la même variable que tout à l'heure :

```
1 // définition de la variable x
2 int x = 0;
3
4 // on change la valeur de x par une opération simple
5 x = 12 + 3;
6 // x vaut maintenant 12 + 3 = 15
```

L'addition

Faisons maintenant une addition de variables :

```
1 // définition de la variable x et assignation à la valeur 38
2 int x = 38;
3 int y = 10;
4 int z = 0;
5 // faisons une addition
6 // on a donc z = 38 + 10 = 48
7 z = x + y;
```

Addition de deux variables

La soustraction

On peut reprendre les exemples précédents, en faisant une soustraction :

```
 /définition de la variable x
int x = 0;

// on change la valeur de x par une opération simple
x = 12 - 3;
// x vaut maintenant 12 - 3 = 9
```

La soustraction

```
int x = 38;   // définition de la variable x et assignation à la valeur 38
int y = 10;
int z = 0;

z = x - y;    // on a donc z = 38 - 10 = 28
```

Soustraction de deux variables

La multiplication

```
int x = 0;
int y = 10;
int z = 0;

x = 12 * 3; // x vaut maintenant 12 * 3 = 36

z = x * y; // on a donc z = 36 * 10 = 360

// on peut aussi multiplier (ou une autre opération) un nombre et une variable :

z = z * ( 1 / 10 ); // soit z = 360 * 0.1 = 36
```

la multiplication

La division

```
float x = 0;
float y = 15;
float z = 0;

```

```
x = 12 / 2; // x vaut maintenant 12 / 2 = 6

z = y / x; // on a donc z = 15 / 6 = 2.5

```

La division

Attention cependant, si vous essayer de stocker le résultat d'une division dans une variable de type char, int ou long, le résultat sera stocké sous la forme d'un entier arrondi au nombre inférieur. Par exemple dans le code précédent si on met z dans un int on aura :

```
float x = 0;
float y = 15;
int z = 0;

x = 12 / 2; // x vaut maintenant 12 / 2 = 6

z = y / x; // on a donc z = 15 / 6 = 2 !
```

Arrondi au nombre inférieur

Le modulo

Après cette brève explication sur les opérations de base, passons à quelque chose de plus sérieux. Le modulo est une opération de base, certes moins connue que les autres. Cette opération permet d'obtenir le reste d'une division.

```
18 % 6 // le reste de l'opération est 0, car il y a 3*6 dans 18 donc 18 - 18 = 0
18 % 5 // le reste de l'opération est 3, car il y a 3*5 dans 18 donc 18 - 15 = 3
```

Le modulo

Le modulo est utilisé grâce au symbole %. C'est tout ce qu'il faut retenir. Autre exemple :

```
int x = 24;
int y = 6;
int z = 0;

z = x % y; // on a donc z = 24 % 6 = 0 (car 6 * 4 = 24)
```

le modulo entre deux variables

Le modulo ne peut-être fait que sur des nombres entiers

Quelques opérations bien pratiques

Voyons un peu d'autres opérations qui facilitent parfois l'écriture du code.

L'incrémentation

Derrière ce nom barbare se cache une simple opération d'addition.

```
1 var = 0;
2 var++; // c'est cette ligne de code qui nous intéresse
```

Incrémenter

"var++;" revient à écrire : "var = var + 1;" En fait, on ajoute le chiffre 1 à la valeur de var. Et si on répète le code un certain nombre de fois, par exemple 30, et bien on aura var = 30.

La décrémentation

C'est l'inverse de l'incrémentation. Autrement dit, on enlève le chiffre 1 à la valeur de var.

```
1 var = 30;
2 var--; // décrémentation de var
```

La décrémentation

Les opérations composées

Parfois il devient assez lassant de réécrire les mêmes chose et l'on sait que les programmeurs sont des gros fainéants ! Il existe des raccourcis lorsque l'on veut effectuer une opération sur une même variable :

```
1 int x, y;
2
3 x += y; // correspond à x = x + y;
4 x -= y; // correspond à x = x - y;
5 x *= y; // correspond à x = x * y;
6 x /= y; // correspond à x = x / y;
```

des opérations composées

Avec un exemple, cela donnerait :

```
int var = 10;

// opération 1
var = var + 6;
var += 6; // var = 16

// opération 2
var = var - 6;
var -= 6; // var = 4

// opération 3
var = var * 6;
var *= 6; // var = 60

// opération 4
var = var / 5;
var /= 5; // var = 2
```

Opérations composées : détails

L'opération de bascule (ou "inversion d'état")

Un jour, pour le projet du BAC, je devais (ou plutôt "je voulais") améliorer un code qui servait à programmer un module d'une centrale de gestion domestique. Mon but était d'afficher un choix à l'utilisateur sur un écran. Pour ce faire, il fallait que je réalise une **bascule programmée** (c'est comme ça que je la nomme maintenant). Et après maintes recherches et tests, j'ai réussi à trouver ! Et il s'avère que cette "opération", si l'on peut l'appeler ainsi, est très utile dans certains cas. Nous l'utiliserons notamment lorsque l'on voudra faire clignoter une lumière. Sans plus attendre, voilà cette astuce :

```
// on définit une variable x qui ne peut prendre que la valeur 0 ou 1
// (soit vraie ou fausse)
boolean x = 0;

x = 1 - x;  // c'est la toute l'astuce du programme !
```

La bascule

Analysons cette instruction. A chaque exécution du programme (oui, j'ai omis de vous le dire, il se

répète jusqu'à l'infini), la variable x va changer de valeur :

- 1er temps : x = 1 - x soit x = 1 - 0 donc x = 1
- 2e temps : x = 1 - x or x vaut maintenant 1 donc x = 1 - 1 soit x = 0
- 3e temps : x vaut 0 donc x = 1 - 0 soit x = 1

Ce code se répète donc et à chaque répétition, la variable x change de valeur et passe de 0 à 1, de 1 à 0, de 0 à 1, etc. Il agit bien comme une bascule qui change la valeur d'une variable booléenne. En mode console cela donnerait quelque chose du genre (n'essayez pas cela ne marchera pas, c'est un exemple) :

```
x = 0
x = 1
x = 0
x = 1
x = 0
...
```

Le résultat de la bascule

Mais il existe d'autres moyens d'arriver au même résultat. Par exemple, en utilisant l'opérateur '!' qui signifie "not" ("non"). Ainsi, avec le code suivant on aura le même fonctionnement :

```
x = !x;
```

La bascule avec l'opérateur `!`

Puisqu'à chaque passage x devient "pas x" donc si x vaut 1 son contraire sera 0 et s'il vaut 0, il deviendra 1.

Les conditions

Qu'est-ce qu'une condition ?

C'est un choix que l'on fait entre plusieurs propositions. En informatique, les conditions servent à tester des variables. Par exemple : Vous faites une recherche sur un site spécialisé pour acheter une nouvelle voiture. Vous imposez le prix de la voiture qui doit être inférieur à 5000€ (c'est un petit budget). Le programme qui va gérer ça va faire appel à un **test conditionnel**. Il va éliminer tous les résultats de la recherche dont le prix est supérieur à 5000€.

Quelques symboles

Pour tester des variables, il faut connaître quelques symboles. Je vous ai fait un joli tableau pour que vous vous repériez bien :

Symbole	A quoi il sert	Signification
==	Ce symbole, composé de deux égales, permet de tester l'égalité entre deux variables	... est égale à ...
<	Celui-ci teste l'infériorité d'une variable par rapport à une autre	...est inférieur à...
>	Là c'est la supériorité d'une variable par rapport à une autre	...est supérieur à...
<=	teste l'infériorité ou l'égalité d'une variable par rapport à une autre	...est inférieur ou égale à...
>=	teste la supériorité ou l'égalité d'une variable par rapport à une autre	...est supérieur ou égal à...
!=	teste la différence entre deux variables	...est différent de...

Table: Les symboles conditionnels

"Et si on s'occupait des conditions ? Ou bien sinon on va tranquillement aller boire un bon café ?"

Cette phrase implique un choix : le premier choix est de s'occuper des conditions. Si l'interlocuteur dit oui, alors il s'occupe des conditions. Mais s'il dit non, alors il va boire un bon café. Il a donc l'obligation d'effectuer une action sur les deux proposées. En informatique, on parle de **condition**. "si la condition est vraie", on fait une action. En revanche "si la condition est fausse", on exécute une autre action.

If...else

La première condition que nous verrons est la condition if...else. Voyons un peu le fonctionnement.

if

On veut tester la valeur d'une variable. Prenons le même exemple que tout à l'heure. Je veux tester si la voiture est inférieure à 5000€.

```
1  int prix_voiture = 4800; // variable : prix de la voiture définit à 4800€
```

D'abord on définit la variable "prix_voiture". Sa valeur est de 4800€. Ensuite, on doit tester cette valeur. Pour tester une condition, on emploie le terme if (de l'anglais "si"). Ce terme doit être suivi de parenthèses dans lesquelles se trouveront les variables à tester. Donc entre ces parenthèses, nous devons tester la variable prix_voiture afin de savoir si elle est inférieure à 5000€.

```
1  if(prix_voiture < 5000)
```

```
{
    // la condition est vraie, donc j'achète la voiture
}

```

Le test d'une condition

On peut lire cette ligne de code comme ceci : "**si** la variable prix_voiture est inférieure à 5000, on exécute le code qui se trouve entre les accolades.

> Les instructions qui sont entre les accolades ne seront exécutées que si la condition testée est vraie !

Le "schéma" à suivre pour tester une condition est donc le suivant :

```
if(/* contenu de la condition à tester */)
{
    // instructions à exécuter si la condition est vraie
}
```

Syntaxe d'une condition

else

On a pour l'instant testé que si la condition est vraie. Maintenant, nous allons voir comment faire pour que d'autres instructions soient exécutées si la condition est fausse. Le terme else de l'anglais "sinon" implique notre deuxième choix si la condition est fausse. Par exemple, si le prix de la voiture est inférieur à 5000€, alors je l'achète. Sinon, je ne l'achète pas. Pour traduire cette phrase en ligne de code, c'est plus simple qu'avec un if, il n'y a pas de parenthèses à remplir :

```
int prix_voiture = 5500;

if(prix_voiture < 5000)
{
    // la condition est vraie, donc j'achète la voiture
}
else
{
    // la condition est fausse, donc je n'achète pas la voiture
}
```

Si "", alors "", sinon ""

Le else est généralement utilisé pour les conditions dites **de défaut**. C'est lui qui à le pouvoir sur toutes les conditions, c'est-à-dire que si aucune condition n'est vraie, on exécute les instructions qu'il contient.

Le else n'est pas obligatoire, on peut très bien mettre plusieurs if à la suite.

Le "schéma" de principe à retenir est le suivant :

```
else  // si toutes les conditions précédentes sont fausses...
{
    // ...on exécute les instructions entre ces accolades
}
```

Syntaxe du else

else if

A ce que je vois, on a pas trop le choix : soit la condition est vraie, soit elle est fausse. Il n'y a pas d'autres possibilités ?

Bien sur que l'on peut tester d'autres conditions ! Pour cela, on emploie le terme else if qui signifie "sinon si..." Par exemple, SI le prix de la voiture est inférieur à 5000€ je l'achète; SINON SI elle est égale à 5500€ mais qu'elle a l'option GPS en plus, alors je l'achète ; SINON je ne l'achète pas. Le sinon si s'emploie comme le if :

```
int prix_voiture = 5500;

if(prix_voiture < 5000)
{
    // la condition est vraie, donc j'achète la voiture
}
else if(prix_voiture == 5500)
{
    // la condition est vraie, donc j'achète la voiture
```

```
}
else
{
    // la condition est fausse, donc je n'achète pas la voiture
}

```

Utilisation de `else if`

A retenir donc, si la première condition est fausse, on teste la deuxième, si la deuxième est fausse, on teste la troisième, etc. "Schéma" de principe du else, idem au if :

```
else if(/* test de la condition */)  // si elle est vraie...
{
    // ...on exécute les instructions entre ces accolades
}
```

Syntaxe du `else if`

> Le "else if" ne peut pas être utilisée toute seule, il faut obligatoirement qu'il y ait un "if" avant !

Les opérateurs logiques

Et si je vous posais un autre problème ? Comment faire pour savoir si la voiture est inférieure à 5000€ ET si elle est grise ? :twisted:

> C'est vrai ça, si je veux que la voiture soit grise en plus d'être inférieure à 5000€, comment je fais ?

Il existe des opérateurs qui vont nous permettre de tester cette condition ! Voyons quels sont ses opérateurs puis testons-les !

Opérateur	Signification
&&	... ET ...
||	... OU ...
!	NON

Table: les opérateurs logiques

ET

Reprenons ce que nous avons testé dans le else if : SI la voiture vaut 5500€ ET qu'elle a l'option GPS en plus, ALORS je l'achète. On va utiliser un if et un opérateur logique qui sera le ET :

```
int prix_voiture = 5500;
int option_GPS = TRUE;

/* l'opérateur && lie les deux conditions qui doivent être
vraies ensemble pour que la condition soit remplie */
if(prix_voiture == 5500 && option_GPS)
{
    // j'achète la voiture si la condition précédente est vraie
}
```

Conjonction de deux conditions

OU

On peut reprendre la condition précédente et la première en les assemblant pour rendre le code beaucoup moins long.

> Et oui, les programmeurs sont des flemmards !

Rappelons quelles sont ces conditions :

```
int prix_voiture = 5500;
int option_GPS = TRUE;

if(prix_voiture < 5000)
{
    // la condition est vraie, donc j'achète la voiture
}
else if(prix_voiture == 5500 && option_GPS)
{
    // la condition est vraie, donc j'achète la voiture
}
else
{
```

```
    // la condition est fausse, donc je n'achète pas la voiture
}
```

Vous voyez bien que l'instruction dans le if et le else if est la même. Avec un opérateur logique, qui est le OU, on peut rassembler ces conditions :

```
int prix_voiture = 5500;
int option_GPS = TRUE;

if((prix_voiture < 5000) || (prix_voiture == 5500 && option_GPS))
{
    // la condition est vraie, donc j'achète la voiture
}
else
{
    // la condition est fausse, donc je n'achète pas la voiture
}
```

Utilisation du OU logique

Lisons la condition testée dans le if : "SI le prix de la voiture est inférieur à 5000€ OU SI le prix de la voiture est égal à 5500€ ET la voiture à l'option GPS en plus, ALORS j'achète la voiture".

> Attention aux parenthèses qui sont à bien placer dans les conditions, ici elles n'étaient pas nécessaires, mais elles aident à mieux lire le code.

NON

> Moi j'aimerais tester "si la condition est fausse j'achète la voiture". Comment faire ?

~~Toi t'as un souci~~ Il existe un dernier opérateur logique qui se prénomme NON. Il permet en effet de tester si la condition est fausse :

```
int prix_voiture = 5500;

if(!(prix_voiture < 5000))
{
```

```
    // la condition est vraie, donc j'achète la voiture
}

```

L'opérateur négation

Se lit : "SI le prix de la voiture N'EST PAS inférieur à 5000€, alors j'achète la voiture". On s'en sert avec le caractère ! (point d'exclamation), généralement pour tester des variables booléennes. On verra dans les boucles que ça peut grandement simplifier le code.

Switch

Il existe un dernier test conditionnel que nous n'avons pas encore abordé, c'est le switch. Voilà un exemple :

```
int options_voiture = 0;

if(options_voiture == 0)
{
    // il n'y a pas d'options dans la voiture
}
else if(options_voiture == 1)
{
    // la voiture a l'option GPS
}
else if(options_voiture == 2)
{
    // la voiture a l'option climatisation
}
else if(options_voiture == 3)
{
    // la voiture a l'option vitre automatique
}
else if(options_voiture == 4)
{
    // la voiture a l'option barres de toit
}
else if(options_voiture == 5)
{
    // la voiture a l'option  siège éjectable
}
else
{

```

```
    // retente ta chance ;-)
}

```

Un grand nombre de `else if`

Ce code est indigeste ! C'est infâme ! Grotesque ! Pas beau ! En clair, il faut trouver une solution pour changer cela. Cette solution existe, c'est le switch. Le switch, comme son nom l'indique, va tester la variable jusqu'à la fin des valeurs qu'on lui aura données. Voici comment cela se présente :

```
int options_voiture = 0;

switch (options_voiture)
{
    case 0:
        // il n'y a pas d'options dans la voiture
        break;
    case 1:
        // la voiture a l'option GPS
        break;
    case 2:
        // la voiture a l'option climatisation
        break;
    case 3:
        // la voiture a l'option vitre automatique
        break;
    case 4:
        // la voiture a l'option barres de toit
        break;
    case 5:
        // la voiture a l'option siège éjectable
        break;
    default:
        // retente ta chance ;-)
        break;
}
```

Utilisation de `switch`

Si on testait ce code, en réalité cela ne fonctionnerait pas car il n'y a pas d'instruction pour afficher à l'écran, mais nous aurions quelque chose du genre :

```
il n'y a pas d'options dans la voiture
```

Si option_voiture vaut maintenant 5 :

```
la voiture a l'option siège éjectable
```

> L'instruction **break** est **nécessaire**, car si vous ne la mettez pas, l'ordinateur, ou plutôt la carte Arduino, va exécuter toutes les instructions. Pour éviter cela, on met cette instruction break, qui vient de l'anglais "casser/arrêter" pour dire à la carte Arduino qu'il faut arrêter de tester les conditions car on a trouvé la valeur correspondante.

La condition ternaire ou condensée

Cette condition est en fait une simplification d'un test if...else. Il n'y a pas grand-chose à dire dessus, par conséquent un exemple suffira : Ce code :

```
int prix_voiture = 5000;
int achat_voiture = FALSE;

if(prix_voiture == 5000) // si c'est vrai
{
    achat_voiture = TRUE; // on achète la voiture
}
else // sinon
{
    achat_voiture = FALSE; // on n'achète pas la voiture
}
```

Est équivalent à celui-ci :

```
int prix_voiture = 5000;
int achat_voiture = FALSE;

achat_voiture= (prix_voiture == 5000) ? TRUE : FALSE;
```

Utilisation de la condition ternaire

Cette ligne :

```
1 achat_voiture= (prix_voiture == 5000) ? TRUE : FALSE;
```

Se lit comme ceci : "Est-ce que le prix de la voiture est égal à 5000€ ? SI oui, alors j'achète la voiture SINON je n'achète pas la voiture"

> Bon, vous n'êtes pas obligé d'utiliser cette condition ternaire, c'est ~~vraiment pour les gros flemmards~~ juste pour simplifier le code, mais pas forcément la lecture de ce dernier.

Nous n'avons pas encore fini avec le langage Arduino. Je vous invite donc à passer à la partie suivante pour poursuivre l'apprentissage de ce langage.

Le matériel Le langage Arduino (2/2)

mmaire

I Découverte de l'Arduino

1. Présentation d'Arduino
2. Quelques bases élémentaires
3. Le logiciel
4. Le matériel
5. **Le langage Arduino (1/2)**

La syntaxe du langage

Les variables

Les conditions

6. Le langage Arduino (2/2)

II Gestion des entrées / sorties

III La communication avec Arduino

IV Les grandeurs analogiques

V Les capteurs et l'environnement autour d'Arduino

VI Le mouvement grâce aux moteurs

VII L'affichage, une autre manière d'interagir

VIII Internet of Things : Arduino sur Internet

rtager

- Twitter
- Facebook
- Google+
- Diaspora*
- Envoyer par mail

lécharger

- PDF (8,7 Mio)
- Archive (1,1 Mio)