Les bases de la programmation

Les programmes en langage Arduino, basé sur les langages C/C++, peuvent être divisés en trois parties principales :

. la structure,

. les valeurs (variables et constantes) ,

. les fonctions.

1. Structure du programme

Dans les fonctions de bases **"setup()"** et **"loop()"** obligatoires dans tous les programmes ou dans toute autre fonction, on utilisera les éléments de langage suivant :

1.1 Syntaxe de base

. Le point-virgule ;

Il est obligatoire à la fin de chaque instruction. Pour le compilateur, les sauts de lignes n'ont pas de signification : c'est le point-virgule qui marque la fin de ligne.
Oublier le point-virgule en fin de ligne donnera une erreur de compilation.

Exemple :

```
int a = 13;     // le point-virgule indique la fin de l'instruction
```

. Les accolades {}

Les accolades sont un élément majeur de la programmation en langage C.

Toute ouverture d'une accolade d'ouverture "{" doit obligatoirement être accompagnée dans le code d'une accolade de fermeture "}" correspondante. On dit souvent qu'il faut que les accolades soient équilibrées (càd autant de"{" que de "}").

L’Arduino IDE inclut une fonctionnalité pratique pour vérifier la correspondance des accolades entre elles. Il suffit de sélectionner une accolade, ou même de cliquer juste après une accolade, et l’accolade d’ouverture ou de fermeture associée sera mise en surbrillance.

Les accolades sont utilisées dans les fonctions, les boucles et les conditions

Exemples :

- Avec des fonctions :

```
void myfunction(datatype argument){        // ouverture de la fonction
            // instructions
 }      // fermeture de la fonction
```

- Dans des boucles :

```
while (boolean expression)
{  // accolade d'ouverture du code de la boucle while
            // instructions
}  // fermeture de la boucle

do
 {   // accolade d'ouverture du code de la boucle do
            // instructions
 } while (boolean expression);  // fermeture de la boucle

for (initialisation; termination condition; incrementing expr)
{  // ouverture du code de la boucle for
            //instructions
 }  // fermeture de la boucle
```

- Dans des conditions :

```
if (boolean expression)
{ // ouverture du code de la condition if
            // instructions
} // fermeture du code de la condition if

else if (boolean expression)
{ // ouverture du code de l'instruction else if
            // instructions
} // fermeture du code de l'instruction else if

      else
         { // ouverture du code de l'instruction else
            // instructions
         } // fermeture du code de l'instruction else
```

. // Les commentaires

Lorsqu'on écrit dans le programme derrière un double slash **//**, c'est pour donner des informations sur le programme. Les commentaires n'ont aucune action sur le programme.

On peut écrire un commentaire sur plusieurs lignes entre **/* et */** .

Exemple : /*commentaire sur plusieurs lignes, commentaire sur plusieurs lignes,
commentaire sur plusieurs lignes */

1.2 Les opérateurs arithmétiques :

. opérateur d'assignement (signe égal unique) =

Cet opérateur stocke la valeur à droite du signe égal dans la variable à gauche du signe égal.

Le signe égal dans le langage de programmation C est appelé opérateur d'assignement. Il a un sens différent de celui qu'il a en algèbre où il indique une équivalence ou une égalité. L'opérateur d'assignement indique au microcontrôleur d'évaluer la valeur ou l'expression qui se trouve à droite du signe égal et de la stocker dans la variable à gauche du signe égal :

variable = valeur ;

Exemple :

```
int sensVal;              // déclare une variable entière de 16 bits nommée senVal
senVal = analogRead(0);       // mémorise la valeur de la conversion analogique numérique
                                 dans la variable
```

. Addition + , Soustraction - , Multiplication * , et Division /

Ces opérateurs renvoient respectivement la somme, la différence, le produit ou le quotient entre deux opérandes (= entre deux termes).

Cette opération est réalisée en utilisant le type des données des opérandes. Ainsi par exemple, 9 / 4 donne 2 dès lors que 9 et 4 sont de type int.

Si les opérandes sont de deux types de données différents, le plus "grand" est utilisé pour le calcul.

Si un des nombres (opérandes) est du type float ou double, le calcul dit "en virgule flottante" est utilisé pour le calcul (càd que 9/4 donne 2.25).

- Syntaxe :

```
result = value1 + value2;
result = value1 - value2;
result = value1 * value2;
result = value1 / value2;
```

- Exemple :

```
y = y + 3;
x = x - 7;
i = j * 6;
r = r / 5;
```

1.3. Les opérateurs de comparaison

==, !=, <, > sont des opérateurs logiques de comparaison :

- x == y (x est égal à y)
- x != y (x est différent de y)
- x < y (x est inférieur à y)
- x > y (x est supérieur à y)
- x <= y (x est inférieur ou égal à y)
- x >= y (x est supérieur ou égal à y)

1.4. Les opérateurs booléens

Ces opérateurs peuvent être utilisés à l'intérieur de la condition d'une instruction if pour associer plusieurs conditions (ou opérandes) à tester.

. && (ET LOGIQUE)

VRAI seulement si les deux opérandes sont VRAI, par exemple :

```
if (digitalRead(2) == HIGH  && digitalRead(3) == HIGH) { // lit l'état de 2 entrées
  // ...
}
// est VRAI seulement si les deux entrées sont à l'état HAUT simultanément.
```

. || (OU LOGIQUE)

VRAI si l'un des deux opérandes est VRAI, par exemple :

```
if (x > 0 || y > 0) { // si x supérieur à 0 ou si y supérieur à 0
  // ...
}
 // est VRAI si soit x, soit y est supérieur à 0.
```

. ! (NON LOGIQUE)

VRAI si l'opérande est FAUX, par exemple :

```
if (!x) {
  // ...
}
  // VRAI si la variable x est FAUSSE (càd si x = 0)
```

1.5. Les opérateurs composés

. ++ (incrément) / -- (décrément)

Ces opérateurs incrémentent ou décrémentent une variable entière de type int ou long (pouvant être unsigned).

- Syntaxe :

```
x++;  // incrémente x de un, sans modifier x
++x;  // incrémente x de un et modifie x

x-- ;   // décrément x de un, sans modifier x
--x ;   // décrémente x de un, et modifie x
```

- Exemples :

```
x = 2; // variable x = 2
y = ++x;      // x contient 3 et y contient 3
y = x++;      // x contient toujours 2, y contient 3
```

. += , -= , *= , /=

Ces opérateurs réalisent une opération mathématique entre une variable **x** et une autre variable ou une constante **y**. Les opérateurs **+**,**-** ,***** et **/** sont juste utiles pour raccourcir la forme complète des opérations mathématiques listées ci-dessous :

```
x += y;   // équivaut à l'expression x = x+y
x -= y;   // équivaut à l'expression x = x- y
x *= y;   // équivaut à l'expression x = x * y
x /= y;   // équivaut à l'expression x= x / y
```

- Exemples :

```
x = 2;
x += 4;        // x contient 6
x -= 3;        // x contient 3
x *= 10;       // x contient 30
x /= 2;        // x contient 15
```

1.6. Structures de contrôle

. if (condition)

L'instruction if ("si" en français), utilisée avec un opérateur logique de comparaison, permet de tester si une condition est vraie, par exemple si la mesure d'une entrée analogique est bien supérieure à une certaine valeur.

Le format d'un test if est le suivant :

```
if (uneVariable > 50)
{
  // faire quelque chose
}
```

Dans cet exemple, le programme va tester si la variable **uneVariable** est supérieure à 50. Si c'est le cas, le programme va réaliser une action particulière. Autrement dit, si l'état du test entre les parenthèses est vrai, les instructions comprises entre les accolades sont exécutées. Sinon, le programme se poursuit sans exécuter ces instructions.

Les accolades peuvent être omises après une instruction if. Dans ce cas, la suite de la ligne (qui se termine par un point-virgule) devient la seule instruction de la condition. Tous les exemples suivants sont corrects :

```
if (x > 120)  digitalWrite(LEDpin, HIGH);

if (x > 120)
digitalWrite(LEDpin, HIGH);


if (x > 120){ digitalWrite(LEDpin, HIGH); }
```

. if / else

L'instruction if/else (si/sinon en français) permet un meilleur contrôle du déroulement du programme que la simple instruction if, en permettant de grouper plusieurs tests ensemble. Par exemple, une entrée analogique peut-être testée et une action réalisée si l'entrée est inférieure à 500, et une autre action réalisée si l'entrée est supérieure ou égale à 500. Le code ressemblera à cela :

```
if (valsensor < 500)
{
  // action A
}
else
{
  // action B
}
```

L'instruction **else** peut contenir un autre test **if**, et donc des tests multiples, mutuellement exclusifs peuvent être réalisés en même temps.

Chaque test sera réalisé après le suivant jusqu'à ce qu'un test VRAI soit rencontré.
Quand une condition vraie est rencontrée, les instructions associées sont réalisées, puis le programme continue son exécution à la ligne suivant l'ensemble de la construction **if/else**.
Si aucun test n'est VRAI, le bloc d'instructions par défaut **else** est exécuté, s'il est présent, déterminant ainsi le comportement par défaut.

Un bloc **else if** peut être utilisé avec ou sans bloc de conclusion **else**.
Un nombre illimité de branches **else if** est autorisé.

```
if (valsensor < 500)
{
  // action A
}
else if (valsensor >= 1000)
{
  // action B
}
else
{
  // action C
}
```

. Boucle for

L'instruction **for** est utilisée pour répéter l'exécution d'un bloc d'instructions regroupées entre des accolades.
Un compteur incrémental est habituellement utilisé pour incrémenter et finir la boucle.
L'instruction **for** est très utile pour toutes les opérations répétitives et est souvent utilisées en association avec des tableaux de variables pour agir sur un ensemble de données ou broches.

Il y a 3 parties dans l'entête d'une boucle for :

```
for (initialisation; condition; incrementation) {
        //instructions
}
```

L'initialisation a lieu en premier et une seule fois. A chaque exécution de la boucle, la condition est testée.
Si elle est VRAIE, le bloc d'instructions et l'incrémentation sont exécutés, puis la condition est testée de nouveau. Lorsque la condition devient FAUSSE, la boucle stoppe.

Exemple :

```
for (int i=0; i <= 255; i++){   // boucle incrémentant la variable i de 0 à 255, de 1 en 1

        //instructions

  } // fin de la boucle for
```

. Boucle while

Les boucles **while** ("tant que" en anglais) bouclent sans fin, et indéfiniment, jusqu'à ce que la condition ou l'expression entre les parenthèses ( ) devienne fausse.
Quelque chose doit modifier la variable testée, sinon la boucle **while** ne se terminera jamais. Cela peut être dans votre code, soit une variable incrémentée, ou également une condition externe comme le test d'un capteur.

```
while(expression){   // tant que l'expression est vraie

         // instructions à effectuer

}
```

Avec **expression,** une instruction (booléenne) qui renvoie un résultat VRAI ou FAUX

Exemple :

```
var = 0;
while(var < 200){  // tant que la variable est inférieur à 200

      // fait quelque chose 200 fois de suite...

  var++; // incrémente la variable

}
```

. boucle do – while

La boucle **do / while** ("faire tant que" en anglais) fonctionne de la même façon que la boucle **while**, à la différence près que la condition est testée à la fin de la boucle, et par conséquent la boucle do sera toujours exécutée au moins une fois.

- Syntaxe :

```
do // faire...
{
   // instructions

} while (condition); // tant que la condition est vraie
```

Avec **condition**, une expression booléenne dont le résultat peut être VRAI ou FAUX.

- Exemple :

```
do // faire...
{
   x = analogRead(A0);  // lit la valeur de la tension d'un capteur

} while (x < 100); // ...tant que x est inférieur à 100
```

Remarque :

L'instruction **break** est utilisée pour sortir d'une boucle do, for ou while, en passant outre le déroulement normal de la boucle.

Exemple :

```
for (int i=0; i <= 100; i++){   // boucle incrémentant la variable i de 0 à 100, de 1 en 1

  x = analogRead(A0);  // lit la valeur de la tension d'un capteur

     if (x < 100)

     {

     break ; // si la mesure est inférieure à un seuil, on sort de la boucle

     }

 } // fin de la boucle for
```

2. Variables et constantes

Les variables sont des expressions que l'on utilise dans les programmes pour stocker des valeurs, telles que la tension de sortie d'un capteur présente sur une broche analogique.

2.1. Les constantes Arduino prédéfinies

Dans le langage Arduino, les constantes sont des variables prédéfinies. Elles sont utilisées pour rendre les programmes plus faciles à lire.

. INPUT ET OUTPUT

Ces constantes sont utilisées pour définir des broches numériques en entrée ou en sortie :

Les broches numériques peuvent être utilisées soit en mode **INPUT** (= en entrée), soit en mode **OUTPUT** (= en sortie).

Modifier le mode de fonctionnement d'une broche du mode **INPUT** (=ENTREE) en mode **OUTPUT**(=SORTIE) avec l'instruction **pinMode()** change complètement le comportement électrique de la broche.

- Broches configurées en entrée (INPUT)

Les broches d'une carte Arduino configurées en mode **INPUT** (=en entrée) à l'aide de l'instruction **pinMode()** sont dites en état de "haute-impédance".

Ces broches ne consomment alors qu'une toute petite intensité (de l'ordre du microampère) du circuit sur lequel elles sont connectées. Leur mode de fonctionnement est équivalent à celui d'un voltmètre.

- Broches configurées en sortie (OUTPUT)

Les broches configurées en mode **OUTPUT** (= en sortie) avec l'instruction **pinMode()** sont en état dit de "basse-impédance" .

Cela veut dire qu'elles peuvent fournir une quantité significative de courant aux autres circuits. Chaque broche de la carte Arduino peut fournir jusqu'à 40 mA d'intensité au circuit sur lequel elle est connectée. Son mode de fonctionnement est équivalent à celui d'un générateur.

Cependant, l'intensité cumulée fournie par les broches de la carte ne doit pas dépasser les 200mA !

. HIGH ET LOW

Lorsqu'on lit ou on écrit sur une broche numérique, seuls deux états distincts sont possibles, la broche ne peut être qu'à deux valeurs : **HIGH (HAUT) ou LOW (BAS).**

HIGH

La signification de la constante **HIGH** (en référence à une broche) est quelque chose de différent selon que la broche est définie comme une ENTREE ou comme une SORTIE.

Si la broche est configurée en ENTREE avec l'instruction pinMode, et lue avec l'instruction digitalRead, le microcontrôleur renverra HIGH (=HAUT) si une tension de 3V ou + est présente sur la broche.

Quand une broche est configurée en SORTIE avec l'instruction pinMode, et mise au niveau HAUT avec l'instruction digitalWrite, la broche est mise à 5V. Dans cet état, la broche peut fournir une certaine intensité (jusqu'à 40 mA par broche, sans dépasser 200 mA pour l'ensemble des broches).

LOW

La constante LOW a également une signification différente selon que la broche est configurée en ENTREE ou en SORTIE.

Quand une broche est configurée en ENTREE avec l'instruction pinMode, et lue avec l'instruction digitalRead, le microcontrôleur renverra un niveau BAS si une tension de 2V ou moins est présente sur la broche.

Quand la broche est configurée en SORTIE avec l'instruction pinMode, et est mise au niveau LOW avec l'instruction digitalWrite, la broche est à 0 volts.

. true ET false

Il existe deux constantes utilisées pour représenter le VRAI et le FAUX dans le langage Arduino : true et false.

false *(= FAUX)*

La constante false est définie comme le 0 (zéro).

true *(=VRAI)*

La constante true est définie comme tout entier qui n'est pas 0 (zéro).

Noter que les constantes **true** et **false** sont écrites en minuscule à la différence des constantes **HIGH**, **LOW**, **INPUT** et **OUTPUT**.

2.2. Les variables - Types de données

Les variables peuvent être de type variés. Les différents types de variables sont décrits ci-dessous :

. Int

Déclare une variable de type int (pour integer, entier en anglais). Les variables de type int sont le type de base pour le stockage de nombres, et ces variables stockent une valeur sur 2 octets. Elles peuvent donc stocker des valeurs allant de - 32 768 à 32 767 (valeur minimale de -2^{15} et une valeur maximale de $2^{15}-1$).

- Syntaxe :

```
int var = val;
```

- **var** : le nom de votre variable de type int
- **val** : la valeur d'initialisation de la variable

- Exemple :

```
int ledPin = 13; // déclare une variable de type int appelée LedPin et valant 13
```

- Remarque :

Quand les variables dépassent la valeur maximale de leur capacité, elles "débordent" et reviennent à leur valeur minimale, et ceci fonctionne dans les 2 sens :

```
int x   // déclaration de la variable de type int appelée x
x = -32,768;  // x prend la valeur -32 768
x = x - 1;       // x vaut maintenant 32 767, car déborde dans le sens négatif

x = 32,767;  // x prend la valeur 32 767
x = x + 1;       // x vaut maintenant la valeur - 32 768, car déborde dans le sens positif
```

. unsigned int

Déclare une variable de type int non-signée. Les variables de type unsigned int (entiers non signée) sont les mêmes que les variables de type int en ce sens qu'elle stocke une valeur sur 2 octets. Cependant, au lieu de stocker des valeurs négatives, les variables de type unsigned int stocke uniquement des valeurs positives, dans une fourchette allant de 0 à 65535 (2^{16}-1).

- Syntaxe :

```
unsigned int var = val;
```

- **var** : le nom de votre variable int
- **val** : la valeur donnée à votre variable

- Exemple :

```
unsigned int ledPin = 13; // déclaration d'une variable entière non signée nommée ledPin et valant 13
```

- Remarque :

Quand la valeur des variables excède leur capacité maximale, elles "débordent" et reprennent leur valeur minimale, et ceci se produit dans les 2 sens.

```
unsigned int x  // déclaration d'une variable int non signée nommée x
  x = 0;          // x vaut 0
  x = x - 1;      // x contient maintenant 65535
  x = x + 1;      // x contient maintenant 0
```

. long

Déclare des variables de type long. Les variables de type long sont des variables de taille élargie pour le stockage de nombre entiers, sur 4 octets (32 bits), de -2 147 483 648 à + 2 147 483 647.

- Syntaxe :

```
long var = valeur;
```

. var : le nom de la variable long

. valeur : la valeur donnée à la variable

- Exemple :

```
long speedOfLight = 186000L;   //déclare une variable de type long
   // le 'L' pour forcer à traiter la variable dans le format de donnée de type long
```

. unsigned long

Déclare une variable de type long non signé. Les variables de type long non signé sont des variables de taille élargie pour le stockage de nombre entier qui stocke les valeurs sur 4 octets (32 bits).

A la différence des variables de type long standard, les variables de type long unsigned ne peuvent pas stocker des nombres négatifs, la fourchette des valeurs qu'elles peuvent stocker s'étendant de 0 à 4 294 967 295 (2^{32}-1)

- Syntaxe :

```
unsigned long var = valeur;
```

. var : le nom de votre variable de type long

. valeur : la valeur donnée à la variable

- Exemple :

```
unsigned long time; // déclare une variable de type long non signé appelée time
void setup()
{
 Serial.begin(9600); // initialise la connexion série à 9600 bauds
}
void loop()
{
 Serial.print("Time: ");
 // Met dans la variable time le temps écoulé depuis le démarrage
 time = millis();
 //affiche le temps écoulé depuis que le programme a démarré
 Serial.println(time);
// attend une seconde avant d'envoyer un nouveau message au PC
 delay(1000);
}
```

. float

Déclare des variables de type "virgule-flottante", c'est à dire des nombres à virgules. Les nombres à virgule sont souvent utilisés pour l'expression des valeurs analogiques.

Les nombres à virgule ainsi stockés peuvent prendre des valeurs entre - 3.4028235E+38 et 3.4028235E+38. Ils sont stockés sur 4 octets (32 bits) de mémoire.

Les variables float ont seulement 6 à 7 chiffres de précision. Ceci concerne le nombre total de chiffres, pas seulement le nombre à droite de la virgule.

- Syntaxe :

```
float var = valeur;
```

. var : le nom de la variable

. valeur : la valeur donnée à la variable

- Exemple :

```
float myfloat; // déclare une variable à virgule appelée myfloat
float sensorCalibrate = 1.117; // déclare une variable à virgule appelée sensorCalibrate

int x; // déclare une variable entière de type int appelée x
int y; // déclare une variable entière de type int appelée y
float z; // déclare une variable nombre à virgule de type float appelée z

x = 1;            // x vaut 1
y = x / 2;          // y vaut 0 car les entiers ne supporte pas les décimales
z = float (x)/ 2.0;   // z vaut 0.5 (float (x) : conversion de x en float, alors x=1.0)
```

. char

Déclare une variable d'un octet de mémoire (8 bits) qui contient une valeur correspondant à un caractère. Les caractères unitaires sont écrits entre guillemets uniques, comme ceci : **'A'**.

Les caractères sont stockés de la même façon que les nombres. A chaque caractère correspond une valeur numérique comprise entre 0 et 127 (code ASCII) :

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Source : www.LookupTables.com

La variable de type char est de type signée, ce qui veut dire qu'elle peut contenir des valeurs allant de -128 à +127.

- <u>Syntaxe</u> :

```
char monChar='B'; // déclare une variable char
```

- <u>Exemple</u> :

```
char myChar = 'A'; // déclare une variable char initialisée avec la valeur A
char myChar = 65;      // expression équivalente car la valeur ASCII de A est 65
```

3. Les fonctions

Une fonction est un bloc d’instruction que l’on peut appeler à tout endroit d’un programme.

En langage Arduino, des bibliothèques de fonctions prédéfinies sont disponibles afin de manipuler facilement les entrées/sorties, gérer le temps, gérer la communication série…

On peut également créer ses propres fonctions qui seront utiles pour l'exécution de tâches répétitives et évitant alors la réécriture des lignes de codes à chaque fois que se présente ces tâches.

3.1. Fonctions prédéfinies des Entrées/Sorties numériques

. pinMode()

Configure la broche spécifiée pour qu'elle se comporte soit en entrée, soit en sortie.

- Syntaxe :

pinMode(broche, mode)

- Paramètres :

broche: le numéro de la broche de la carte Arduino dont le mode de fonctionnement (entrée ou sortie) doit être défini.

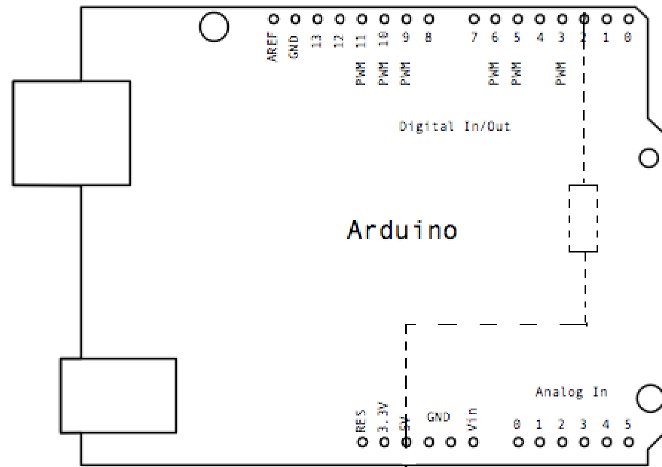
mode: soit INPUT (entrée en anglais) ou OUTPUT (sortie en anglais)

. digitalWrite()

Met un niveau logique HIGH (HAUT en anglais) ou LOW (BAS en anglais) sur une broche numérique.

Si la broche a été configurée en SORTIE avec l'instruction pinMode(), sa tension est mise à la valeur correspondante : 5V pour le niveau HAUT, 0V (masse) pour le niveau BAS.

Si la broche est configurée en ENTREE, écrire un niveau HAUT sur cette broche a pour effet d'activer la résistance interne de 20K sur cette broche.



A l'inverse, mettre un niveau BAS sur cette broche configurée en ENTREE désactivera la résistance interne.

- Syntaxe :

digitalWrite(broche, valeur)

- Paramètres :

broche: le numéro de la broche de la carte Arduino

valeur : HIGH ou LOW (ou bien 1 ou 0)

- Exemple :

```
int ledPin = 13;              // LED connectée à la broche numérique n° 13

void setup()
{
  pinMode(ledPin, OUTPUT);      // met la broche utilisée avec la LED en SORTIE
}

void loop()
{
  digitalWrite(ledPin, HIGH);   // allume la LED
  delay(1000);                  // pause 1 seconde
  digitalWrite(ledPin, LOW);    // éteint la LED
  delay(1000);                  // pause 1 seconde
}
```

Remarques :

- Les broches analogiques peuvent être utilisées en tant que broches numériques, représentées par les nombres 14 (entrée analogique 0) à 19 (entrée analogique 5).

- Cette instruction met la valeur 0/1 dans le bit de donnée qui est associé à chaque broche, ce qui explique qu'on puisse le mettre à 1, même si la broche est en entrée.

- Ne pas oublier qu'une broche numérique ne peut fournir que 40mA (milliampères) tant en entrée qu'en sortie, et que l'ensemble des broches de la carte Arduino ne peut fournir que 200mA. Par conséquent, limiter l'intensité utilisée pour chaque broche à une dizaine de mA par des résistances adaptées : 220 Ohms pour une LED par exemple.

. digitalRead()

Lit l'état (= le niveau logique) d'une broche précise en entrée numérique, et renvoie la valeur HIGH (HAUT en anglais) ou LOW (BAS en anglais).

- Syntaxe :

digitalRead(broche)

- Paramètres :

broche : le numéro de la broche numérique que vous voulez lire. (int)

- Valeur retournée :

Renvoie la valeur HIGH (HAUT en anglais) ou LOW (BAS en anglais)

- Exemple :

```
int ledPin = 13; // LED connectée à la broche n°13
int inPin = 7;   // un bouton poussoir connecté à la broche 7
int val = 0;     // variable pour mémoriser la valeur lue

void setup()
{
  pinMode(ledPin, OUTPUT);      // configure la broche 13 en SORTIE
  pinMode(inPin, INPUT);      // configure la broche 7 en ENTREE
}
void loop()
{
  val = digitalRead(inPin);   // lit l'état de la broche en entrée et met le résultat dans la variable
  digitalWrite(ledPin, val);    // met la LED dans l'état du BP  (càd allumée si appuyé et inversement)
}
```

Dans ce programme, la broche 13 reflète fidèlement l'état de la broche 7 qui est une entrée numérique.

Remarques :

- Si la broche numérique en entrée n'est connectée à rien, l'instruction digitalRead() peut retourner aussi bien la valeur HIGH (HAUT en anglais) ou LOW (BAS en anglais) (et cette valeur peut changer de façon aléatoire)

- Les broches analogiques peuvent être utilisées en entrée numériques et sont désignées par les numéros 14 (entrée analogique 0) à 19 (entrée analogique 5).

3.2. Fonctions prédéfinies des Entrées/Sorties analogiques

. analogRead()

Lit la valeur de la tension présente sur la broche spécifiée. La carte Arduino comporte 6 voies connectées à un convertisseur analogique-numérique 10 bits. Cela signifie qu'il est possible de transformer la tension d'entrée entre 0 et 5V en une valeur numérique entière comprise entre 0 et 1023. Il en résulte une résolution (écart entre 2 mesures) de : 5 volts / 1024 intervalles, autrement dit une précision de 0.0049 volts (4.9 mV) par intervalle.

Une conversion analogique-numérique dure environ 100 µs pour convertir l'entrée analogique, et donc la fréquence maximale de conversion est environ de 10 000 fois par seconde.

- Syntaxe :

analogRead(broche_analogique)

- Paramètres :

broche_analogique : le numéro de la broche analogique (et non le numéro de la broche numérique) sur laquelle il faut convertir la tension analogique appliquée (0 à 5 sur la plupart des cartes Arduino)

- Valeur retournée :

valeur int (0 to 1023) correspondant au résultat de la mesure effectuée

- Exemple :

```
int analogPin = 3;              // Capteur analogique relié à la broche A3
int val = 0;                    // variable de type int pour stocker la valeur de la mesure

void setup()
{
  Serial.begin(9600);        //  initialisation de la connexion série
}

void loop()
{

  // lit la valeur de la tension analogique présente sur la broche
  val = analogRead(analogPin);

   // affiche la valeur (comprise en 0 et 1023) dans la fenêtre terminal PC
  Serial.println(val);

}
```

Remarques :

- La tension de référence par défaut est le 5V : il est possible d'utiliser une autre valeur si besoin.

- Les broches analogiques sont utilisées en entrée. Il n'est pas nécessaire de les configurer au préalable à l'appel de la fonction analogRead

- Si la broche analogique est laissée non connectée, la valeur renvoyée par la fonction **analogRead()** va fluctuer en fonction de plusieurs facteurs (tels que la valeur des autres entrées analogiques, la proximité de votre main vis à vis de la carte Arduino, etc.).

. analogWrite()

Génère un signal carré de fréquence 490 Hz sur une broche de la carte Arduino (onde PWM - Pulse Width Modulation en anglais ou MLI - Modulation de Largeur d'Impulsion en français).

Après avoir appelé l'instruction analogWrite(), la broche générera un signal carré stable avec un rapport cyclique (fraction de la période où la broche est au niveau haut) de durée spécifiée (en %), jusqu'à l'appel suivant de l'instruction analogWrite() (ou bien encore l'appel d'une instruction digitalRead() ou digitalWrite() sur la même broche).

- Syntaxe :

analogWrite(broche, valeur);

- Paramètres :

- **broche**: la broche utilisée pour "écrire" l'impulsion. Cette broche devra être une broche ayant la fonction PWM, Par exemple, sur la UNO, ce pourra être une des broches 3, 5 ,6 ,9 ,10 ou 11.

- **valeur**: la largeur du "duty cycle" (proportion de l'onde carrée qui est au niveau HAUT) : entre 0 (0% HAUT donc toujours au niveau BAS) et 255 (100% HAUT donc toujours au niveau HAUT).

- Exemple :

Le programme suivant fixe la luminosité d'une LED proportionnellement à la valeur de la tension lue depuis un potentiomètre :

```
int ledPin = 9;      // LED connectée sur la broche 9
int analogPin = 3;   // le potentiomètre connecté sur la broche analogique 3
int val = 0;         // variable pour stocker la valeur de la tension lue

void setup()
{
  pinMode(ledPin, OUTPUT);   // configure la broche en sortie
}

void loop()
{
  val = analogRead(analogPin);   // lit la tension présente sur la broche en entrée
  analogWrite(ledPin, val / 4);  // Résultat d'analogRead entre 0 to 1023,
                                 // résultat d'analogWrite entre 0 to 255
                                 // => division par 4
}
```

- **Remarques** :

- Il n'est pas nécessaire de faire appel à l'instruction pinMode() pour mettre la broche en sortie avant d'appeler la fonction analogWrite().

- L'impulsion PWM générée sur les broches 5 et 6 pourront avoir des "duty cycle" plus long que prévu. La raison en est l'interaction avec les instructions millis() et delay(), qui partagent le même timer interne que celui utilisé pour générer l'impulsion de sortie PWM.

3.3. Fonctions prédéfinies des Entrées/Sorties avancées

. tone()

Génère une onde carrée (onde symétrique avec rapport cyclique (niveau haut/période) à 50%), à la fréquence spécifiée en Hertz (Hz) sur une broche.

La durée peut être précisée, sinon l'impulsion continue jusqu'à l'appel de l'instruction noTone().

La broche peut être connectée à un buzzer piézoélectrique ou autre haut-parleur pour jouer des notes (les sons audibles s'étendent de 20Hz à 20 000Hz).

Une seule note peut être produite à la fois. Si une note est déjà jouée sur une autre broche, l'appel de la fonction tone() n'aura aucun effet (tant qu'une instruction noTone() n'aura pas eu lieu).

Si la note est jouée sur la même broche, l'appel de la fonction tone() modifiera la fréquence jouée sur cette broche.

- Syntaxe :

  tone(broche, frequence)

  tone(broche, frequence, durée)

- Paramètres :

  broche : la broche sur laquelle la note est générée.

  frequence : la fréquence de la note produite en hertz (Hz)

  durée : la durée de la note en millisecondes (optionnel)

. noTone()

Stoppe la génération d'impulsion produite par l'instruction tone(). N'a aucun effet si aucune impulsion n'est générée.

- Syntaxe :

  noTone(broche)

- Paramètres :

  broche : la broche sur laquelle il faut stopper la note.

. pulseIn()

Lit la durée d'une impulsion (soit niveau HAUT, soit niveau BAS) appliquée sur une broche (configurée en ENTREE).

Par exemple, si le paramètre valeur est HAUT, l'instruction pulseIn() attend que la broche passe à HAUT, commence alors le chronométrage, attend que la broche repasse au niveau BAS et stoppe alors le chronométrage.

L'instruction renvoie la durée de l'impulsion en microsecondes. L'instruction s'arrête et renvoie 0 si aucune impulsion n'est survenue dans un temps spécifié.

Il est préférable de travailler avec des impulsions d'une durée de 10 microsecondes à 3 minutes.

- Syntaxe :

  pulseIn(broche, valeur)
  pulseIn(broche, valeur, delai_sortie)

- Paramètres :

  broche : le numéro de la broche sur laquelle vous voulez lire la durée de l'impulsion.

  valeur : le type d'impulsion à "lire" : soit HIGH (niveau HAUT) ou LOW (niveau BAS

  delai_sortie (optionnel): le nombre de microsecondes à attendre pour début de l'impulsion. La valeur par défaut est 1 seconde. (type unsigned long)

- Valeur renvoyée :

  La durée de l'impulsion (en microsecondes) ou 0 si aucune impulsion n'a démarrer avant le délai de sortie. (type unsigned long)

- Exemple :

```
int broche = 7; // variable de broche
unsigned long duree;  // variable utilisée pour stocker la durée

void setup()
{
 pinMode(broche, INPUT); // met la broche en entrée
}

void loop()
{
 duree = pulseIn(broche, HIGH); // met la durée de l'impulsion de niveau HAUT dans la variable duree
}
```

3.4. Fonctions prédéfinies de gestion du temps

. delay(ms)

Réalise une pause dans l'exécution du programme pour la durée (en millisecondes) indiquée en paramètre.

- Syntaxe :

delay (ms);

- Paramètres :

ms (unsigned long): le nombre de millisecondes que dure la pause

. unsigned long millis()

Renvoie le nombre de millisecondes depuis que la carte Arduino a commencé à exécuter le programme courant. Ce nombre débordera (càd sera remis à zéro) après 50 jours approximativement.

- Syntaxe :

variable_unsigned_long = millis();

- Exemple :

```
unsigned long time;

void setup(){

  Serial.begin(9600);

}

void loop(){

  Serial.print("Time: ");

  time = millis();

  //affiche sur le PC le temps depuis que le programme a démarré
```

```
  Serial.println(time);

  // pause d'une seconde afin de ne pas envoyer trop de données au PC

  delay(1000);

}
```

3.5. Fonctions prédéfinies de communication

La librairie **Serial** est utilisée pour les communications par le port série entre la carte Arduino et un ordinateur ou d'autres composants.

Le port série communique sur les broches 0 (RX) et 1 (TX) avec l'ordinateur via le port USB. C'est pourquoi, si on utilise cette fonctionnalité, il n'est pas possible d'utiliser les broches 0 et 1 en tant qu'entrées ou sorties numériques.

Le logiciel Arduino IDE dispose d'un moniteur série, qui permet de recevoir et d'envoyer des informations via une liaison série. Il est particulièrement intéressant pour les tests des programmes puisqu'il permet d'afficher les valeurs des variables, les états logiques des entrées et des sorties de l'Arduino, etc…

Il suffit pour cela de cliquer sur le bouton du moniteur série dans la barre d'outils puis de sélectionner le même débit de communication que celui utilisé dans l'appel de la fonction begin() pour établir la liaison série.

Les principales fonctions de la librairie **Serial** :

. **begin();** (Configure la vitesse de transmission du port série)

. **print();** (Envoie une donnée sous forme de chaine de caractères sur le port série)

. **println();** (Envoie une donnée sur le port série et fait un saut à la ligne)

. **write();** (Ecrit des données binaires sur le port série. Ces données sont envoyées comme une série d'octets)

. **read();** (Lis les données contenues dans la mémoire tampon (buffer) du port série)

. **flush();** (Vide la mémoire tampon de la liaison série)

. **available();** (Donne le nombre d'octets (caractères) disponible pour lecture dans la file d'attente (buffer) du port série)

Une fois la liaison série établie dans la fonction **setup()** avec la fonction **Serial.begin()**, il est possible d'envoyer des données depuis la carte Arduino vers le moniteur série avec la fonction "**print()**" de la classe "**Serial**".

On peut envoyer différents types d'informations :

. Envoie d'une chaîne de caractères, sans saut de ligne à la fin :

Serial.print("Test");

. Envoie d'une chaîne de caractères, avec saut de ligne à la fin :

Serial.println("Test");

. Envoie de la valeur d'une variable, sans saut de ligne à la fin :

int a = 3;

Serial.print(a);

- Exemple :

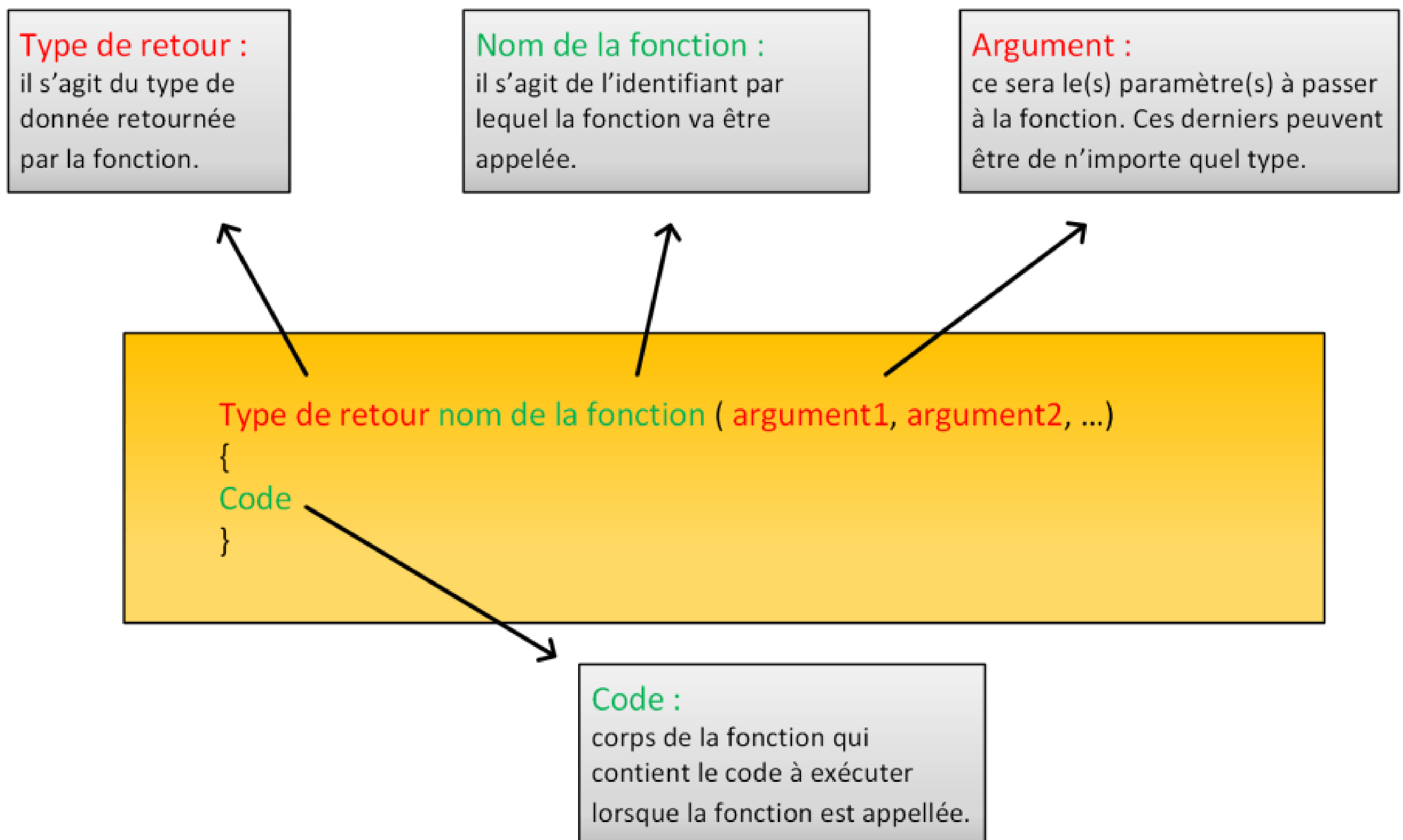
```
int analogValue = 0; // variable pour stocker la valeur analogique sur la broche A0
void setup() {
// initialise le port série à 9600 bauds
Serial.begin(9600);
}
void loop() {
// lit la valeur analogique sur la broche A0
analogValue = analogRead(0);

// affiche cette valeur
Serial.println(analogValue);
}
```

3.6. Fonctions propres au programme

Pour pouvoir utiliser une fonction propre à un programme, il faut au préalable la déclarer et comme son appel doit être possible à tout moment du déroulement du programme, il faut qu'elle soit déclarée de façon globale. C'est-à-dire que cela se fera en dehors des fonctions **setup()** et **loop()**.

Une fonction possède un nom, des paramètres d'entrée et des paramètres de sortie. On appelle **prototype d'une fonction**, la spécification de ces trois données :



Et tout ce que fait la fonction doit être placé entre deux accolades.

Il existe deux types de fonctions :

- Les fonctions qui ne renvoie aucune donnée,
- Les fonctions qui renvoie une donnée.

. Si la fonction ne renvoie aucune donnée, son type de retour sera **void** (vide).

<u>Exemple</u> :

Ce programme affiche le résultat de la division de 10 par 3 avec 1 à 5 chiffres après la virgule, dans le moniteur série.

Pour cela, on déclare une fonction appelée **affichfloat()** dont le type de retour est void (vide) et les paramètres d'entrées sont le nombre à virgule, **float a**, à afficher et un entier, **b**, représentant le nombre de chiffres après la virgule du nombre **a** qui seront affichés.

La fonction est appelée dans une boucle **for()** de la fonction setup() et le résultat de la division est affiché dans le moniteur série.

```
void affichfloat(float a, int b) { // Déclaration de la fonction « affichfloat »
    Serial.println(a,b); // Affichage de la variable a avec b chiffres après la virgule
}

void setup() {
    Serial.begin(9600);
    float result = 10.0/3.0;
    for(int i = 1 ; i < 6 ; i++){

    affichfloat(result,i);  // Appel de la fonction « affichfloat »

}
}

void loop() {
}
```

. Si la fonction renvoie une donnée, elle devra le faire avec le mot-clef **return**.

Exemple :

Le programme suivant affiche la table de multiplication de 2 dans le moniteur série. Pour cela, une fonction appelée **multiplication()** dont le type de retour est un entier **c** et les paramètres d'entrées sont 2 entiers **a** et **b** est déclarée.

La fonction est appelée dans une boucle **for()** de la fonction setup() et le résultat de la multiplication est affiché dans le moniteur série.

```
int multiplication (int a, int b) {  // Déclaration de la fonction « multiplication »

  int c ;
  c = a * b ;
  return c ;

}

void setup() {

Serial.begin(9600);

for(int i = 1 ; i < 10 ; i++){

   int result = multiplication(2,i);   // Appel de la fonction « multiplication »

   Serial.println(result);  // Affichage de la donnée de retour de la fonction « multiplication »

}
}

void loop() {

}
```