Les pointeurs et tableaux

1. Petites mise au point préalable :

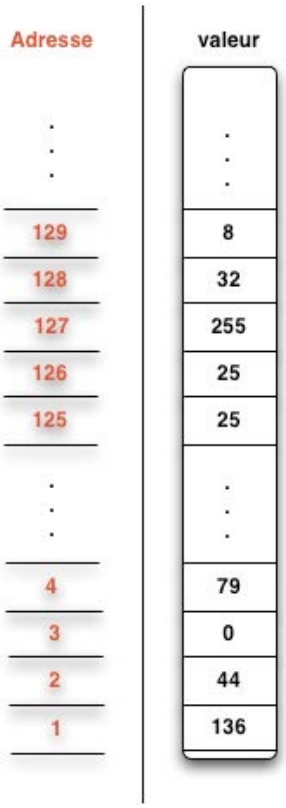
Les pointeurs étant intimement liés au monde des variables et à la mémoire, une petite mise au point s'impose.

1.1 La mémoire :

Quel que soit le type de mémoire utilisée dans le monde du numérique[1], ram(Ex: mémoire vive de votre ordinateur), magnétique (Ex:disque dur), flash (Ex: clé usb), etc… Elles possèdent toutes un point commun, elles permettent de stocker des **valeurs** numériques[1] dans un espace/zone donné(e) et tout comme la boite aux lettres de votre maison ou appartement dans une rue cet espace/zone sera identifié(e) par une position unique (numérotation successive 1,2,3, ...) dans cette rue.

Rem: Tout espaces/zones d'une même mémoire ont une taille identique (généralement 1 octet).

Voici une vue schématique de notre mémoire.

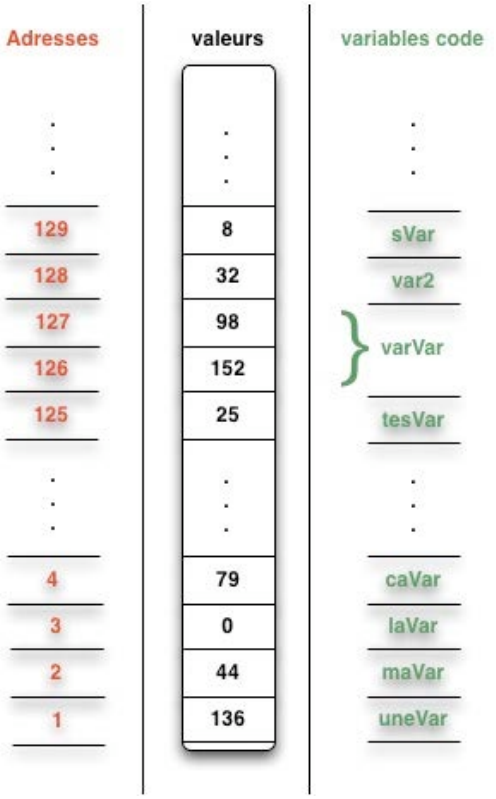


1.2 Les variables :

Les variables en programmation sont la **représentations** dans votre code d'un ou plusieurs[3] espace(s) mémoire qui vous permettront de stocker des **valeurs** et les modifier (faire varier).

Elles sont symbolisée dans le code par un nom [4] permettant de les identifier et distinguer dans ce code.

Chacune possède donc une adresse permettant de la localiser dans cette mémoire comme vu précédemment.



Retenez bien ce schéma, lorsque je parlerai de **valeurs** il s'agira bien de la **valeur** en mémoire propre à une variable.

Rem: Nous pouvons voir que la variable "varVar" prend 2 espaces mémoire[4], son adresse cera la première, ici 126.

*!!! Pour ceux qui auraient déjà des notions sur les tableaux, ici "varVar" est bien une simple variable contenant **une** seule valeur étendue à plusieurs espaces et non un tableau.*

2. Dans le vif du sujet, les pointeurs :

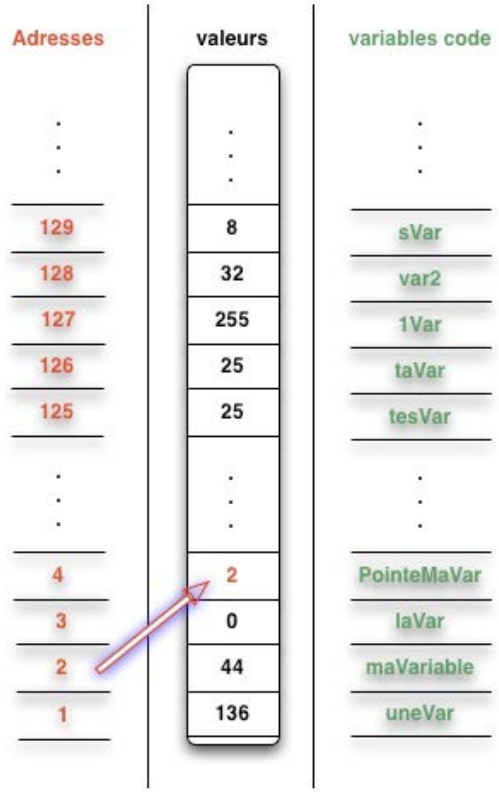
2.1 Les pointeurs démystification :

Je vais peut être vous étonner mais les pointeurs ne sont rien d'autre qu'un type de variable tout comme sont les variables de type char, int, float, boolean, etc.

Tout comme les autres types de variables, les pointeurs servent à stocker des **valeurs** numériques[1] en mémoire, ces **valeurs** sont de type adresse et ces adresses, je vous le donne en mille, sont des adresses d'espaces/zones mémoires donc de variables.

En fait ce ne sont que de simples variables classiques dans lesquelles nous serons capables de manipuler des **valeurs** de type adresse, par contre il y a une petite particularité à l'utilisation qui la différencie des autres et cette particularité est symbolisée par le petit mais terrifiant symbole '*' .

Les pointeurs n'étant qu'un type particulier de variables, elles possèdent également une adresse et ils peuvent donc très bien avoir pour **valeur** l'adresse même d'un autre pointeur.

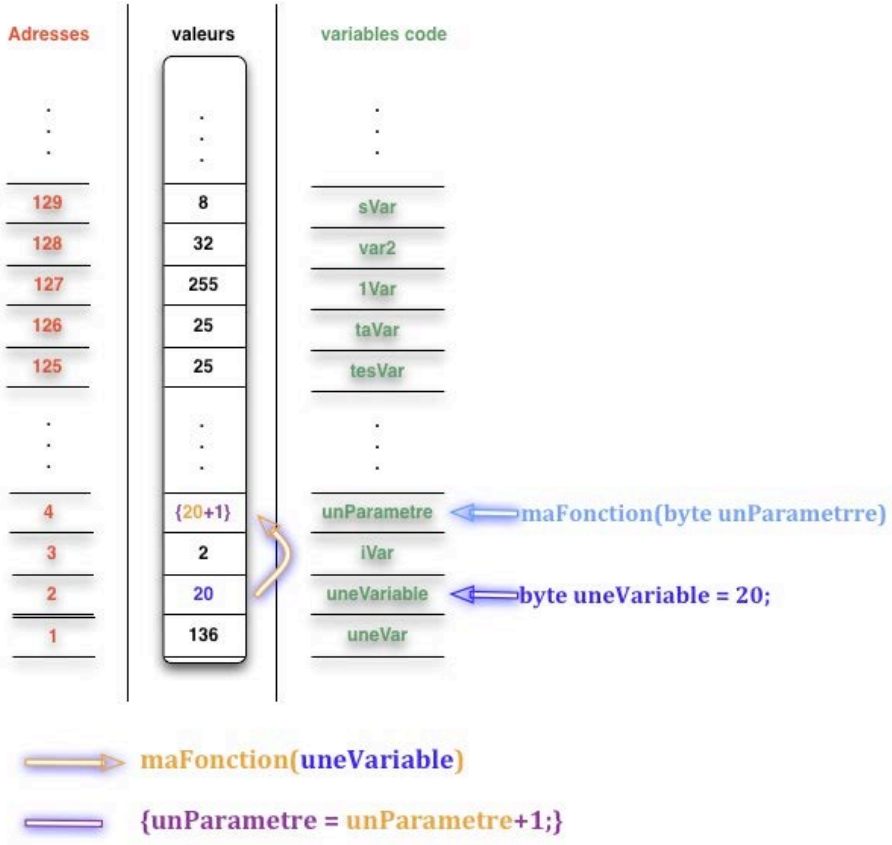


La **valeur** de mon pointeur sera l'adresse de ma variable "maVariable".

2.2 En quoi le fait de stocker des adresses peut être utile ? :

Comme un exemple vaut mieux qu'un long discours, voici un simple appel de fonction dans notre fonction loop.

```
void loop()
{
   byte uneVariable = 20;
   maFonction(uneVariable);
}


 void maFonction(byte unParametre)
 {
       unParametre = unParametre+1
 }
```



Si nous analysons chaque étape de ce programme, on remarque que nous avons créé une variable de type byte dans la fonction (bloc²) loop, initialisé cette variable à 20, transmis cette **valeur** au paramètre "unParametre" de la fonction "maFonction", puis nous avons incrémenté(+) ça **valeur** de 1 et ... c'est tout, "uneVariable" n'a pas changé.

La variable "unParametre" de notre fonction ayant une existence courte² sa **valeur** disparaitra avec elle.

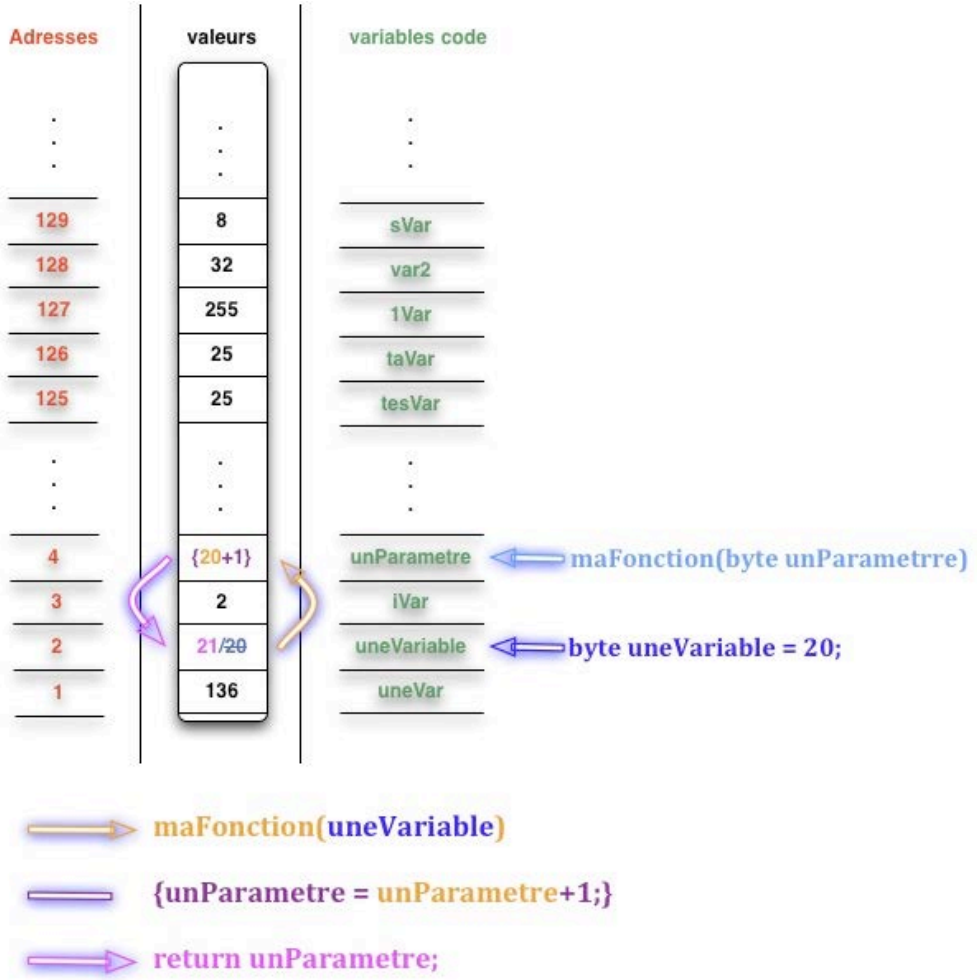
Rem: la valeur en mémoire ne "disparait" pas réellement, c'est juste l'espace mémoire qui n'est plus lié à la variable (déréférencé), donc libre pour une nouvelle variable temporaire ou autre allocation.

Notre variable temporaire "unParametre" peux très bien retrouver cette valeur au prochain appel de la fonction, c'est pourquoi il est important d'initialiser les variables à la déclaration.

Arf moi ce que j'aurais voulu c'est obtenir un résultat sur ma variable "uneVariabe" .

Pourtant il existe un moyen très simple d'obtenir le résultat souhaité, comme vous le savez une fonction peut retourner une **valeur**, nous allons donc profiter de cette possibilité pour modifier la **valeur** de notre variable locale² à loop selon la **valeur** retournée et stockée temporairement dans *"unParametre"* par notre fonction.

```
void loop()
{
   byte uneVariable = 20;
   uneVariable = maFonction(uneVariable);
}


 byte maFonction(byte unParametre)
 {
       unParametre = unParametre+1
       return unParametre;
 }
```



Ici la **valeur** (20) de notre variable "uneVariable" est toujours transmise en paramètre et nous effectuons toujours une incrémentation (+) de 1 sur cette **valeur**.

La nouveauté c'est que maintenant notre fonction va retourner le résultat de cette dernière opération via la valeur de "unParametre".

Nous pouvons donc directement récupérer ce résultat et l'affecter (=) à notre variable "uneVariable" , sa **valeur** aura bien été modifiée comme on le désirait.

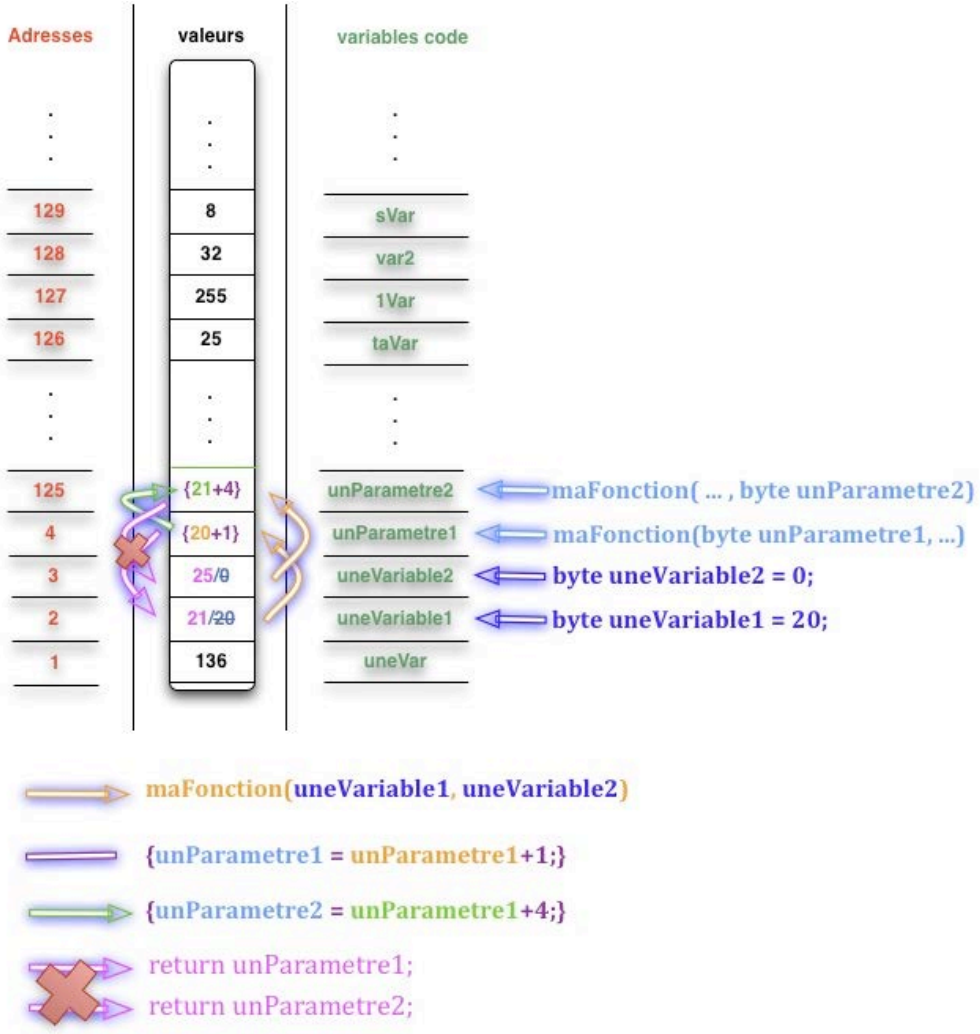
Ben c'est super, je vais pouvoir faire ce que je veux via une fonction sur toute les variables maintenant ...

Allez sur notre lancée , on voudrait maintenant ajouter des possibilités à notre fonction pour modifier plusieurs variables en un coup!

On se dit facile: un paramètre en plus à notre fonction et le tour est joué!

Donc en toute logique on va commencer par déclarer nos variables, ensuite notre fonction avec un nouveau paramètre.

```
Void loop()
{
   byte uneVariable1 = 20;
   byte uneVariable2 = 0;
   unParametre1 = maFonction(uneVariable1, uneVariable2); ?????
}

byte maFonction(byte unParametre1, byte unParametre2)
{
        unParametre1 = unParametre1+1;
        unParametre2 = unParametre1+4;
        return unParametre1,  unParametre2; ?????
}
```



Et là, pendant que vous finalisez votre fonction, c'est le drame (?????) … vous vous rendez compte que votre fonction ne peut pas retourner plusieurs **valeurs** et que même si elle le pouvait , comment les récupérer ?????

Il ne reste plus qu'à abandonner le doux rêve d'utiliser une fonction pour changer les **valeurs** de plusieurs variables en un coup .

Rassurez-vous il y a un moyen d'y parvenir et c'est maintenant qu'on va comprendre où je voulais en venir depuis le début et en quoi ma longue introduction à bien pu servir.

Comme je vous l'ai signalé, une variable sert à stocker des **valeurs** numériques[1] dans un espace/zone mémoire et cet espace/zone est identifié(e) via un numéro/emplacement unique, une adresse.

C'est là qu'une idée vous traverse l'esprit: récemment quelqu'un a parlé d'une variable capable de stocker des adresses et vu que ces adresses permettent d'identifier un espace/zone mémoire on devrait pouvoir manipuler ces derniers directement non ?

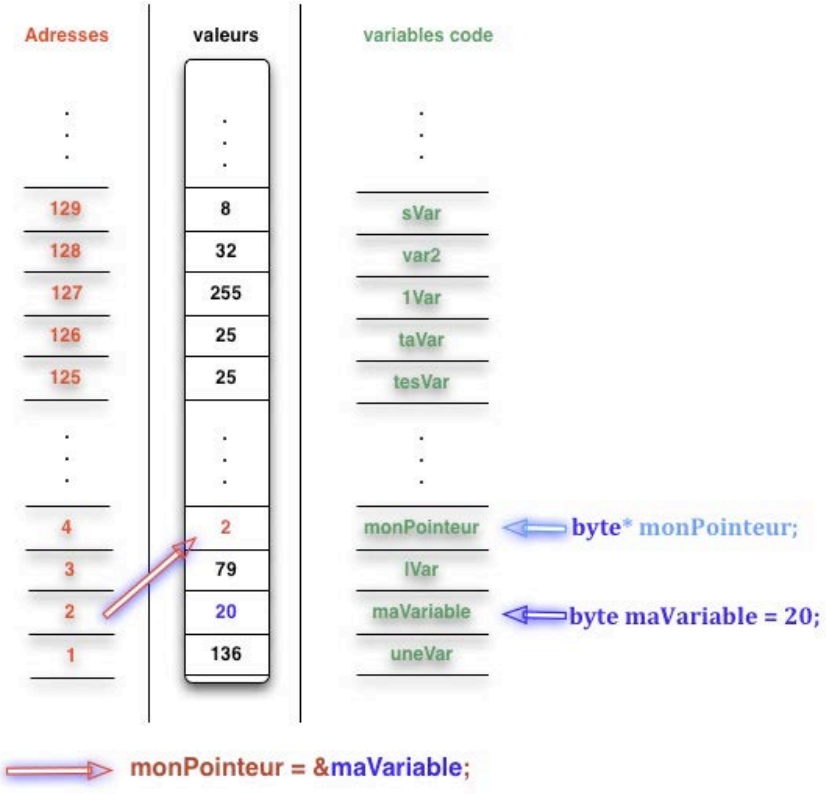
Peut être même grâce à ce petit '*' ?

C'est maintenant que nos pointeurs entrent en jeux et ici que les Romains s'empoignèrent.

2.3 Déclaration et initialisation :

La première chose à apprendre sur l'utilisation des pointeurs est "*comment les déclarer et les initialiser dans le code*".

```
void loop()
{
        byte maVariable = 20;
        byte* monPointeur =  &maVariable;
}
```



Nous voici donc avec deux nouveaux symboles qui ont leur utilité propre au monde des pointeurs.

Commençons par le fameux '*' à ne pas confondre avec l'opérateur de multiplication, qui à la déclaration signifie tout simplement : "je veux créé une variable de type pointeur".

Tien il y a un petit problème là, non ???

Si '*' identifie le type de variable créée (pointeur), que vient faire 'byte' dans l'histoire ?

Faute de frappe, mauvais copier coller, … ?

Eh bien non pas d'erreur de ma part, sachez que si grâce aux pointeurs nous pouvons manipuler des variables via leur adresse, nous ne connaissons pas le type de cette variable pointée.

Donc le '*' représente bien le type de variable déclarée (ici un pointeur) et le "byte" est le type de la variable pointée par ce pointeur.

Ce n'est pas le pointeur qui contient un "byte" mais la variable pointée par son adresse.

On dit que le pointeur pointe sur la variable pointée.

Rem: Il se peut que le symbole '' soit placé derrière le type de variable pointée ou devant le nom de notre variable pointeur.*

```
byte*   monPointeur ;

byte   *monPointeur;
```

Je vous rassure toute de suite, dans les deux cas cela signifie la même chose, il s'agit juste d'un libre choix laissé aux développeurs.

```
= &maVariable;
```

Découvrons maintenant à quoi peut bien servir le symbole '&' dans le cas des pointeurs.

Ce que l'on constate, c'est qu'il se trouve devant la variable "maVariable" et ne semble donc pas directement lié à mon pointeur ?

En fait sa signification répond à une question que vous vous êtes peut-être posée: Si les pointeurs prennent comme **valeur** des adresses de variables, comment obtient-on l'adresse d'une variable ???

Elle signifie comme vous l'aurez deviné : "donne moi l'adresse de la variable devant laquelle je me trouve".

Nous pouvons donc directement affecter (=) une adresse à un pointeur.

Rem: Vous pourriez également rencontrer ce symbole à d'autres endroits dans certains codes.

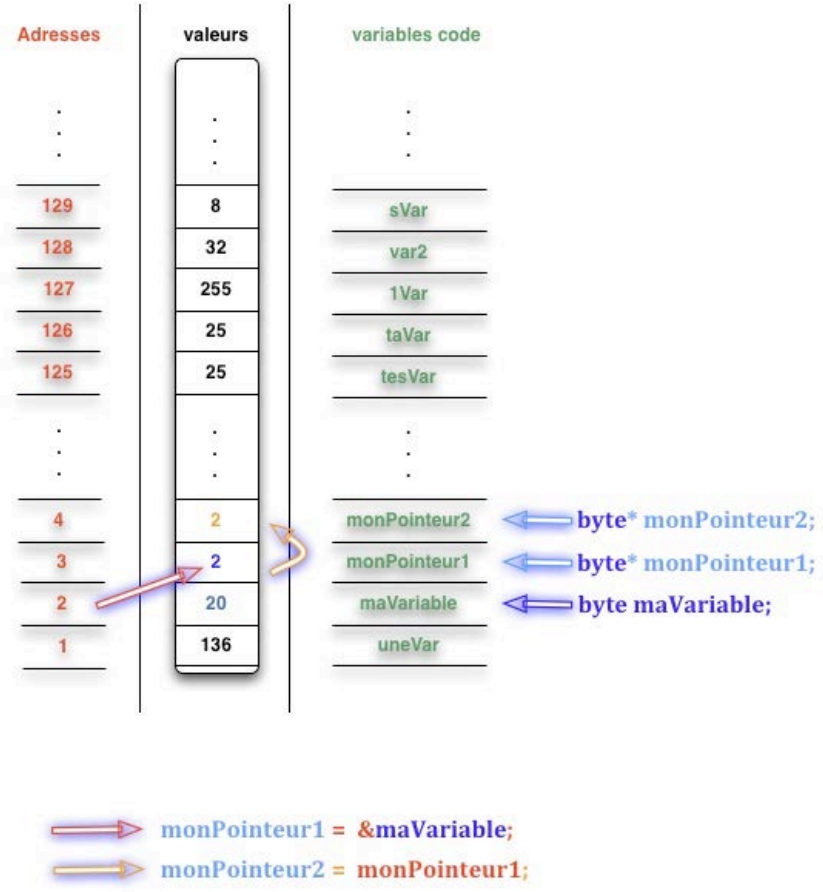
```
byte & ref = uneVariable;

uneFonction(byte & uneRef)
```

Ici il n'a pas la même signification, il s'agit d'un alias de variable/objet.

N'étant pas directement lié (il y a une relation quand même) aux pointeurs je ne m'étendrai pas dessus dans ce cours.

Maintenant je vais vous prouver qu'un pointeur n'est rien d'autre qu'un simple type de variable.

```
void loop()
{
        byte maVariable;
        byte* monPointeur1 = &maVariable;;
        byte* monPointeur2 = monPointeur1;
}
```



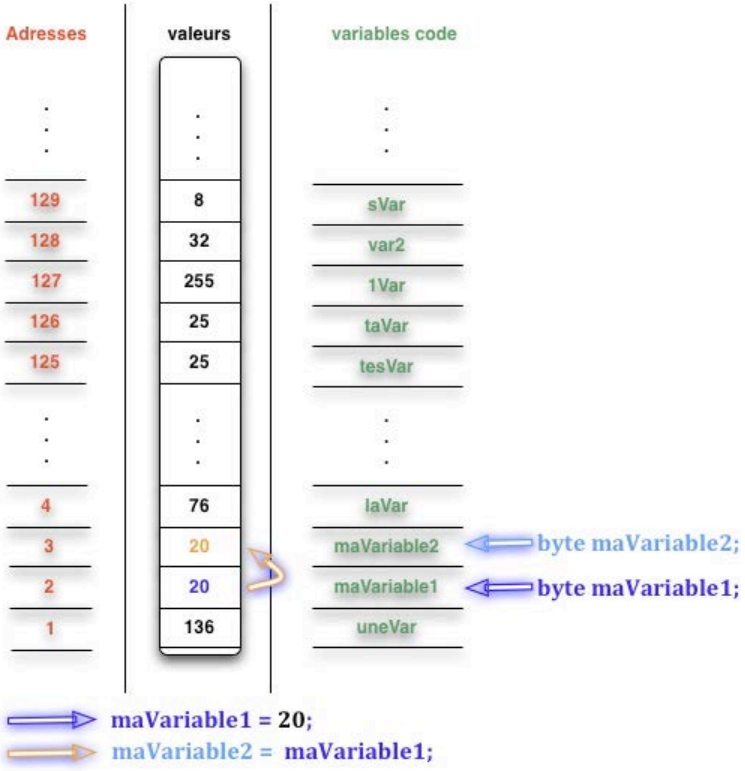
Que s'est il passé ici ?

Et bien j'ai déclaré une variable de type byte et deux pointeurs (*), j'ai affecté(=) pour **valeur** à "monPointeur1" l'adresse de ma variable "&maVariable" et ensuite à "monPointeur2" je lui affecte(=) la **valeur** de "monPointeur1" par son **nom** (ni plus, ni moins, pas de *, ni de &) donc l'adresse de ma variable "maVariable".

"monPointeur2" aura pour **valeur** la même adresse que "monPointeur1" et pointera donc sur la même variable.

C'est comme si j'avais fais ceci avec une variable classique :

```
byte maVariable1 = 20;

byte maVariable2 = maVariable1 ;
```



"maVariable2" prend la même(=) **valeur** (20) que "maVariable1", hé bien si vous comparez au schéma précédent avec nos deux pointeurs c'est pareil, on ne fait qu'une simple affectation(=) par **valeurs** du même type.

Donc sans sa petite excroissance '*' une variable de type pointeur se comporte comme une autre.

On peux même aller jusqu'à dire que c'est la variable qui sert d'utilitaire à notre symbole '*' et pas l'inverse, celui-ci pouvant très bien vivre sans l'autre.

Rem: Nous avons vu que nous pouvions initialiser nos pointeur directement, soit via l'adresse d'une variable, soit via la valeur d'un autre pointeur.

Hors il est tout à fait imaginable que l'on n'ait pas de variable ou d'autre pointeur pour l'initialiser ou tout simplement que l'on en ait pas le besoin au moment de la déclaration, nous ne pouvons pas non plus initialiser notre pointeur à une valeur quelconque telle que 0 ou autre.

Hé bien sachez que l'on peux déclarer nos pointeurs à la constante NULL.

*Petit plus: puisqu'il s'agit de simples variables nous pouvons parfaitement y appliquer les même opérations (=, +, -, ==, <,>, ...) sur leurs **valeurs** (adresse) que les autres types de variables.*

2.4 **Manipulation** :

A présent que nous savons ce que sont les pointeurs (de simples variables), comment les déclarer et initialiser, il est temps de passer à la pratique.

Reprenons notre dernier exemple, celui où l'on avais tenté en vain de modifier deux variables via une fonction mais qui ne peut retourner qu'une seule **valeur**.

```
byte maFonction(byte unParametre1, byte unParametre2)
{
        unParametre1 = unParametre1+1;
        unParametre2 = unParametre1+4;
        return unParametre1,  unParametre2; ?????
        return unParametre1;
         ?????+?????
        return unParametre2;
}
```

Que faudrait-il changer à notre fonction pour pouvoir manipuler directement nos variables ?

Comme nous l'avons vu depuis le début de ce chapitre, les pointeurs permettent de manipuler directement des variables par leur adresse et en particulier[6] les variables locales[2] qui ne sont pas visibles en dehors de leurs blocs respectifs.

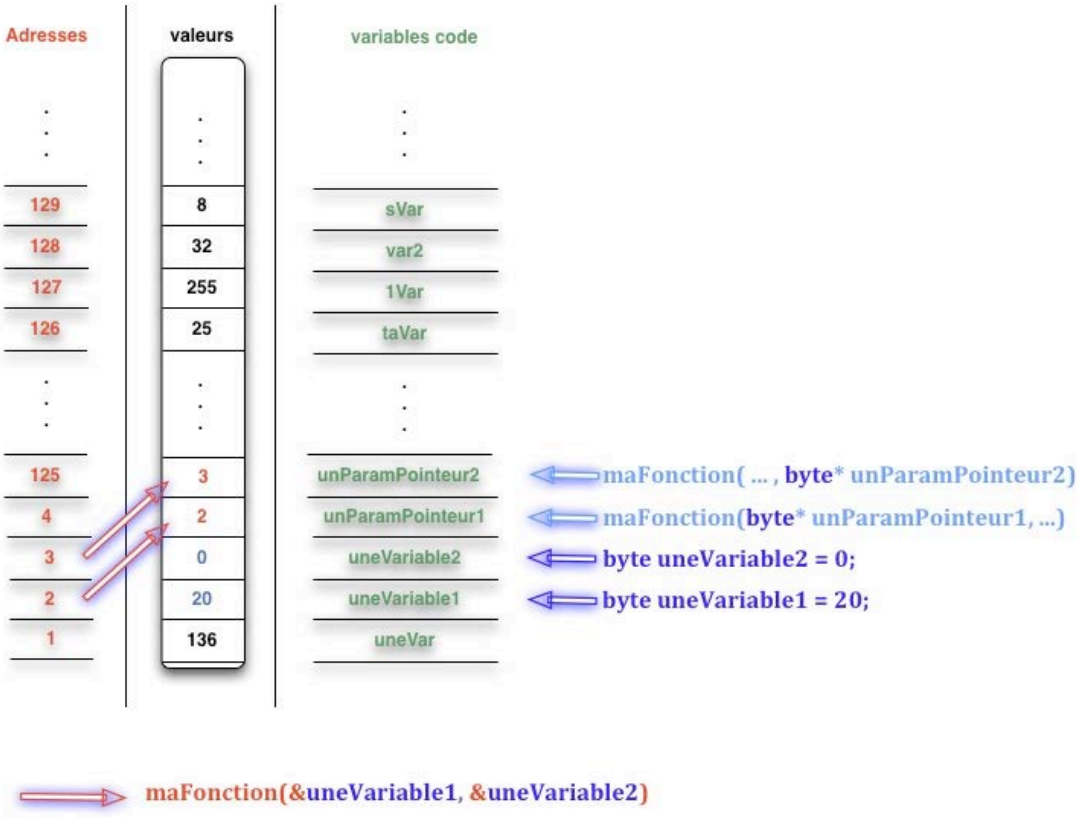
C'est une des raisons principale de l'existance des pointeurs: rendre visible ce qui ne peut être vu en dehors de leurs blocs[2], ici des variables locales .

```
void maFonction(byte* unParamPointeur1, byte* unParamPointeur2)
{
}
```

Nous déclarerons donc en paramètres de fonction des variables pointeurs de variable de type byte (le pointeur pointe l'adresse d'une variable de type byte).

Notre fonction toute prête à réceptionner des adresses de variables, nous allons la mettre à l'épreuve et tester tout cela.

```
void loop()
{
   byte uneVariable1 = 20;
   byte uneVariable2 = 0;
   maFonction(&uneVariable1, &uneVariable2);
}
void maFonction(byte* unParamPointeur1, byte* unParamPointeur2)
{
}
```

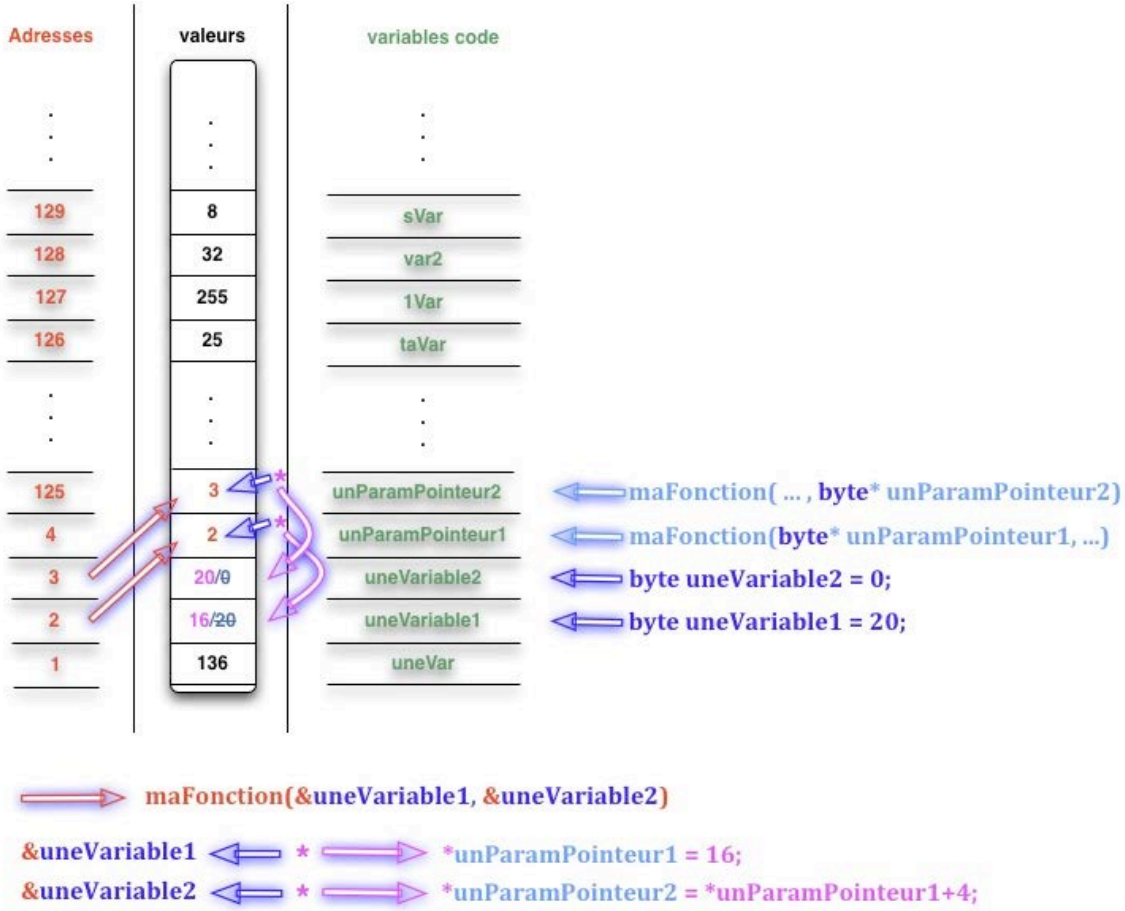


Donc ici rien de nouveau si on compare à la déclaration et initialisation des pointeurs vu au point précédent, nous déclarons nos variables de type pointeur en paramètres de fonction "(byte*

unParamPointeur1, byte* unParamPointeur2)" (déclaration) et nous leurs transmettons l'adresse de nos variables à l'appel de cette fonction "(&uneVariable1, &uneVariable2);" (initialisation).

Bon jusqu'à présent tout ceci a pu paraître "facile" (tout est relatif me direz-vous) et c'est maintenant que les choses se corsent, car nous allons manipuler nos variables grâce à leurs adresses et donc nos pointeurs.

```
void loop()

{

   byte uneVariable1 = 20;

   byte uneVariable2 = 0;

   maFonction(&uneVariable1, &uneVariable2);

}

void maFonction(byte* unParamPointeur1, byte* unParamPointeur2)

{

  *unParamPointeur1 = 16;

  *unParamPointeur2 = *unParamPointeur1+4;

}
```



Je vous sens dubitatifs face à toute ces étoiles qui vous envoient dans la lune, alors une explication supplémentaire est nécessaire.

Une chose dont je n'ai pas parlé sur ce fameux symbole '*', c'est qu'il peut avoir plusieurs significations ...

A la déclaration il signifiait : "je veux créer une variable de type pointeur".

```
byte* monPointeur;
```

Mais à l'utilisation c'est autre chose :

```
*monPointeur =

 = *monPointeur
```

Ici il signifiera en premier : "Je veux affecter(=) une **valeur** à l'espace/zone mémoire se trouvant à la **valeur** de mon pointeur" .

En deuxième par contre il signifiera : "je veux(=) la **valeur** de l'espace/zone mémoire se trouvant à la **valeur** de mon pointeur ".

Dans les deux cas nous nous adressons à une zone mémoire directement et c'est grace à ce petit '*' à l'utilisation et la **valeur** (adresses de variables) de nos pointeurs que nous allons pouvoir manipuler ces espaces/zones.

Donc dans notre exemple :

```
*unParamPointeur1 = 16;
```

Nous disons : "Affecte (=) la **valeur** 16 à la zone mémoire pointée par la **valeur** de mon pointeur "unParamPointeur1" ".

En second dans notre exemple :

```
*unParamPointeur2 = *unParamPointeur1+4;
```

Nous disons : "Dans la zone mémoire pointée par la **valeur** de mon pointeur "unParamPointeur2", affecte (=) lui la **valeur** située dans la zone mémoire pointée par la **valeur** de mon pointeur "unParamPointeur1" et incrémente (+) cette **valeur** pointée de 4."

L'équivalent dans la fonction "loop" si on avait manipulé directement les variables par leur nom et non via leur adresse dans la fonction "maFonction":

```
uneVariable1 = 16;

uneVariable2 = uneVariable1+4;
```

=

```
 *unParamPointeur1 = 16;

 *unParamPointeur2 = *unParamPointeur1+4;
```

Voici un code Arduino pour tester, vérifier, comparer ce que nous avons vu jusqu'a présent :

```
void setup()
{
  Serial.begin(9600);
  delay(2000);
  int uneVariable1 = 20;
  int uneVariable2 = 0;

  Serial.print("1) uneVariable1 -> ");
  Serial.println(uneVariable1);
  Serial.print("2) uneVariable2 -> ");
  Serial.println(uneVariable2);
  Serial.println("");

  Serial.print("3) &uneVariable1 -> ");
  Serial.println((int)&uneVariable1);
  Serial.print("4) &uneVariable2 -> ");
  Serial.println((int)&uneVariable2);
  Serial.println("");

  Serial.println("5) maFonction(&uneVariable1, &uneVariable2) --> void maFonction(int
*unParamPointeur1, int *unParamPointeur2)");
  maFonction(&uneVariable1, &uneVariable2);
  Serial.println("16) fin de maFonction");
  Serial.println("");
```

```
  Serial.print("17) uneVariable1 -> ");
  Serial.println(uneVariable1);
  Serial.print("18) uneVariable2 -> ");
  Serial.println(uneVariable2);
  Serial.println("");
}

void loop(){}
void maFonction(int *unParamPointeur1, int *unParamPointeur2)
{
  Serial.println("");
  Serial.print("6) unParamPointeur1 -> ");
  Serial.println((int)unParamPointeur1);
  Serial.print("7) unParamPointeur2 -> ");
  Serial.println((int)unParamPointeur2);
  Serial.println("");

  Serial.print("8) *&unParamPointeur1 -> ");
  Serial.println((int)*&unParamPointeur1);
  Serial.print("9) *&unParamPointeur2 -> ");
  Serial.println((int)*&unParamPointeur2);
  Serial.println("");

  Serial.print("10) *unParamPointeur1 -> ");
  Serial.println(*unParamPointeur1);
  Serial.print("11) *unParamPointeur2 -> ");
```

```
  Serial.println(*unParamPointeur2);
  Serial.println("");

  Serial.print("12) *unParamPointeur1 = 16;");
  Serial.println("");
  Serial.println("13) *unParamPointeur2 = *unParamPointeur1+4;");
  Serial.println("");

  *unParamPointeur1 = 16;
  *unParamPointeur2 = *unParamPointeur1+4;

  Serial.print("14) *unParamPointeur1 -> ");
  Serial.println(*unParamPointeur1);
  Serial.print("15) *unParamPointeur2 -> ");
  Serial.println(*unParamPointeur2);
  Serial.println("");
}
```

Piqûre de rappel:

byte* unPointeur; //* *Je déclare une nouvelle variable de type pointeur, la variable pointée sera de type byte.*

unPointeur = &uneVariable //& *je donne pour* ***valeur*** *à un pointeur l'adresse d'une variable.*

= unPointeur // *je demande la* ***valeur*** *d'un pointeur qui est l'adresse d'une variable pointée par celui-ci.*

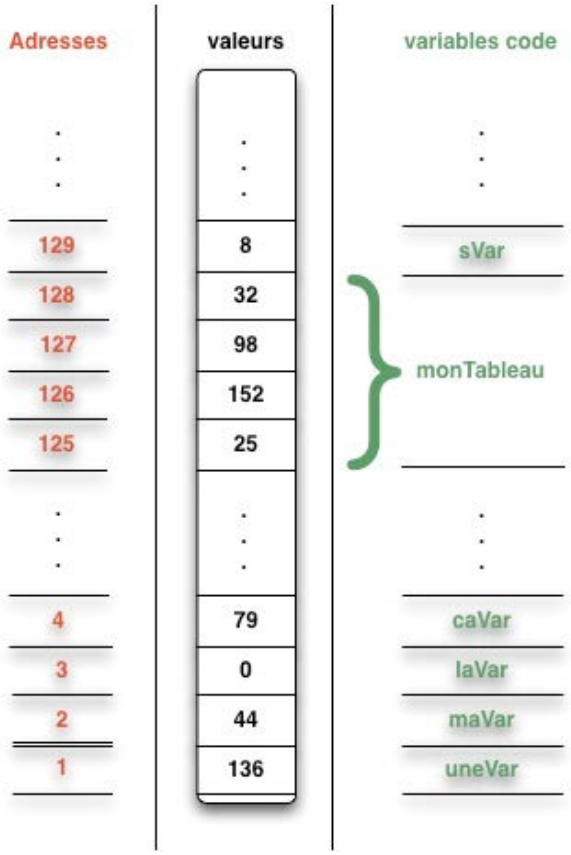
Pointeur = // Je demande l'espace mémoire situé à la **valeur** de mon pointeur (une adresse) pour lui affecter une **valeur** du type pointé.

= *Pointeur //* Je veux affecter la **valeur** du type pointé par la **valeur** de mon pointeur (une adresse).

3. Les tableaux

3.1 description :

Nous avons vu en début de ce tutoriel qu'une variable servait à stocker une **valeur** numérique[1] en mémoire, hé bien les tableaux sont en quelque sorte des super variables qui **permettent** de stocker plusieurs **valeurs** numériques[1] contigues en mémoire.



*Rem: dans la description des variables nous avions vu une variable "varVar" dont la **valeur** prenait 2 emplacements mémoire[3], ici chaque valeur de type byte de notre tableau prend 1 seul espace mémoire.*

*Maintenant si les différentes **valeurs** de notre tableau étaient du même type que notre variable "varVar", ils auraient pris chacun deux emplacements mémoire également donc on double la place prise par notre tableau en mémoire.*

Les tableaux et les pointeurs même combat :

Pour ceux qui espéraient que le chapitre sur les pointeurs était terminé, mauvaise nouvelle on n'en a pas fini avec eux ...

Figurez-vous que les tableaux et les pointeurs ont un gros point commun dans leurs utilisations.

Si vous regardez bien le schéma précédent vous aurez remarqué qu'il y a plusieurs valeurs pour une seule variable.

Mais alors comment récupérer chaque valeur de ma variable individuellement ???

He bien tout simplement par la seule façon de les distinguer individuellement, via l'adresse de chacun de ses espaces/zone mémoire.

Et si je vous disais maintenant que tableau et pointeur sont en quelque sorte la même chose, vous me diriez: "rien à voir puisque nous n'avons pas stocké d'adresse en mémoire ???".

Pour ce faire, à la différence des pointeurs qui stockent l'adresse d'autres variables pour pouvoir les manipuler, ici ce dont nous avons besoin c'est de l'adresse du premier espace/zone mémoire du tableau, les autres étant la suite du premier on peut facilement en déduire leur positions.

Maintenant pour connaître l'adresse du premier élément il suffit de se référer à sa propre adresse, donc à partir de là on peux presque dire qu'un tableau est un pointeur constant sur son premier emplacement mémoire.

Mais attention à bien retenir que tableau et pointeur ce n'est pas pareil, ils fonctionnent selon le même principe, la manipulation d'adresse.

Mais malgré toute leurs similitudes ils ne sont pas la même chose, tout comme un char, un int, un float et autres types de variables ne le sont pas, même si au final ils servent tous à stocker des valeurs numériques[1] en mémoire ...

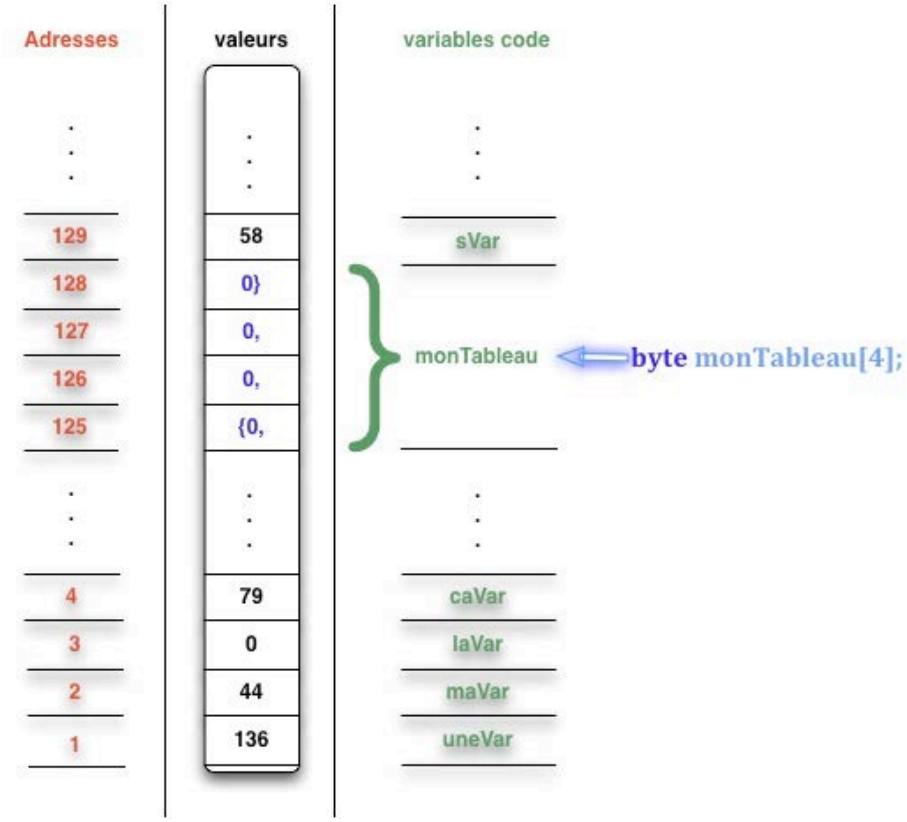
Il ne faut pas oublier que le code que vous écrivez n'est qu'une série d'instructions à destination du compilateur qui lui interpretera votre code pour en faire une série d'instructions (bytecode proche des instructions machine) compréhensible par votre μc .

Cela peut paraître confus mais vous comprendrez mieux par la suite.

3.2 Déclaration et initialisation :

Comme pour toute variable qui se respecte, commencons par la déclaration et initialisation.

```
byte monTableau[4] = {0};

byte monTableau[] = {0, 0, 0, 0};
```



Nous allons commencer avec de nouveaux symboles "[ ] " qui signifient tout simplement: "Je veux créer une variable de type tableau".

Tout comme c'était le cas pour les pointeurs nous devons définir le type pointé par notre variable de type tableau.

Ce n'est pas la variable tableau qui sera de type **byte** mais les valeurs ciblées.

Ensuite comme vous l'aurez remarqué il y a deux façons de déclarer un tableau, mais il faut d'abord savoir quelque chose d'important sur les tableaux: Lorsque nous déclarons un tableau nous sommes obligés d'initialiser sa taille et ce pour la bonne et simple raison qu'il faut réserver les différents espaces/zones mémoire du tableau.

Imaginez que nous déclarons un tableau de taille inconnue et directement après nous déclarons une nouvelle variable, où se situera cette nouvelle variable dans ma mémoire ??? Directement après le premier espace/zone de notre tableau, 150 cases plus loin et ensuite les éventuelles autre case de mon tableau ???

He bien non, il nous faut obligatoirement initialiser la taille de notre tableau pour réserver les espaces/zones de celui-ci.

Ne pas oublier également que cette réservation dépendra du type[3] de variable stockée, qui selon celui-ci pourra prendre un ou plusieurs espaces/zones mémoire par valeur du tableau.

Donc dans ma première déclaration je lui donne directement sa taille entre les deux symboles "[4]", ici je réserve un tableau de 4 espaces/zones de type byte.

Dans le deuxième cas je ne donne pas sa taille "[?]", mais je l'initialise directement à la déclaration en lui donnant les valeurs de mon tableau " = {0, 0, 0, 0}; ", la réservation ce fais donc instantanément et avec les valeurs initialisées également.

Dans nos deux cas ici, il s'agira de tableau statique.

Mais alors comment déclarer un tableau dont la taille ne sera connue qu'à l'execution ?

En fait il y a un moyen très simple de faire cela, tout simplement via une autre variable.

```
int length = 4;

byte monTableau[length] = {0, 0, 0, 0};
```

!!! Selon la version du compilateur, faire ceci sera interdit, d'ailleurs la plus part vous dirons que ce n'est pas recommandé,mais il existe une autre façon de le faire via ce que l'on appelle l'allocation dynamique.

Je n'en parlerai pas ici, le but de ce tutoriel étant de vous montrer le vrai visage des pointeurs et tableaux, mais sachez que cela permet d'allouer (réserver) ou réallouer un ou plusieurs espace/zone mémoire à une variable.

Petit avertissement quand même, lorsque vous entendez ou entendrez parler de "fuites mémoire" c'est qu'il s'agit certainement de zones mémoire allouées manuellement qui n'aurait pas été désallouées (libérées) en fin d'utilisations, donc prudence.

<u>Rem</u>:

1) Dans le premier exemple j'ai également initialisé les valeurs de mon tableau mais ce n'était pas obligatoire comme nous avons fourni directement la taille de notre tableau à sa déclaration. Par contre, comme expliqué dans le point 2.2, par sécurité on devra toujours initialiser ses variables à la déclaration.

2) Vous aurez d'ailleurs remarqué qu'à ma première initialisation, je n'ai mis qu'une seule valeur entre mes accolades "{0}", il s'agit simplement d'un petit raccourci signifiant qu'on initialise tous nos espaces/zone à la ***même valeur****.*

Imaginez si vous aviez dû initialiser un tableau de taille 1000 avec pour valeur 0 à chaques espace/zone, ce raccourci se révelera donc très pratique et économe pour notre clavier et nos petites mimines ...

3.3 Manipulation :

3.3.1 Affecter et lire une valeur de notre tableau :

A l'utilisation vous allez voir qu'il n'y a pas de grande différences avec les pointeurs, peut être plus intuitif pour les tableaux grace à nos petits crochets "[]" qui comme les pointeurs ne signifieront pas la même chose à l'utilisation qu'à la déclaration.

```
monTableau[0] = 20;

uneVariableDuMemeType = monTableau[0];
```

Je pense que la majorité d'entre vous en voyant ce code penseront (à juste titre) que ceci "[0]" signifie : "je veux attribuer la valeur 20 à l'emplacement ou "index" 0 de mon tableau"...

Hé bien ce n'est pas tout à fait exact ...

En fait nos "[]" signifient en réalité : "donne moi l'espace mémoire situé à l'adresse" et le "0" signifie tout simplement : "incrémenté (+) cette adresse de 0".

Maintenant il ne me manque qu'une seule chose ... , quelle adresse ?

C'est là que notre variable de type tableau intervient, ici elle se comportera comme un pointeur, donc l'adresse pointée sera son adresse qui est l'adresse de la première zone mémoire de notre tableau ...

Ils nous suffit donc à présent de fusionner ses phrases pour découvrir la signification complète de "monTableau[0]" : "donne moi l'espace mémoire situé à l'adresse de la première zone mémoire de mon tableau et incrément cette adresse de 0" ...

Simple non ? (tousse)

Tiens, la signification de "[]" ne vous rappelle pas quelque-chose vu récemment ?

Un truc qui nous permettait de manipuler des espaces/zones mémoires via leurs adresses ???

En fait sa signification est exactement la même que "*", la seule différence ici c'est que nous allons incrémenter notre adresse d'une valeur correspondant à l'emplacement désiré par rapport à l'adresse du premier élement de notre tableau.

Rem: Ici vu que nous voulions affecter une valeur au premier emplacement mémoire de notre tableau, celle-ci correspond à l'adresse du premier élement de notre tableau incrémenté de 0, ce fameux 0 qui vous perturbe et que vous avez envie de remplacer par 1.

Il y a de quoi en perdre son latin non ?

Faites le test :

```
void setup()
{
  Serial.begin(9600);
  delay(2000);
  byte tab[] = {1,2,3,4};
  Serial.print("");
  Serial.print("tab :");
  Serial.println((int)tab); //cast en (int) car un type adresse n'est pas de type int  ...
  Serial.print("&tab :");
  Serial.println((int)&tab);
  Serial.print("&tab[0] :");
  Serial.println((int)&tab[0]);

  Serial.println("");

  Serial.print("tab[0] :");
  Serial.println(tab[0]);
  Serial.print("*tab :");
  Serial.println(*tab);
  Serial.print("*tab+0 :");
  Serial.println(*tab+0);
  Serial.print("*&tab[0] :");
  Serial.println((int)*&tab[0]);
}
```

Petit bonus pour le dernier qui est un mélange de tous nos petits symboles, nous demandons en premier la zone mémoire situé à l'adresse de première zone mémoire de notre tableau incrémenté de '0', puis nous demandons l'adresse de cette zone mémoire '&' et en dernier '*' nous demandons la valeur de la zone mémoire située à l'adresse retenue par nos instruction précédente ...

A présent je voudrais affecter ou connaître la valeurs du dernier emplacement de mon tableau de taille 4.

Alors ici si je n'avais pas parler d'incrémentation d'adresse mais plutôt de position dans le tableau, beaucoup ce seraient dit au premier abord, par réflexe ou instinctivement : "puisque j'ai un tableau de taille 4, je n'ai cas demander la case [4] de mon tableau ... non ?" .

Hé bien non, vous auriez eu droit à un beau buzzz comme sur les plateaux d'émissions télés vous signalent une erreur dans la réponse.

Puisque l'adresse de notre premier emplacement tableau se trouve à son adresse incrémentée de 0, pour trouver le suivant il nous suffit d'incrémenter cette adresse de 1 et ainsi de suite ... jusqu'à arriver à notre dernier emplacement qui sera en partant de 0 tailleDeNotreTableau-1 donc 3 pour ce cas-ci.

```
byte monTableau[4] = {0};

uneVariable = monTableau[3] //dernère zone mémoire de notre tableau
```

<u>Rem</u>: Si vous faites l'erreur et que vous dépassez l'espace mémoire alloué à votre tableau le compilateur ne vous le signalera pas !

Par contre vous risquez fort d'obtenir une valeur quelconque et sans doute même la valeur d'une autre variable de votre programme ...

3.3.2 **parcourir un tableau** :

Comme vu précédemment un tableaux est identifié par son adresse de départ, on peux donc facilement en déduire où se situe chaque espaces/zones mémoire par rapport cette adresse de départ.

Maintenant s'il y a bien un intérêt avec les tableaux c'est que grâce à cette succession d'adresses contigues nous allons pouvoir facilement parcourir ceux-ci grâce à une boucle.

```
for(int i = 0; i< 4; i++)
{
   monTableau[i];
}
```

Je pense qu'ici tout est clair nous allons parcourir notre tableau en incrémentant l'adresse de 'i' ...

Tiens mais tu nous as dit qu'une variable pouvait prendre plusieurs emplacements mémoire, donc nous allons devoir incrémenter différemment selon le type accepté par notre tableau non ?

Hé bien non, c'est là qu'intervient le fait qu'on ait déclaré le type de variable que pourra prendre notre tableau, le compilateur étant assez intelligent, il se chargera pour nous de faire la bonne incrémentation.

Comme je l'ai déjà dit, il faut voir le code comme une multitude de directives au compilateur pour qu'il sache convertir votre code en "langage machine" au final.

3.3.3 Tableaux et fonctions :

Nous allons maintenant voir comment transmettre un tableau à une fonction.

Ici nous avons un premier problème, comment transmettre un tableau de plusieurs valeurs à une fonction ?

Si vous faite le lien avec ce que l'on a vu sur les pointeurs, il vous sera facile de deviner comment transmettre une tableau en paramètre d'une fonction.

Nous allons simplement donner l'adresse de la première zone mémoire de notre tableau à la fonction.

```
void loop()
{
   byte monTableau[4] = {0};
   uneFonction(monTableau);
}


void uneFonction(byte unParamTableau[])
{
   for(int i = 0; i< 4; i++)
   {
         Serial.print(unParamTableau[i]);
   }
}
```

Et puisqu'il s'agit d'une adresse nous pouvons également faire :

```
void uneFonction(int *unParamTableau)
{
}
```

C'est aussi simple que cela pour vous qui manipulez les adresses depuis un moment.

Et pourtant je vois là un problème par rapport aux pointeurs auquel vous risquez d'être confrontés à un moment.

```
for(int i = 0; i< 4; i++)
```

Elle est bien jolie ma fonction mais il y a une chose qu'elle ne peut savoir juste avec l'adresse de mon tableau ?

Quel taille fait mon tableau ?

Ici j'ai pu mettre la taille 4 parce que je la connaissais d'avance, ce qui veux dire également que ma fonction n'est valide que pour des tableaux de taille 4 ...

Et si maintenant j'avais besoin de ma fonction pour d'autres tableaux de taille différentes ?

En fait à cette question il n'y a qu'une seule réponse, il nous faut ajouter un paramètre pour connaitre la taille de nos différents tableaux dans une fonction.

```
void loop()
{
   byte monTableau[4] = {0};
   uneFonction(monTableau);
}


void uneFonction(byte unParamTableau[], int taille)
{
   for(int i = 0; i < taille; i++)
   {
         Serial.print(unParamTableau[i]);
   }
}
```

Rem:

1) il existe un moyen de connaître la taille d'un tableau grâce à l'opérateur sizeof qui est capable de donner la taille (nb espaces/zones alloués) prise en mémoire d'un type de variable.

Par exemple pour l'arduino une variable de type int donnera 2 emplacements occupés en mémoire, pour un char il donnera 1 et pour un tableau de 5 int il donnera 10, ...

Maintenant lorque vous transmettez un tableau en paramètre de fonction vous le faites via son adresse à un paramètre de type pointeur. Or, si vous demandez à sizeof de donner sa taille en mémoire il vous donnera la taille de ce pointeur et non de notre tableau ...

Au final je ne recommande pas d'utiliser sizeof pour connaître la taille de votre tableau mais plutôt d'utiliser une constante ou simple variable que vous initialiserez à la taille de votre tableau au moment ou vous codez puisque celui-ci sera statique.

Entre ( ) sizeof est un opérateur uniquement utile (directive au compilateur) à la compilation, inexistant à l'exécution donc en cas de tableau dynamique ce n'est même pas la peine.

2)Il existe également un type de tableau pour lequel nous pouvons omettre de spécifier la taille.

Il s'agit des chaînes de caractères qui doivent toujours se terminer par un '\0' et donc, on peux facilement en déduire où se situe la fin de notre tableau en le parcourant (je vous en parlerai un peu plus tard).

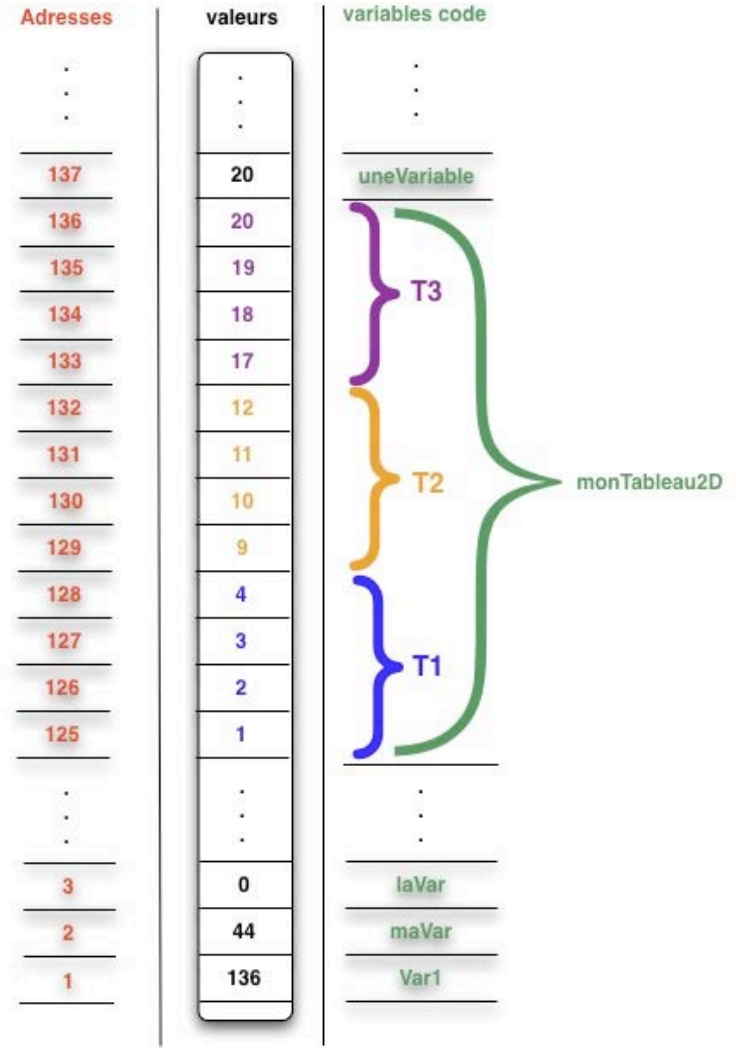
3.4 plusieurs dimensions :

3.4.1 Description :

Figurez-vous que nos tableaux peuvent avoir plusieurs dimensions et ce n'est pas de la science-fiction.

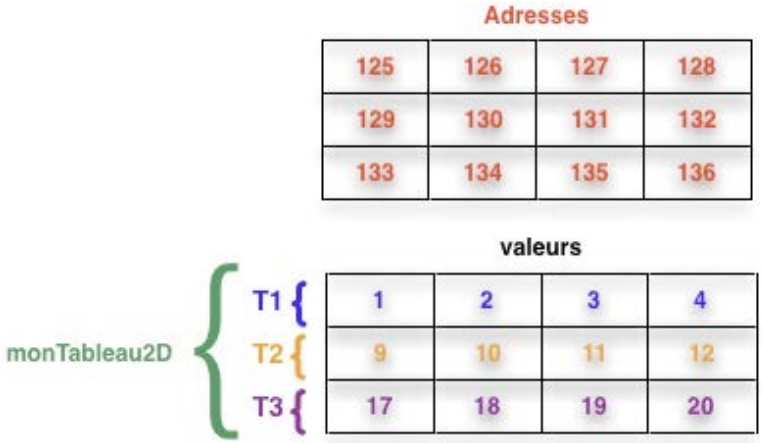
En fait un tableau à plusieurs dimensions est tout simplement un tableau permettant de stocker ... des tableaux de même type ...

Voici une représentation schématique pour vous montrer à quoi peut ressembler un tableau à 2 dimensions en mémoire.



Si nous regardons attentivement ce schéma nous remarquons que notre variable de type tableau "**monTableau2D**" représente plusieurs tableaux contigus de **même taille** en mémoire, sa première valeur de type tableau de byte se situe à la même adresse et que la première valeur de ce dernier .

Pour mieux comprendre l'intérêt d'un tableau multidimensionel on va le schématiser sous forme de grille ou matrice; chaque tableaux sera représenté par une ligne et disposé en colone l'un par rapport à l'autre.



On peux y voir tout de suite l'utilité pour un jeu de dame par exemple, touché-coulé, matrice, etc …

En fait un tableau n'étant qu'un simple type de variable il nous suffit d'appliquer les même règles que vues précédemment pour un simple tableau à notre variable "**monTableau2D**" et de faire de même pour chacune des ses valeurs qui sont également du type tableau.

3.4.2 Déclaration et initialisation :

Pour la pratique nous allons commencer tout naturellement par déclarer et initialiser notre tableau à 2 dimensions.

```
byte monTableau2D[3][4] = { 0 };

byte monTableau2D[3][4] = { {1,2,3,4} , {9,10,11,12} ,{17,18,19,20} };
```

Alors ici rien de compliqué, nous déclarons un tableau de taille [3] et chacune de ses 3 valeurs sera un tableau de taille [4] , nous remarquerons que nos trois tableaux seront de tailles identiques et les valeurs de mes 3 tableaux de type "byte".

Nous allons également initialiser notre variable ainsi que chacune de ses trois valeurs de type tableau, pour ce faire il nous suffit d'initialiser nos trois tableaux en un coup ou indépandamment et les fournir à notre variable tableau comme on l'aurait fait avec n'importe quel autre type de variable.

Tiens,on ne peux pas faire ?

```
byte monTableau2D[][] = { {1,2,3,4} , {9,10,11,12} ,{17,18,19,20} };
```

Hé bien non, cela aurait été trop simple pour pas changer ... vous aurez droit à un beau :

declaration of 'monTableau2D' as multidimensional array must have bounds for all dimensions except the first

Sachez que hormis la première dimension, nous sommes obligés de spécifié la taille des autres .

Donc ceci est valide :

```
byte monTableau2D[][4] = { {1,2,3,4} , {9,10,11,12} ,{17,18,19,20} };
```

3.4.3 **parcourir un tableau multidimensionnel** :

Maintenant nous allons voir comment parcourir notre tableau multidimensionnel pour affecter ou retrouver nos valeurs.

Comme nous l'avons vu pour nos simple tableaux, il suffisait de spécifier l'incrémentation de notre espace/zone mémoire par rapport à l'adresse de départ de celui-ci.

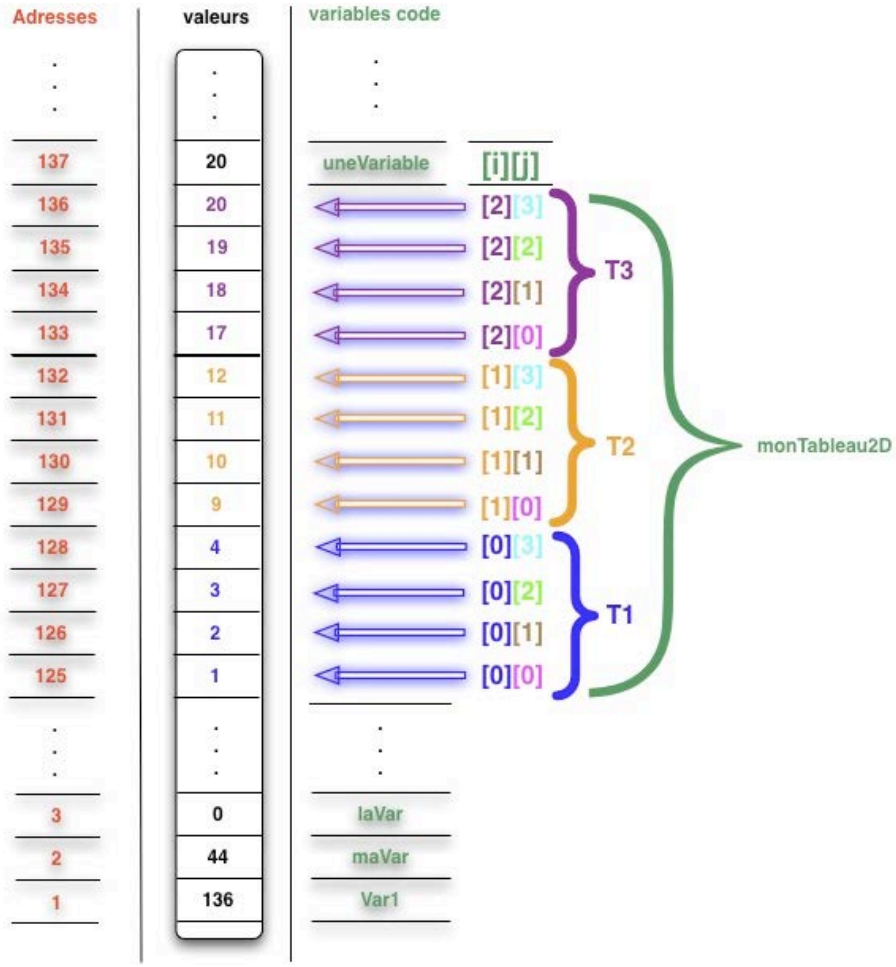
```
monTableau2D[0]
```

Donc une fois nos tableaux récupérés, il nous suffira de spécifier l'incrémentation de notre espace/zone mémoire par rapport à l'adresse de départ de ceux-ci.

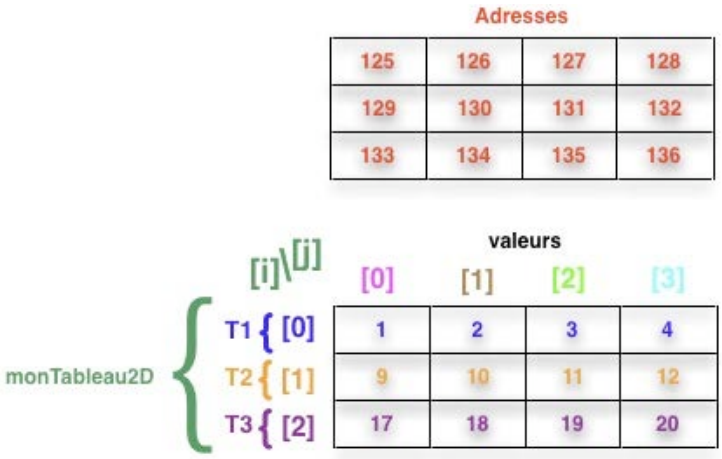
```
monTableau2D[0][0]
```

C'est aussi simple que ça.

```
for(int i = 0; i < 3; i++)
{
   for(int j = 0; j < 4; j++)
   {
     tab2D[i][j];
   }
}
```



Vue sous un autre angle (grille, matrice, ...).



Pour parcourir simplement nos tableaux afin de lire ou affecter chaque valeurs consécutivement nous allons les parcourir ligne par ligne comme représenté sur le 2ème schéma.

Nous avons donc une première boucle afin de récupérer chaque tableau individuellement et à chaque tableau ainsi récupérer "[i]" nous allons les parcourir "[j]" dans une deuxième boucle imbriquée dans la première.

3.4.4 Tableaux multidimensionnel et fonctions :

Alors ici ça se complique, je vais directement vous montrer ce que l'on peut faire et ce que l'on ne peut pas faire.

```
void uneFonctionT(byte unParamTableau2D[3][4])
```

```
void uneFonctionT(byte unParamTableau2D[][4])
```

```
void uneFonctionT(byte unParamTableau2D[][])
```

On remarquera que, comme pour l'initialisation, nous somme obligés à minima de spécifier la taille de notre deuxième dimension ...

Nous remarquons également que comme pour les simples tableaux le fait de devoir spécifier la taille de nos tableaux directement en paramètre diminue l'intérêt de nos fonctions ...

Pour palier à se (ses) problème(s) il existe deux solutions.

La première serait de transmettre chaque tableau individuellement à notre fonction et l'utiliser comme ont l'aurait fait pour un simple tableau.

```
byte monTableau[3][4] = {{1,2,3,4}, {9,10,11,12}, {17,18,19,20}};

for(int i = 0; i < 3; i++)

{

   uneFonction(monTableau[i]);

}

void uneFonction(byte unParamTableau[], int taille)

{

   for(int i = 0; i < taille; i++)

   {

         unParamTableau[i];

   }

}
```

Ici nous allons manipuler nos tableaux indépendamment comme de simples tableaux, donc nous pourrons utiliser n'importe quelles fonctions pouvant manipuler un simple tableau.

La deuxième solution c'est de travailler à partir de l'adresse de notre tableau multidimensionnel.

```
 byte monTableau[3][4] = {{1,2,3,4}, {9,10,11,12}, {17,18,19,20}};
 maFonction(*monTableau, 3, 4); //ou maFonction(monTableau[0], 3, 4);


void maFonction(byte *unParamTableau2D, int tailleC, int tailleL)
{
   for(int i = 0; i < tailleC; i++)
   {
      for(int j = 0; j < tailleL; j++)
      {
         uneVarDeTypeByte = *(unParamTableau2D+((i*tailleL)+j)));
         *(unParamTableau2D+((i*tailleL)+j)) = (i*tailleL)+j);
      }
   }
}
```

Ce que nous fesons ici peut parraître compliqué mais en fais tout ce que nous faisons c'est transmettre l'adresse de notre tableaux et parcourir toutes les adresses de notre tableau multidimensionnel successivement grâce à un calcul "simple".

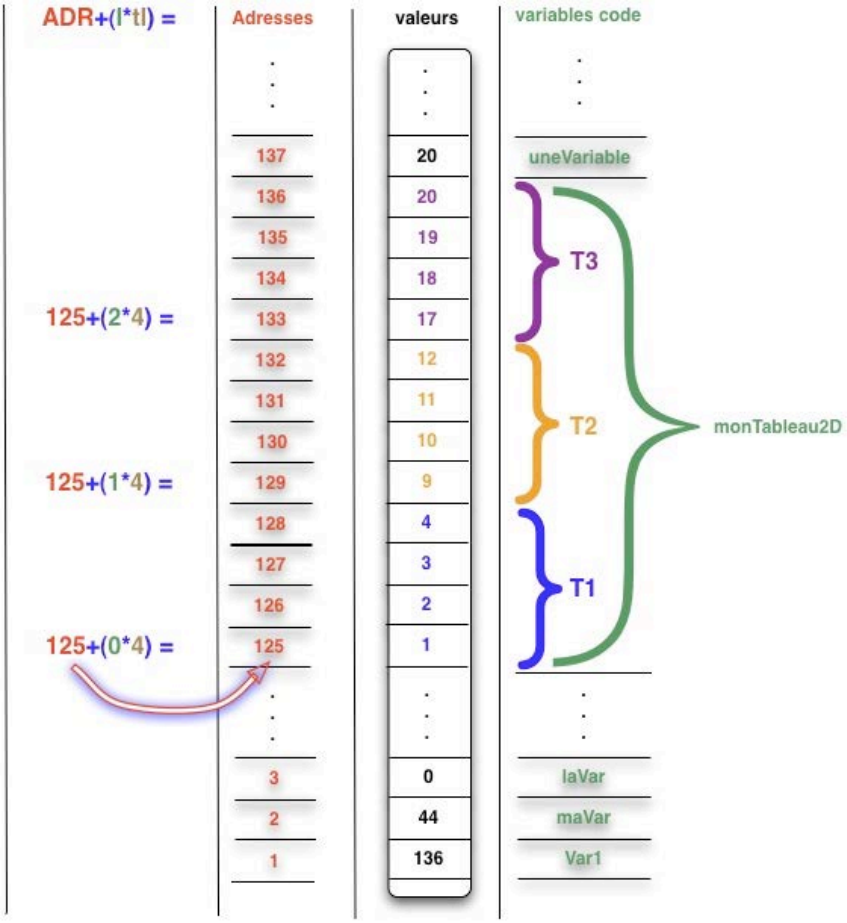
```
unParamTableau2D+((i*tailleL)+j)
```

Petite explication sur ce calcul "simple".

Comme nous l'avions fait avec notre boucle imbriquée nous allons récupérer l'emplacement de nos différents tableaux 'i', mais contrairement à notre parcours par incrémentation "[i]" en dehors d'une fonction le compilateur ne connaissant pas la taille de nos différents tableaux, il ne sait pas où ils se situent l'un par rapport à l'autre, c'est donc à nous de le faire.

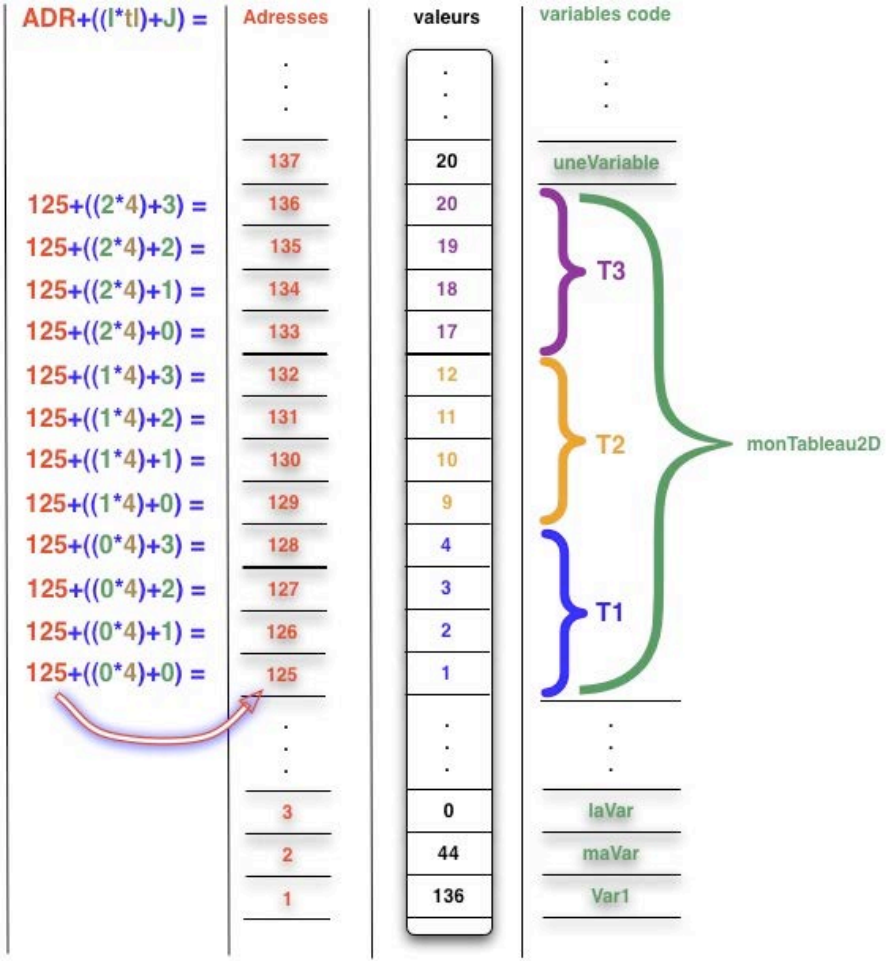
Il existe un moyen très "simple" de connaître l'emplacement de chacun nos tableaux en mémoire, si on regarde bien nos schémas et puisque nous connaissons la taille (identique) de nos différents tableaux on peux facilement déduire ou se situe l'adresse de chacun d'eux.

unParamTableau2D+(i*tailleL)



Maintenant que nous savons ou se situe chacun des tableaux dans notre mémoire, nous pouvons facilement en déduire chaque valeur en incrémentant l'adresse de chacun d'eux.

```
unParamTableau2D+((i*tailleL)+j
```



Une fois chaque adresses récupérées, il nous suffit plus que d'y appliquer notre chère petit symbole '*' .

```
*(unParamTableau2D+((i*tailleL)+j))
```

Rem: Comme dit précédemment le fait d'avoir spécifié le type de variable pointée permettra au compilateur de faire les bonne incrémentation selon le type de nos tableaux.

Hé bien voilà, que rajouter d'autre ?

Ah si, il reste encore un moyen de faire la même chose.

Nous pouvons également parcourir toutes nos valeurs en un coup car même s'il s'agit de plusieurs tableaux ils se suivents successivement en mémoire comme de simple tableaux.

```
byte monTableau[3][4] = {{1,2,3,4}, {9,10,11,12}, {17,18,19,20}};

maFonction(*monTableau, 3*4);

void maFonction(byte *unParamTableau2D, int tailleT)

{

   for(int i = 0; i < tailleT; i++)

   {

        uneVarDeTypeByte = *(unParamTableau2D+i);

        *(unParamTableau2D+i) = i;

   }

}
```

Nous allons donc parcourir notre tableau multidimensionnel comme s'il sagissait d'un simple tableau, pas besoin de schéma pour comprendre.

Bon d'un côté cela réduit tout l'intérêt des tableaux multidimensionnels, mais c'est pratique pour par exemple réinitialiser toutes les valeurs de nos tableaux, etc ...

Rem: Ici pour l'exemple nous avons travaillé avec un tableau à 2 dimensions, mais il est possible de faire la même chose avec plus de dimensions, 3D (imaginé un cube), 4D (imaginé euh ...), ...

3.4.5 Tableaux de tailles différentes ? :

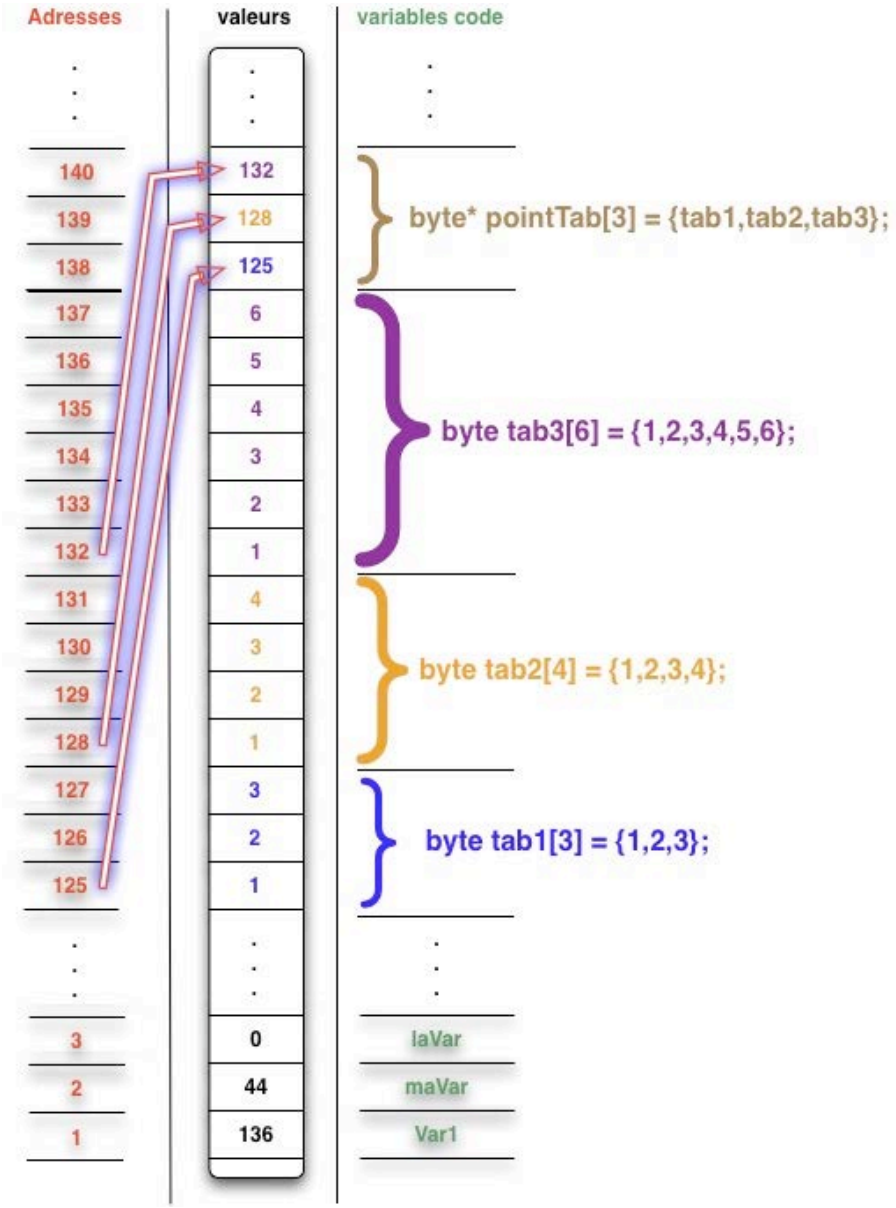
Comme je l'ai spécifié au sujet des tableaux multidimensionnels, nos différents tableaux représentés dans ceux-ci doivent obligatoirement posséder un taille identique.

Maintenant imaginons que vous ayez besoin d'un tableau de tableaux toujours de même type mais de taille différentes, comment outrepasser cette restriction ?

Il existe pourtant une solution "simple", il nous suffira tout simplement de créer un tableau de type pointeur de variable de type tableaux dont chacune de ses valeurs de type tableau seront indépendantes l'une de l'autre.

```
void setup()
{
   byte tab1[3] = {1,2,3};
   byte tab2[4] = {1,2,3,4};
   byte tab3[6] = {1,2,3,4,5,6};

   byte* pointTab[3] = {tab1,tab2,tab3};
   int taillesTabs[3] = {3,4,6};
   maFonction(pointTab, 3, taillesTabs);
}
void loop(){}
void maFonction(byte* unParamPointTab[], int nbTabs, int paramTaillesTabs[])
{
   for(int i = 0; i < nbTabs; i++)
   {
      for(int j = 0; j < paramTaillesTabs[i]; j++)
      {
         *(*(unParamPointTab))+j;
```

```
        }
    }
}
```



Alors ici je vais vous laissez vous débrouiller tout seul, il suffira de rassembler tout ce que l'on a apris jusqu'a présent et de l'appliquer, cela fera également un bon exercice de révision.

4 Arithmétique de pointeurs :

Je vous en ai parlé dans le chapitre sur les pointeurs, vu qu'il s'agit de simples variables et qu'une variable permet de stocket des valeurs numériques[1] en mémoire nous allons pouvoir y appliquer certaines opérations arithmétiques telles qu'une addition ou soustraction.

Par contre puisque nos valeurs sont de type adresse mémoire et non de type entier tel que "int", il ne sera pas autorisé d'y appliquer n'importe quel opérateur arithmétique, nous pourons y additioner ou soustraire un entier, affecter une adresse, faire des comparaisons entre adresses (<,>, ==, ...).

4.1 Addition et soustraction (+,-) :

Alors ici je ne vais pas vous apprendre grand-chose pour la simple et bonne raison que nous avons déjà pratiqué une de ces opérations et récemment en plus ...

Il s'agit tout simplement d'additioner un entier au pointeur, comme nous l'avons fait avec nos tableaux et adresses, je suppose que je n'ai pas besoin de vous démontrer l'intérêt de la chose ?

Maintenant pour la soustraction d'un entier à un pointeur, l'utilité n'est pas différente de l'addition nous aurions très bien pu lire nos tableaux à partir de leur dernière adresse par exemple il suffira juste de modifier la(les) boucle(s) en décrémentant nos adresses d'un entier ...

Une chose à savoir néanmoins, c'est que nous ne pouvons pas additionner deux pointeurs pour la simple et bonne raison que c'est totalement inutile, comme le fait de multiplier ou diviser un pointeur, donc le compilateur rejetera ces opérations.

Par contre il est tout à fait autorisé de soutraire deux pointeurs, ce qui peux être utile pour connaître la taille d'un tableau (adresseFin-adresseDebut) par exemple, il faudra juste bien faire attention que le résultat entre bien dans le type de variable affecté, si nous avions un tableau de taille supérieure à 255 par exemple, le résultat ne poura pas entrer dans une variable de type byte limité au nombre de 0 à 255 ...

Rappel : Lorsque l'on fait une addition ou soustraction d'un entier à un pointeur, la valeur incrémentée ou soustraite est proportionnelle au type de variable pointée.

Par exemple une variable pointée de type int (2 octet en mémoire), si nous incrémentons de 1, c'est 1 fois le type de variable pointé donc nous incrémenton de 2 en réalité.

On revoit encore ici l'utilité de spécifier le type de variable pointée.

4.2 **Affectation** (=) :

Ici je ne ferais que deux remarques ou plutôt rappels:

1) Lorsqu'on affecte un pointeur à une autre pointeur il faut qu'il soit tous deux du même type obligatoirement !!!

2) On ne peux affecter une variable de type entier (byte, char, int, ...) à un pointeur, ceux-ci prenant pour **valeur** une adresse et non un entier, même si ça peut paraître similaire puice qu'au final ce sont des nombres.

Chaque type de variable à sa place, il ne faut pas les mélanger (un cast est toléré par contre) ...

4.3 **comparaison** (<,>, ==, ...) :

Figurez-vous que nous pouvons comparer nos pointeurs pour savoir si la valeur (l'adresse) de l'un est avant l'autre (<,>) ou s'il pointe la même adresse, etc ...

La seule obligation ici, c'est que les deux pointeurs comparés doivent pointer le même type de variable !

5. Les chaîne de caractères :

Nous allons parler ici de tableaux un peu spéciaux, il s'agit des chaînes de caractères, les mêmes que vous avez sous les yeux en ce moment.

Je dois juste donner une petite précision qui a son importance pour ceux qui l'ignoreraient.

Les chaînes de caractères servent à stocker des caractères contigus en mémoire ça on le sait, par contre il faut bien comprendre qu'un caractère est représenté en mémoire par une valeurs numérique[1] qui sera sa correspondance ASCII et non 'a', 'b, 'c', ...

Par exemple pour le caractère 'g' sa valeur décimale en mémoire sera 103, en majuscule 'G' ce sera 71, pour le caractère '1' ce sera 49, pour le '2' sera 50, etc ...

Je vous propose d'aller consulter le tableau de correspondance ici.

http://fr.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange .

Rem :

1) Un char peux prendre pour valeur des nombres allant de -128 à 127, mais il existe une extension de la table ascii (peut différer d'un système à un autre) , donc pour pouvoir atteindre ses autres caractères il nous faudra passer notre char en unsigned char(char non signé) permettant d'avoir des valeurs allant de 0 à 255 .

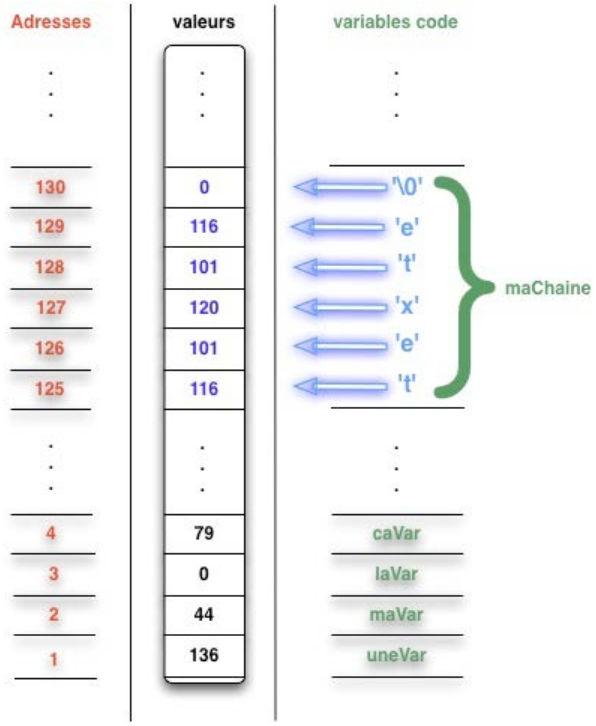
2) Il existe également d'autres normes d'encodage des caractères tels que utf (1 à 4 octet), unicode (sur 2 octet) ... qui permettent plus de possibilités que l'ascii, mais conservent la même correspondance pour les 127 premier caractères.

Cette petite parenthèse effectuée, parlons à présent de ces fameuses chaînes.

Une chaîne de caractère en programmation n'est rien d'autre qu'un tableau de type char avec une petit particularité en plus, c'est qu'elle se termine **toujours** par un caractère de fin de chaîne, qui est '\0' ou 0 en décimal.

Donc lorsque vous formez votre chaîne, il ne faut jamais oublier le petit '\0', se qui signifie également que la taille de notre chaîne sera toujour de nb caractères+1.

Par exemple la chaîne "texte" n'est pas de 5 caractères mais 6!



Bon maintenant cette particularité peut vous paraître inutile, comme un boulet que l'on traîne au bout de nos "chaînes" (jeux de mot), et pourtant elle nous sera bien utile, car grâce à elle nous pourrons facilement identifier la position du dernier emplacement de notre tableau, donc il sera simple d'en déduire sa taille.

5.1 Déclaration et initialisation :

```
char maChaine1[] = "texte";

char maChaine2[] = {'t','e','x','t','e','\0'};

char maChaine3[] = {116,101,120,116,101,0};

char maChaine4[6] = "";

char maChaine5[6] = {0};
```

Comme on peux le voir il y a plusieurs façons d'initialiser nos chaines de caractère.

On remarquera que via une chaine constante "texte", le '\0' sera comptabilisé automatiquement par le compilateur, nous remarquons également que nous pouvons fournir directement la valeur décimale de correspondance ascii, en revanche pour les autres rien de bien différent par rapport aux autres type de tableaux .

5.2 Parcourir une chaîne de caractères et fonction :

A présent il ne nous reste plus qu'a parcourir nos chaînes et les manipuler via une fonction.

Pour cet exemple je vais faire les deux directement.

```
void setup()
{
   char maChaine[] = "texte";
   int taille = tailleDeMaChaine(maChaine);
}
int tailleDeMaChaine(char *unParamChaine)
{
   int i = 0;
   char c = ' ';
   while(c != '\0')
   {
      c = unParamChaine[i];
      i++;
   }
   return i;
}
```

Alors ici je viens de faire une des fonctions les plus courantes pour ce qui concerne les chaînes de caractères: celle qui retourne la taille de notre chaine.

Comme vous le remarquez je n'ai pas spécifié la taille de mon tableau en paramètre puisque ma boucle sera interrompue grace à notre "signal" de fin de chaîne '/0' et comme pour n'importe quel tableau je l'ai parcouru via l'incrémentation "[i]" de son adresse.

Il ne nous reste plus qu'à retourner le résultat.

Voilà nous pouvons dire merci d'être forcer d'ajouter ce petit caractère '/0' (0) de fin de chaine .

Rem: La valeur retournée sera la taille de notre chaine, '\0' compris.

6) <u>FIN</u> :

Voilà nous en avons fini avec ce tutoriel, à présent la mémoire, les pointeurs et tableaux n'auront plus de secret pour vous ... Je dirais testé tout nos petit symboles *, &, [], ... et faite des découvertes.

En tout cas j'espère ne pas avoir été trop fouillis, et que ce tutoriel vous aura semblé agréable malgré le sujet difficile.

[1] Il faut savoir qu'à la base votre ordinateur, microcontrôleur et autres systèmes électroniques numériques ne sont pas capable de grand-chose, tout ce qu'ils savent faire c'est manipuler (opérations logiques) des nombres sous formess binaires (base2) représentés symboliquement par les chiffres 0 et 1.

Par contre grâce à ces nombres et leurs manipulations nous pouvons représenter une infinité de choses, un poids, un coût, un caractère de notre alphabet (char), un déplacement, une information analogique, une limite, une adresse, etc, ...

Rem: même lorsque vous manipulez des variables de type char, ce sont bien des nombres (voir correspondance ascii) que vous stockez en mémoire et non des caractères directement.

[2] Les variables peuvent être soit globales, soit locales:

- globale : Déclarées en dehors de tout bloc* généralement déclarées au début de votre code, leurs existences durera tout le long de l'éxecution de votre programme.

Elles seront visibles dans tout le code.

- locales: leurs existences sera courte, juste le temps d'éxecution du bloc* dans lequel elles auront été déclarées, créées.

Elles ne sont visible que dans le bloc* dans lequel elles auront été déclarées, créées.

* Un bloc en programmation sont des portions de codes appartenant à une condition, boucle, fonction, etc, … Un bloc est repérable et généralement délimité par deux accolades ouvrantes et fermantes "{ }" (Ex: if(){}, while(){}, for(){}, fonction(){}, ...).

[3] Une variable peut prendre un ou plusieurs espaces en mémoire, selon son type (entier, réel, caractère, ...), langage ou plateforme utilisée.

[4] Le nom c'est vous qui le choisissez. Il doit être unique, tout comme le sera son adresse.

!!! Une fois compilé votre code sera traduit dans un language de plus bas niveau (assembleur, byte code, language machine, ...), le nom que vous fournirez ne sert qu'à faciliter l'écriture et lecture dans un language compréhensible par le commun des mortels. Une fois votre code compilé, votre nom de variable n'existera plus.

Donc ne pas essayer de manipuler votre variable via une chaîne de caractères (char[], String, ...), car cette chaîne ne correspondra plus au nom de la variable donné dans votre code.

[6] Sachez qu'il n'y a pas que les variables qui possèdent une adresse. Toutes données stockées en mémoire en ont une, donc une fonction peut avoir une adresse, une structure, un objet, ...