UTILISATION DES POINTEURS.

Objets particuliers du langage C ils permettent de traiter des variables par utilisation de leurs adresses en mémoire. Un pointeur a pour contenu l'**adresse d'un objet C typé**. Ils permettent de :

- Définir des structures dynamiques, *(Qui évolue au cours du temps)* par opposition aux tableaux, par exemple, qui sont des structures de données statiques dont la taille est figée à leur déclaration.
- De manipuler de façon simple des données de tailles importantes. *(Au lieu de passer à une fonction un élément "volumineux" on pourra par exemple lui fournir un pointeur vers cet élément ...)*
- Les tableaux ne peuvent stocker qu'un nombre fixé d'éléments de même type. En stockant des pointeurs dans les cases d'un tableau, il sera possible de **stocker des éléments de tailles diverses**, et même de rajouter des éléments au tableau en cours d'utilisation. *(Tableaux dynamiques étroitement liée à l'usage de pointeurs)*
- Avec les pointeurs il devient possible de coder des fonctions qui retournent plusieurs valeurs ce qui n'est pas faisable avec return.
- Ils sont parfaitement adaptés pour créer des structures chaînées.

Gestion dynamique de la SRAM sur Arduino :

Permettant un appel récursif des procédures et des fonctions, le langage C engendre pour le compilateur des contraintes spécifiques relatives à la gestion de la mémoire vive. Le sectionnement et la répartition des données statiques et des données dynamiques en sont forcément directement influencés avec une allocation dynamique de certaines données. Il en résulte des risques de "collision" entre les données dynamiques *(HEAP)* et la zone occupée par la PILE.

Fonctionnement de la mémoire vive.

La mémoire vive *(256 + 2Ko)* est généralement divisée en 4 zones :

- Les 256 premiers octets pour les registres généraux du microcontrôleur *(Représentée en jaune sur la Fig.1)* occupent "le bas" de la SRAM.
- La zone nommée BSS qui contient toutes les variables globales, allouées statiquement au moment de l'édition de lien lors de la compilation. La BSS est utilisé par de nombreux compilateurs pour désigner une zone de données contenant les variables statiques déclarées dans les initialisations, et forcées avec des octets à zéro.
- Le TAS (Nommé HEAP) est destiné aux allocations dynamiques dans lequel on peut attribuer et libérer des blocs de mémoire. Le TAS se fragmente généralement au cours de l'évolution du programme, avec un risque notable de le rendre inutilisable. *Défragmenter HEAP par une séquence de code de type "Ramasse miettes" est faisable mais relativement dangereux, car si l'on déplace une variable en cours d'utilisation, les conséquences peuvent s'avérer ingérables.*

L'utilisation de la mémoire dynamique doit être réduite au minimum, et si possible uniquement durant les phases d'initialisations quand on peut vérifier qu'il y a un risque de manque de mémoire. Noter que la classe String est principalement basée sur l'allocation dynamique. Utiliser String conduit à fragmenter très rapidement la zone HEAP.

- La PILE nommée STACK qui mémorise temporairement :
  * Les paramètres associés à l'appel des fonctions et procédures,
  * Les adresses de retour des fonctions et procédures,
  * Les variables locales aux fonctions et procédures.

La pile est une zone de mémoire commençant en haut de la SRAM qui se charge vers le bas de façon linéaire et continue lors des appels des fonctions ou des procédures. Elle se réduit vers le haut lors des retours.

La disposition standard de la SRAM montrée en Fig.1 consiste à placer les variables de données au début de la mémoire vive interne, suivie de la BSS. Le TAS disponible HEAP pour l'allocateur de mémoire dynamique sera placé juste après la BSS. Ainsi, il n'y a pas de risque d'écrasement entre la mémoire dynamique et les variables SRAM. Le TAS et la PILE peuvent toutefois se heurter même si les espace exigés pour les variables dynamiques ne sont pas exagérés, mais que l'allocation de mémoire se fragmentant au fil du temps. De nouvelles demandes ne peuvent alors plus se loger dans les "trous" des régions déjà libérées, ou par une plongée de la PILE issue d'une fonction avec beaucoup de variables locales. Enfin des appels récursifs d'une fonction ou d'une procédure peuvent conduire à une plongée dans la PILE.

La surveillance de la disponibilité de place dans la SRAM interne au microcontrôleur est un impératif si l'on ne veut pas risquer une collision des variables dynamiques avec la zone de la PILE.
Il importe donc de surveiller, à divers stades stratégiques du code, la quantité de Mémoire disponible au cours du déroulement d'un programme. Le logiciel Verif_SRAM_disponible.ino montre comment quelques instructions permettent de surveiller le risque de collision entre TAS et PILE. Noter que la SRAM disponible pour les données fait bien 2048 octets. *(Mise en évidence par la zone violet pastel sur la Fig.1)* Elle commence en 0100, et se termine donc en 08FF. Les divers pointeurs présentent des identificateurs réservés par le compilateur C d'Arduino et n'ont pas besoin d'êtres déclarés. Noter que les diverses adresses sur la Fig.1 *(Sauf 01FF et 08FF)* sont fonction du logiciel en cours.

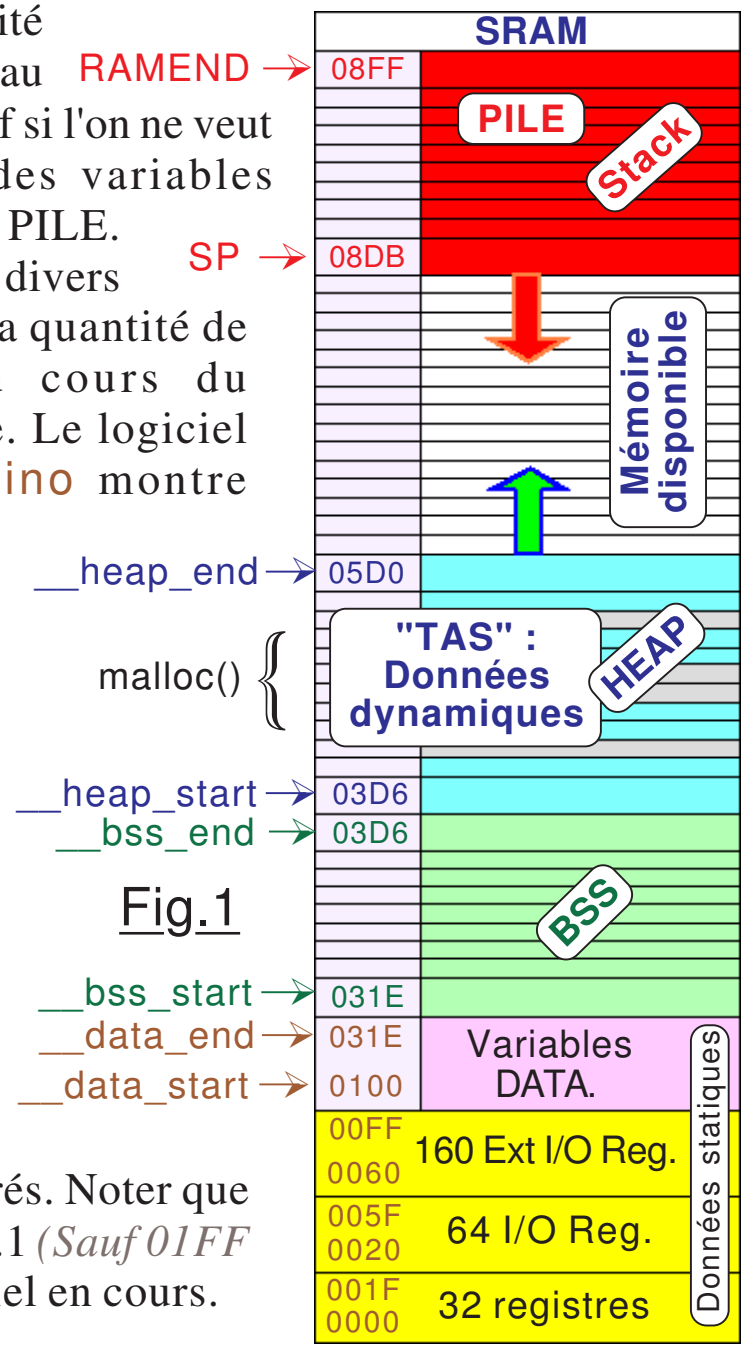


Fig.1

Utilisation basique des POINTEURS :

Déclaration d'un pointeur.

type *Nom_du_pointeur = NULL;

Le symbole '*' précise au compilateur que Nom_du_pointeur est un pointeur qui ciblera une variable dont la taille en octets est défini par type. Le type de variable pointée peut être aussi bien un élément primaire *(int, char, float, String ...)* qu'une structure complexe.

Deux variantes sont équivalentes pour déclarer un pointeur :

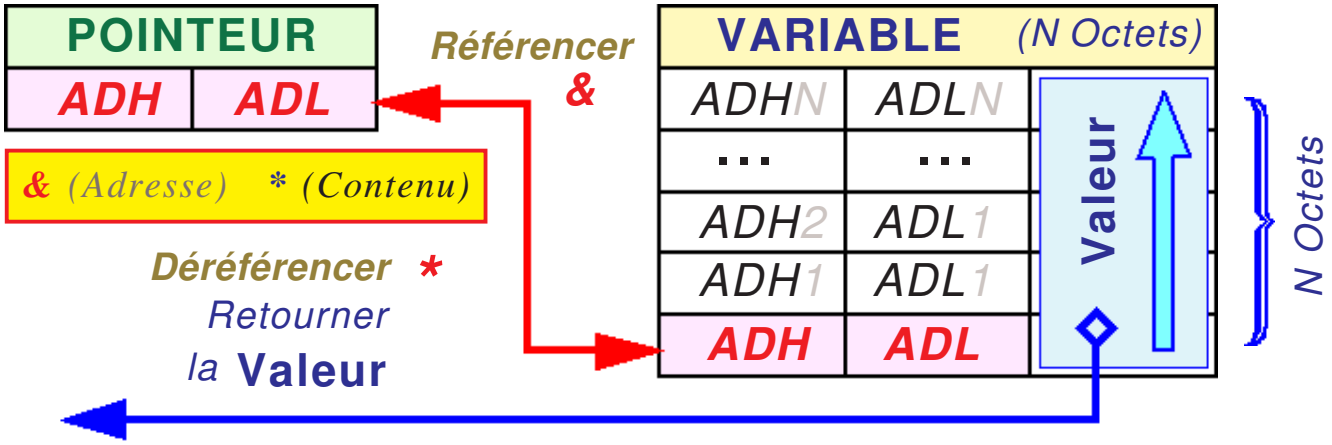
type* Nom_du_pointeur ou type *Nom_du_pointeur

Un pointeur non initialisé présente un danger potentiel car il pointe n'importe où dans la mémoire. Si l'on affecte une valeur à une cellule mémoire à l'aide de ce pointeur, on peut fort bien écraser une donnée ou du code programme. L'option = NULL n'est pas obligatoire mais précise au compilateur que Nom_du_pointeur n'a pas encore de cible. *(Cette affectation est fortement recommandée)*

Un pointeur stocke une adresse mémoire. (Référence) et peut être considéré comme un nombre allant de 0 à (TailleMémoire - 1) du microcontrôleur considéré. Un pointeur occupe donc toujours la même taille quel que soit le type de l'objet indexé. Il s'agit en général de la plus grande taille directement gérable par le processeur : Sur une architecture 16 bits comme celle d'Arduino utilisant un ATmega328 la taille des pointeurs est de 2 octets. Sur une machine 32 bits elle passe à 4 octets, avec 64 bits : 8 octets etc.

Référencer / Déréférencer un pointeur.

- Référencer consiste à affecter l'adresse d'une VARIABLE au pointeur.
- Déréférencer consiste à lire le contenu de la VARIABLE pointée.



L'utilisation d'un pointeur se fait en trois phases :
1) Déclarer le pointeur avec le type de sa cible.
2) Placer l'adresse de la cible dans le pointeur.
```
type* Nom_du_Pointeur = &Nom_de_la_Variable;
```
3) Traiter ou utiliser le contenu de la donnée pointée.

Référencer un pointeur sur une variable.

```
Nom_du_pointeur = &Nom_de_la_variable;
```

L'opérateur & concaténé à Nom_de_la_variable **préalablement défini** retourne l'adresse du premier octet de l'objet pointé.

On peut initialiser un pointeur directement lors de sa déclaration :
```
*Nom_du_pointeur= &Nom_de_la_variable;
```

Dans l'écriture précédente nous n'avons pas à connaitre l'adresse de la cible, d'autant plus qu'elle change à chaque lancement du programme puisque le compilateur alloue les blocs de mémoire libres dans le TAS.
On peut si on le désire affecter une valeur d'adresse à convenance :

Référencer un pointeur par une constante.

```
*Nom_du_pointeur = (type*) ADRESSE;
```

Dans cette affectation (type*) est un rappel du type affecté au pointeur. ADRESSE est aussi bien une valeur entière qu'une constante de type int préalablement définie :

```
int DEBUT_LISTAGE = 0x300; // Valeur hexadécimale 0300. @
byte *PTR = (byte*) DEBUT_LISTAGE;
```

Est équivalent à :

```
byte  *PTR = (byte*) 0x300; // Valeur hexadécimale 0300.
```

NOTE : Le langage C++ exige que tout soit rigoureusement typé. Un entier de type int n'est pas de type pointeur. Il faut donc forcer la structure convenable avec la conversion explicite (type*).

Accéder à l'objet pointé. *(Déréférencer)*

```
Identificateur_pour_une_Variable = *Nom_du_pointeur;
```

Ce n'est qu'après avoir déclaré et initialisé un pointeur, qu'il devient possible avec l'opérateur '*' d'accéder à l'objet ciblé en mémoire. Dans cet exemple Identificateur_pour_une_Variable reçoit le contenu de **l'objet typé** pointé par Nom_du_pointeur.

Travail de base avec les POINTEURS :

Référencer un pointeur par la valeur d'un autre pointeur.

```
POINTEUR1 = POINTEUR2;
```

Les pointeurs étant fondamentalement des objets comme les autres, on peut librement leur affecter la valeur d'un autre objet de même type. *(Donc de type pointeur ciblant une donnée de "même taille".*

Déférencer deux pointeurs dans une même instruction.

```
*POINTEUR1 = *POINTEUR2;
```

Dans cette écriture, la variable ciblée par POINTEUR1 reçoit le contenu de la donnée ciblée par POINTEUR2.

ATTENTION : L'instruction ne sera correctement exécutée que si les deux pointeurs sont de même type. Dans le cas contraire la valeur affectée à la variable ciblée sera incorrecte.

Comparer la valeur d'un pointeur dans les structures if.

```
byte *Nom_du_pointeur= NULL;
```

Puis plus avant dans le programme lors du traitement :

```
if (Nom_du_pointeur >= (byte*) 0x400) {Instructions pour true}
```

Dans cette instruction, l'adresse est indiquée directement sous la forme d'une constante hexadécimale.

Mais il est également possible d'utiliser un identificateur préalablement initialisé à une valeur entière de type int. Exemple :

```
int DEBUT_LISTAGE = 0x500; // Valeur hexadécimale 0500.
if (Nom_du_pointeur >= (byte*) DEBUT_LISTAGE) {Instructions}
```

Noter que le type de DEBUT_LISTAGE est int puisque l'adresse est en deux octets sur un ATmega328. Par contre, dans le test le type est byte car il concerne le pointeur Nom_du_pointeur ciblant un OCTET.

Comparer la valeur de la variable ciblée par un pointeur.

Le fait que la variable soit pointée ne modifie en rien son type. Donc la comparaison se fait da façon tout à fait banale. Exemple :

```
if (*PTR == 0) {Serial.print('0');} // Si zéro compléter avec "0".
```

La seule nuance réside dans le fait que l'on précise la variable servant à la comparaison non pas par son identificateur, mais par son pointeur dont l'identificateur est associé à "*". *(Retourne le contenu de la cible)*

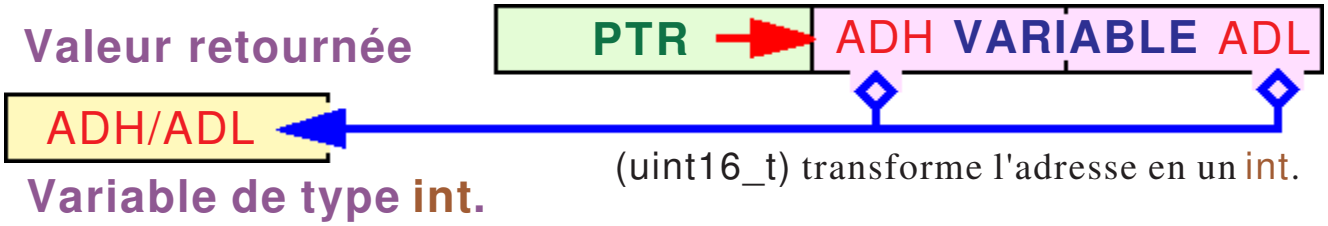
Afficher les adresses associées à un pointeur :

Afficher l'adresse de la donnée pointée.

Pour afficher le contenu de PTR en standard on devrait écrire :
~~Serial.print(PTR);~~
Mais ce codage est refusé par le compilateur car l'instruction concernant **Serial**.print ne peut traiter directement le type pointeur. Pour résoudre cette incompatibilité il faut "réaliser un cast" :

> Pour afficher la **valeur** d'un pointeur il faut coder :
> **Serial**.print((uint16_t) PTR , HEX);

Dans cette écriture c'est l'adresse de la variable pointée qui sera affichée. L'option facultative HEX impose l'affichage en hexadécimal.

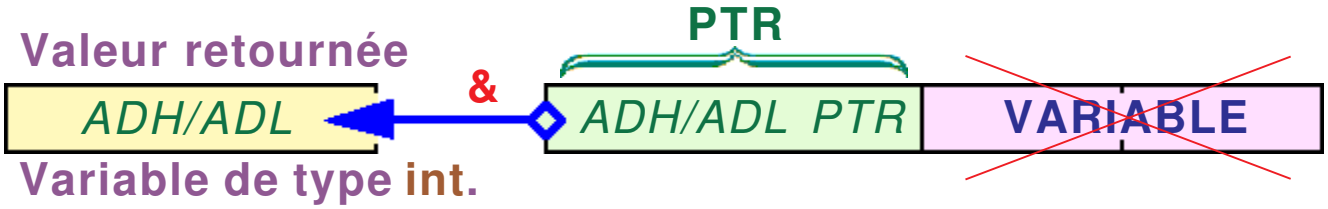


NOTE : Fondamentalement quand on utilise l'identificateur d'un objet (Sans "*" ou "&") l'opérateur d'affectation "=" retourne sa valeur, quel que soit son type. Si l'objet est un pointeur, "=" retourne la valeur de ce dernier, donc l'adresse du premier octet de sa cible.

Afficher l'adresse physique en mémoire du pointeur.

Il suffit en standard de faire précéder l'identificateur du pointeur par l'opérateur "&" pour en obtenir son adresse. Naturellement (uint16_t) déjà précisée ci-avant relative à **Serial**.print s'impose.

> Pour afficher **l'adresse** d'un pointeur il faut coder :
> **Serial**.print((uint16_t) &PTR , HEX);



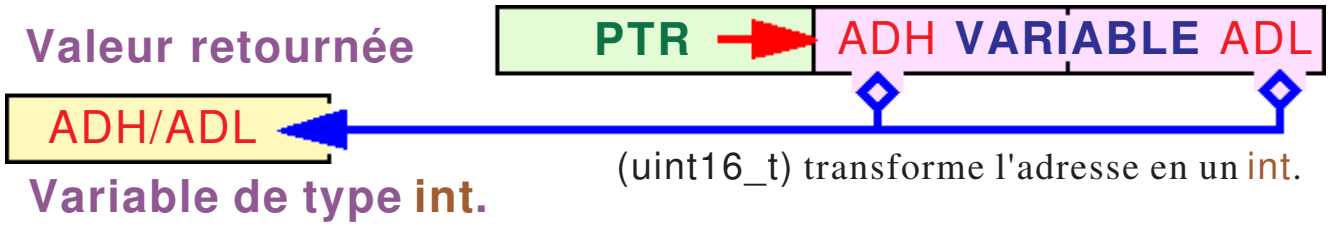
NOTE : Fondamentalement l'opérateur "&" retourne l'adresse en mémoire du premier octet de l'objet avec lequel il est associé, quel que soit la type de cet objet. Que ce soit une variable quelconque, ou celle d'un type pointeur il n'y a pas de différence.

Récupérer dans un entier les adresses d'un pointeur.

Le codage est strictement identique à celui décrit en page 5 pour afficher ces valeurs sur la ligne série. Il faut les transformer en int.

```
int ADRESSE; // Variable de réception des valeurs.
```

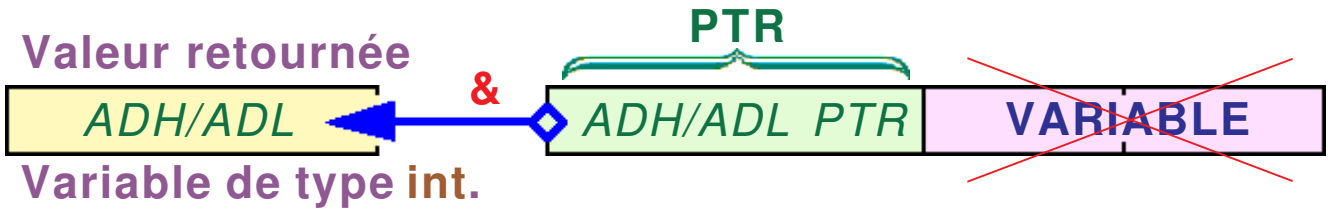
• Récupérer l'adresse de la cible :




```
ADRESSE = (uint16_t) PTR;
```

La variable de réception ADRESSE de type int reçoit sous forme d'un entier le contenu de PTR qui est l'adresse du premier octet de la variable ciblée. *(Valeur $0000 si PTR = NULL)*

• Récupérer l'adresse du pointeur :




```
ADRESSE = (uint16_t) &PTR;
```

La variable de réception ADRESSE de type int reçoit sous forme d'un entier l'adresse physique en mémoire d'implantation de PTR.

Récupérer / Afficher le contenu de la variable pointée.

Le fait que la variable soit pointée ne modifie en rien son type. Donc la comparaison se fait da façon tout à fait banale. Il suffit en standard de faire précéder l'identificateur du pointeur par l'opérateur "*" pour obtenir le contenu de la variable pointée.

Exemple :

```
void Affiche_le_contenu_de_la_cible() {
  Serial.print(*PTR); // Par défaut affiche la valeur en Décimal.
  Serial.print(*PTR, HEX); // Affiche la valeur en Hexadécimal.
  Serial.print(char(*PTR)); } // Affiche en équivalent ASCII.
```

La seule nuance réside dans le fait que l'on précise la variable traitée non pas directement avec son identificateur, mais par son pointeur dont le nom est précédé du caractère "*". *(Retourne le contenu de la cible)*

Pointeur provisoirement non typé déclaré avec "void".

void *Nom_du_pointeur;

Avec void on déclare un pointeur indifférencié qui peut changer de type lors de son utilisation.

Un exemple pratique d'utilisation est donné dans le petit programme Declaration_pointeur_avec_void.ino qui en passage de paramètre dans des procédures peut cibler aussi bien un byte qu'un int.

```
void *PTR; // Définit un pointeur sans préciser son type.
byte OCTET = 0x73;
int  ENTIER = 0x1234;
void loop() {
  PTR = &PTR;  // Pointe sur un octet.
  LISTE_octet_SRAM ("Pointe l'octet : ",(byte*) PTR);
  PTR = &ENTIER;  // Pointe sur un entier.
  LISTE_un_Entier ("Cible l'entier : ",(int*) PTR);
  INFINI: goto INFINI; }
```

Pointeur utilisés en paramètres de fonction.

Un exemple beaucoup plus étoffé que le résumé listé ci-dessous est donné dans Pointeurs_en_paramètres_de_fonctions.ino :

```
byte *PTR1 = NULL; // Pointeurs pour cet exemple.
byte OCTET; // Variable de réception de la fonction.

void loop() {
  PTR1 = (byte*) 0x100; // PTR1 pointe sur 0x100.
  Affiche_PTR(PTR1); // PTR1 passé en paramètre. (1)
  OCTET = Valeur_cible(PTR1); (2)
  Affiche_Octet();
  INFINI: goto INFINI; }

void Affiche_PTR(byte *INDEX) {
  Serial.print("Valeur de PTR = $");
  Serial.print((uint16_t) INDEX,HEX); }

byte Valeur_cible(byte *PTR) {return *PTR; }  (3)
```

Dans cet exemple on peut observer que :

En (1) le pointeur PTR1 est passé en paramètre à la procédure *Affiche_PTR*. En (2) PTR1 est passé en paramètre à la fonction *Valeur_cible*, mais il sert également au retour le la valeur en (3).

Pointeurs et cibles déclarés constants :

Notons au passage qu'utiliser le qualificateur const n'est pas spécifiquement liée à la notion de pointeurs, mais reste recommandée à chaque fois que cette précaution est pertinente. C'est une bonne stratégie de programmation qui permet au compilateur de détecter les erreurs de logique dans le source dès la traduction en binaire.

Adresse pointée qualifiée de constante.

```
byte * const Nom_du_pointeur = (byte*) 0x0300;
```

Le qualificateur const est intercalé entre le caractère "*" et l'identificateur du pointeur. L'adresse de la cible ne peut plus être changée. *(Donc indiquer NULL en adresse ne sera pas très utile !)*

```
uint8_t * const Nom_du_pointeur = (uint8_t*) 0x0300;
```

Écriture équivalente à celle de l'instruction précédente, c'est une autre façon de déclarer le type byte pour la variable.

Donnée pointée qualifiée de valeur constante.

```
const byte *Nom_du_pointeur = &Valeur_a_figer;
```

Le qualificateur const s'applique sur la valeur de la donnée pointée qui ne peut plus être modifiée. Il importe de l'avoir initialisée avant de déclarer ce pointeur, ou cette donnée sera inutilisable.

```
byte const *Nom_du_pointeur = &Valeur_a_figer;
```

Écriture équivalente pour figer la valeur de la cible.

Adresse et donnée pointée qualifiées de constantes.

```
byte const * const Nom_du_pointeur = &Valeur_a_figer;
```

Le pointeur contant vectorise vers une valeur elle-même constante qui ne peut pas être modifiée par le code compilé, mais qui peut tout de même changer s'il s'agit d'une "cellule" dont la mise à jour est matérielle. *(Registre significatif de l'état d'une E/S par exemple)*
Le compilateur n'est pas autorisé à faire une optimisation en se basant sur le fait que la valeur pointée est constante, mais générera une erreur si le programme tente de la modifier par ses instructions.

Le petit programme Pointeurs_utilises_avec_CONST.ino illustre l'utilisation de const dans les divers cas présentés ci-avant. On notera avec ce petit logiciel d'expérimentation que les pointeurs sont placés en mémoire les uns "contre" les autres, mais PTR0 déclaré const et initialisé à NULL n'est pas placé en premier mais intercalé entre les autres.

Variables prédéfinies pour gérer la PILE :

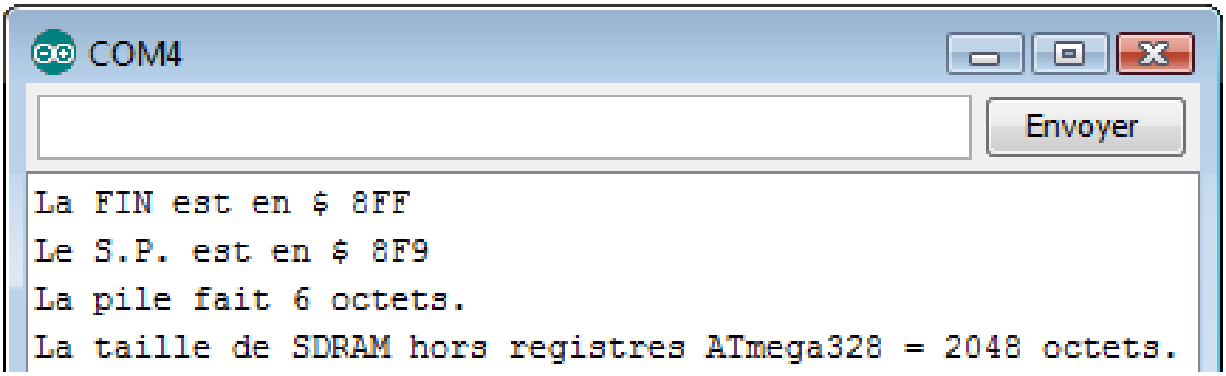
Plusieurs identificateurs réservés sont définis en standard pour faciliter la gestion de la mémoire du microcontrôleur. Ces pointeurs permettent de traiter les adresses des diverses zones spécifiques d'utilisation de la mémoire SRAM. L'organisation générale relative à la gestion de la SRAM est donnée en page 2.

Variables entières prédéfinies.

Deux entiers dont les identificateurs RAMEND et SP sont réservés et n'ont pas à être déclarés pour permettre de gérer la PILE et la taille de la SRAM. La variable RAMEND est spécifique, car le compilateur interdit d'en changer la valeur par programme.
La variable SP peut être modifiée par programme. Mais il est très dangereux potentiellement de modifier la valeur du S.P.

Un exemple d'emploi de ces deux pointeurs prédéfinis est donné dans le petit programme Variables_PREDEFINIES.ino listé en partie ci-dessous qui illustre l'utilisation possible de RAMEND et de SP :

```
void loop() {
// RAMEND = 1234; Instruction non acceptée par le compilateur
//          car RAMEND ne doit pas être modifiée par programme.
// Pas besoin de déclarer RAMEND ni SP.
 Serial.print("La FIN est en $ "); Serial.println(RAMEND,HEX);
 Serial.print("Le S.P. est en $ "); Serial.println(SP,HEX);
 Serial.print("La pile fait "); Serial.print(RAMEND-SP);
 Serial.println(" octets.");
 Serial.print("La taille de SDRAM hors registres ATmega328 = ");
 Serial.print(RAMEND - 0xFF); // Registres du µP exclus.
 Serial.println(" octets.");
INFINI: goto INFINI; }
```



Pointeurs prédéfinis pour gérer la RAM :

Assez similaires aux deux variables RAMEND et SP présentées en page 11, ce sont des objets de type pointeur identifiés par des noms réservés dans avr-libc *(Bibliothèque standard qui n'a pas à être déclarée)* qui doivent être déclarés par des directives extern. L'allocateur de mémoire mis en œuvre par avr-libc fait face aux nombreuses contraintes liées à la faible taille de RAM disponible sur les microcontrôleurs. Cette librairie prévoit des options d'ajustement qui peuvent être utilisées s'il y a plus de ressources disponibles que dans la configuration par défaut. *(RAM externe possible sur certains µP)*

Comme montré dans la Fig.1 située en page 3, l'allocation standard de mémoire dynamique dans le TAS, et celle de la PILE sont séparés dans des zones de RAM distinctes pour éviter les risques de collision. Un certain nombre de variables peuvent être initialisées pour adapter le comportement de malloc() à des exigences particulières de l'application en cours de développement. Toutes modifications de ces paramètres doivent être effectuées avant le premier appel à malloc(). Noter que certaines fonctions de la bibliothèque peuvent également utiliser de la mémoire dynamique. *(Par exemple certaines fonctions de stdio.h telles que l'installation standard des I/O)* Les modifications d'implantation mémoire devront s'effectuer rapidement dès le début de la séquence de démarrage.

Pointeurs spécifiques pour délimiter les zones en SRAM.

Les pointeurs prédéfinis pour gérer globalement la SRAM représentée sur la Fig.1 de la page 3 sont listés ci-contre :

- __heap_end
- __heap_start
- __bss_end
- __bss_start
- __data_end
- __data_start
- __brkval

__heap_end est fondamentalement une "constante" dont la valeur ne change pas pendant l'exécution d'un programme. **En utilisation standard de la SRAM** une valeur particulière pour __heap_end ne présente pas de sens pratique réel. La valeur zéro symbolise le fait qu'il n'y a pas de RAM externe au microcontrôleur. Elle précise à la fonction malloc() que le "haut" du TAS n'est pas vraiment déterminé puisque son extension varie de façon dynamique, et qu'il importe de vérifier la valeur du S.P. au moment d'allouer de la mémoire.

Tout se passe comme si en permanence __heap_end était égal à la valeur actuelle du SP, bien que sa valeur reste égale à zéro.
On ne change la valeur de __heap_end que pour utiliser une RAM additionnelle externe. En standard, la PILE occupe alors toute la RAM interne du µP et __heap_start pointe vers le début de la RAM externe, avec __heap_end qui indique la fin de cette RAM complémentaire.
Les variables __malloc_heap_start et __malloc_heap_end peuvent être utilisés pour restreindre la fonction malloc() à une zone de mémoire bien définie. Ces variables sont statiquement initialisées pour pointer respectivement vers __heap_start *(Initialisé par l'éditeur de liens pour pointer juste après la BSS)* et __heap_end qui est forcé à 0, ce qui permet à malloc() de maintenir le TAS en dessous du S.P.
Si le TAS est déplacé en RAM externe, __malloc_heap_end doit être ajustée en conséquence pendant l'exécution du programme en écrivant directement la valeur dans cette variable, ou peut être fait automatiquement en lieu et temps en ajustant la valeur de __heap_end.

Description des pointeurs de limite des zones en SRAM.

__heap_start : Début de la zone HEAP et fin de la BSS.
__bss_end : Fin de la BSS et début de la zone HEAP.
__bss_start : Début de la BSS et fin de la zone DATA.
__data_end : Fin de la zone DATA et début de la BSS.
__data_start : Début de la zone DATA en $100.

Le pointeur __brkval.

Fondamentalement il représente la première cellule mémoire non encore attribuée dans la zone dynamique HEAP. Si __brkval est égal à zéro, c'est que malloc() par exemple, ou l'allocateur de mémoire dynamique plus généralement n'a pas encore réservé de la mémoire.

Déclaration des pointeurs prédéfinis.

Les pointeurs prédéfinis doivent être déclarés par la directive extern suivi de int , char ou byte qui est le type des objets pointés

```
extern int __heap_end; // Toujours égal à zéro.
extern int __heap_start;
extern int __bss_end;
extern int __bss_start;
extern int __data_end;
extern int __data_start; // Égal à $100 en standard.
extern int __brkval;
```

☞ __malloc_heap_start, __malloc_heap_end ne sont pas à déclarer.

Vérification de "HEAP" encore disponible :

Pouvoir s'assurer en cours de développement d'un programme que la mémoire des données dynamiques n'approche pas la saturation est un impératif. L'utilisation des pointeurs spécifiques dans une procédure minimale avec affichage des paramètres pertinents sur la ligne série USB permet une telle vérification. Le code ci-dessous donné dans Verif_SRAM_disponible.ino précise les quelques lignes à ajouter au programme pour introduire ce type de vérification :

```
void setup() { Serial.begin(115200); } (1)

void loop() {
   Serial.print("SRAM libre = "); (2)  Serial.println(SRAM_LIBRE()); (3)
   INFINI: goto INFINI; }

int SRAM_LIBRE() { // Fonction qui retourne la taille de SRAM disponible.
   extern int __heap_start, *__brkval; (4)
   int BIDON; // Dernière variable allouée, donc en "haut" de la PILE.
   if (__brkval == 0) {return (int) &BIDON - (int) &__heap_start;}
                  else {return (int) &BIDON - (int) __brkval;} } @
```

Voir P12 et P13 les pointeurs prédéfinis.

Ce code utilise le fait que __heap_start correspond à la fin de BSS.

(1) : Cette ligne fait généralement partie du programme en cours de développement, donc ne pénalise pas l'occupation de SRAM.

(2) : Possibilité de minimiser ce texte pour gagner de la place.

(3) : Appel en n'importe quel endroit du programme.

(4) : Déclaration en local des deux pointeurs servant aux calculs.

NOTE : La variable système __brkval représente la première cellule mémoire non encore attribuée dans la zone dynamique HEAP. Si __brkval est égal à zéro, c'est que malloc n'a pas encore utilisé de la mémoire dynamique, dans ce cas on utilise l'adresse de __heap_start.

@ : La ligne de code utilise une écriture standard pour des raisons de lisibilité. Mais dans un souci de compacité du programme elle est généralement réduite par utilisation de l'opérateur ternaire et devient :

```
return (int) &BIDON - (__brkval == 0 ? (int) &__heap_start
                                     : (int) __brkval);
```

Le programme Saturer_SRAM.ino donne un exemple de boucle allant jusqu'à la quasi saturation de la SRAM.

Passage des paramètres par référence :

Cette notion méconuue en langage C est apportée par le C++ qui procure les avantages du passage par pointeur avec la simplicité du passage par valeur grâce au concept novateur de référence. Une référence permet de faire appel à des variables valides dans une autre portée. Par exemple on peut manipuler une variable située dans une procédure ou une fonction à partir d'une autre fonction. La déclaration d'une référence se fait simplement en intercalant un caractère & *(Esperluette ou ET commercial)* entre le type de la variable et son identificateur. Exemple de passage d'argument par référence :

Exemple : Programme Argument_par_reference.ino

```
byte A = 72, B, C; // Variables banales sur un octet.
void setup() {Serial.begin(115200);}
void loop() {
 Ajoute_1(A); // Modifie A. (1)
 Serial.print(" Valeur de A = "); Serial.println(A);
 Ajoute_1(B); // Modifie B.
 Serial.print" Valeur de B = "); Serial.println(B);
 C = Ajoute_1(B); (2)
 Serial.print(" Valeur de B = "); Serial.println(B);
 Serial.print(" Valeur de C = "); Serial.println(C);
 INFINI: goto INFINI; }
byte Ajoute_1(byte &X) // Déclaration de la référence. (3)
 {Serial.print(" Valeur de X = "); Serial.println(X); X ++;}
```

(1) Appel de la fonction sans utiliser l'opérateur d'affectation. Seule la variable A passée en argument par référence est modifiée.

(2) Modifie la variable B car elle est passée en référence. Mais comme il s'agit d'une fonction, on peut utiliser son résultat qui ici est affecté à C. À l'issue de cette instruction B et C auront donc la même valeur.

(3) **On déclare la référence en paramètre obligatoirement sous forme d'une fonction typée.** Il est tout aussi possible d'utiliser le résultat dans un Serial.print du genre : Serial.println(Ajoute_1(C));

Le concept de passage par référence ne doit en aucun cas être confondu avec celui de passage par adresse même si les deux formes utilisent le caractère &. **En passage par référence il n'y a pas déclaration d'un pointeur. De plus la fonction ne peut avoir qu'un seul paramètre.**

Opérateurs valides sur les pointeurs :

Naturellement on va retrouver l'incrémentation et la décrémentation qui permettent au pointeur de se déplacer d'une donnée à la suivante. Mais dans le cas des pointeurs, l'incrément ne fait "1" qui si le type pointé est char ou byte. Pour les autres types de données pointées, la valeur ajoutée ou retranchée à l'adresse préservée par le pointeur sera fonction de leur taille.

Arithmétique de base.

Les opérations arithmétiques permises avec les pointeurs sont l'addition et la soustraction d'une valeur entière à un pointeur. L'adresse contenue dans ce dernier augmente ou diminue de la [valeur de l'opérande N multipliée par la taille de la donnée pointée] pour cibler un ou N objets en amont ou en aval de la cible actuelle.

Pointeur = Pointeur + N; *(Voir Fig.1)*

L'adresse de l'objet ciblé par Pointeur est augmentée de N fois la taille du type du pointeur pour cibler N objets plus en amont.

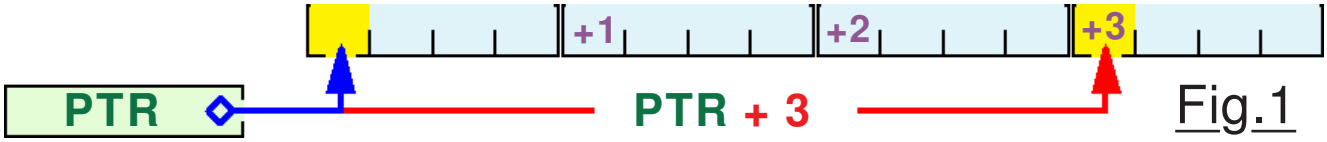


Fig.1

Pointeur = Pointeur - N; *(Voir Fig.2)*

L'adresse de l'objet ciblé par Pointeur est diminuée de N fois la taille du type du pointeur pour cibler N objets plus en aval.

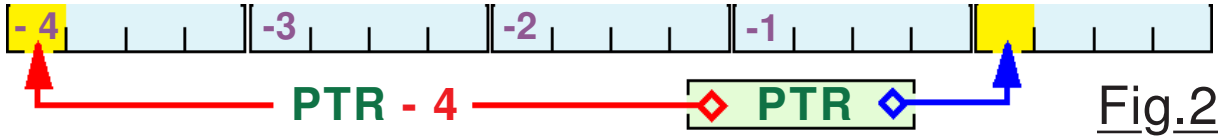


Fig.2

Pointeur++; Pointeur--;

Décale l'adresse d'un élément en amont ou en aval d'un nombre d'octet correspondant à la taille du type pointé. C'est la forme condensée de l'instruction Pointeur = Pointeur + 1 ou de l'instruction Pointeur = Pointeur - 1.

ATTENTION : Il importe de faire très attention avec ce type d'opération à ne pas déborder du "bloc des données pointées", car le programme généré n'effectue aucune vérification. On peut librement cibler n'importe où dans la RAM avec toutes les conséquences potentielles qui en résultent.

Recopie de chaînes de caractères avec les pointeurs.

La recopie d'une chaîne de caractères de type String se fait comme la recopie directe des objets de type String. Quelles que soient les tailles des chaînes origine et de destination au moment de l'affectation :

```
String TITRE = "BONJOUR"; String TEXTE = "XXXX";
String *SOURCE = &TITRE; String *COPIE = &TEXTE;
 *COPIE = *SOURCE;  } sont deux instructions
 TITRE = TEXTE;     } qui ont le même effet.
```

TEXTE est recopié dans TITRE quelles que soient les tailles initiales de ces deux chaînes de caractères.

Soustraction de deux pointeurs de même type.

Cette opération particulière permet de savoir **combien d'objets** du type pointé **sont intercalés entre les adresses ciblées** par les deux pointeurs. Le résultat est un entier positif ou négatif en fonction de l'ordre des pointeurs précisés dans l'instruction.

Exemple : *(Voir Fig.3)* :

```
int *PTR1 = NULL, *PTR2 = NULL;
int var_int, var_ENT; // variables du programme.

void loop() {
 PTR1 = &var_int; PTR2 = &var_ENT;
 PTR1--;  PTR2++;
 // PTR2 - PTR1 retourne +3.
 // PTR1 - PTR2 retourne -3.
```

La soustraction entre deux pointeurs de même type indique le nombre d'éléments compris entre les deux adresses.

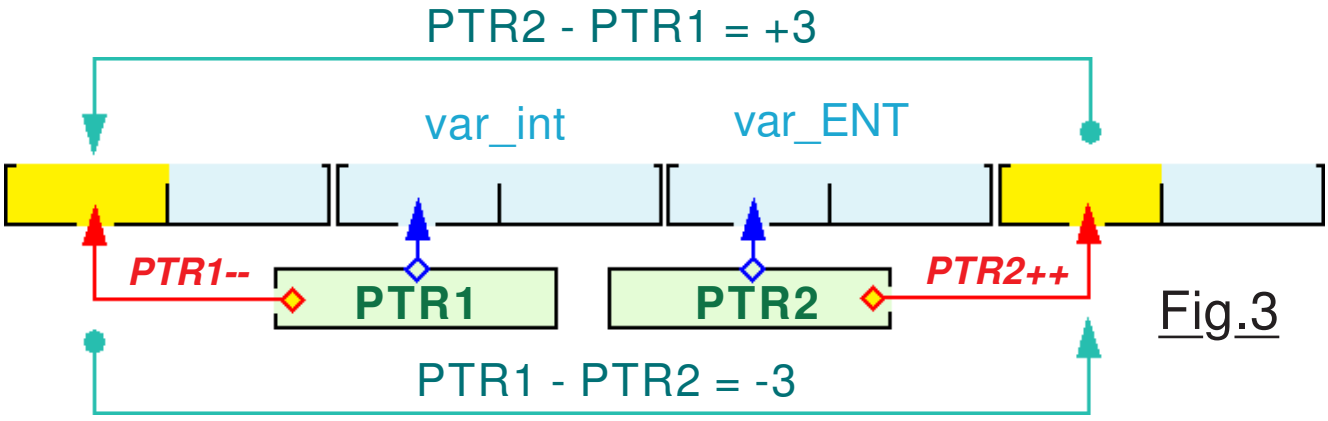


Fig.3

NOTE : Comme il s'agit d'une soustraction, c'est la valeur de l'adresse la plus grande qui conditionne le signe du résultat. Le signe n'est pas vraiment significatif, donc il ne faut pas tenir compte de ce dernier.

Opérations combinées avec les pointeurs :

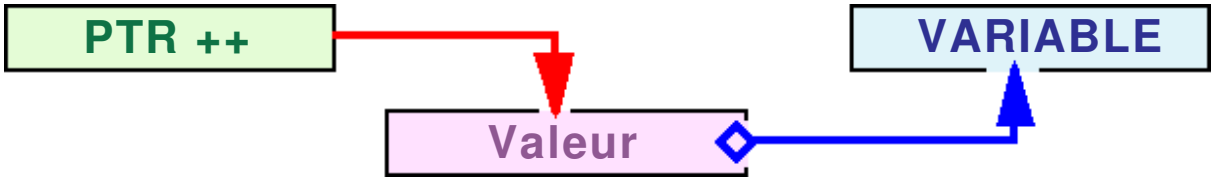
L'usage optimisé de la priorité des opérateurs en langage C autorise des expressions imbriquées dans des instructions combinées et compactes. On peut complexifier les expressions à convenance, mais c'est privilégier la compacité du code au détriment de la clarté et de la simplicité. C'est une "erreur" à éviter.

Expressions combinées.

L'opérateur d'incrémentation ayant la priorité sur celui de déréférencement, c'est donc celui qui sera appliqué en premier. S'il est postfixé, l'opérateur d'incrémentation ne prendra effet qu'à la fin de l'expression, donc au moment de l'affectation. Voici les différents effets obtenus en fonction des combinaisons de ces deux opérateurs :

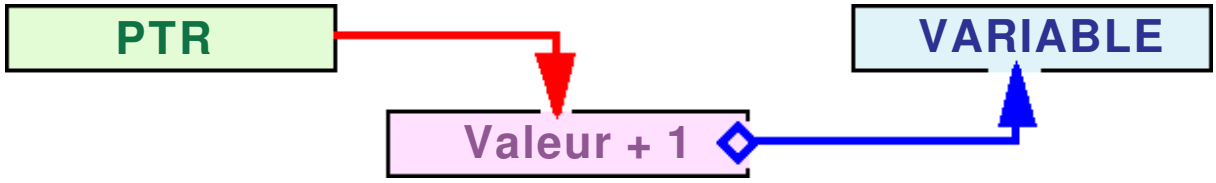
`VARIABLE = *++PTR;`

Incrémente d'abord PTR, puis déréférence sa cible dans VARIABLE.



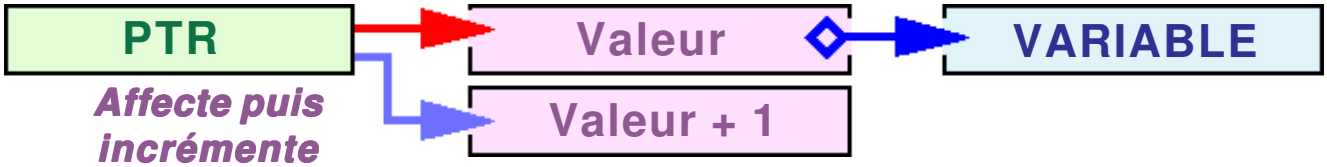
`VARIABLE = ++*PTR;`

Incrémente la cible de PTR, puis la déréférence dans VARIABLE.



`VARIABLE = (*PTR)++;`

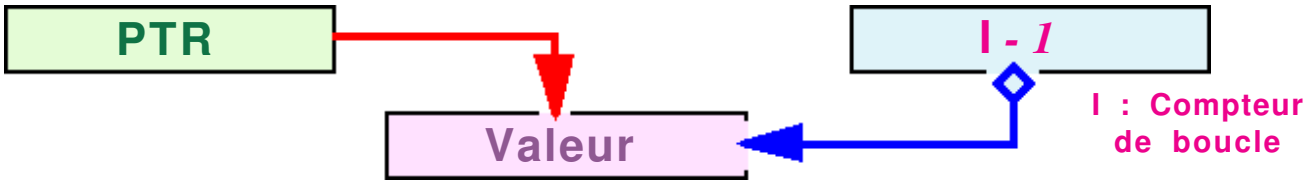
Affecte à VARIABLE la valeur pointée par PTR, puis incrémente la valeur de la cible toujours pointée par PTR.



Autre exemple dans une boucle :

`for (byte I=1; I < 11; I++) {*PTR++ = I-1;}`

Affecte la valeur pointée par PTR avec I-1 puis incrémente PTR.



Exemple : Programme Combinaison_operations.ino

Parallèle entre tableaux et pointeurs :

Fondamentalement, un tableau qui ne contient que des variables de type identique, *(long, int, char, double…)* fonctionne avec un pointeur. Les "[…]" servent à la fois à préciser que l'identificateur est celui d'un tableau, et à définir le nombre d'éléments qui vont le constituer. Comme la déclaration est précédée du type des éléments, la réservation statique de la place en mémoire est possible dès la compilation. On peut considérer qu'un tableau est un pointeur constant qui contient l'adresse du premier élément stocké en mémoire. Ensuite, quand on cherche à accéder à un élément, c'est le mécanisme interne qui génère l'adresse à partir de celle du premier élément, du type des données et de l'ordre de celui que l'on désigne, précisé avec "[n]".

Identifiant d'un tableau utilisé seul, avec & ou avec *.

Pour un objet C déclaré Tableau[N], l'identifiant Tableau utilisé seul correspond à l'adresse du début du tableau en mémoire. *(L'identifiant* ***Tableau*** *utilisé ici est en fait un pointeur.)* Exemples :

Tableau retourne l'adresse du premier élément de Tableau.
&Tableau retourne comme ci-avant l'adresse du premier élément.
*Tableau retourne le contenu du premier élément de Tableau.
Tableau[0] retourne le contenu du premier élément de Tableau.
*&Tableau retourne le contenu du premier élément de Tableau.
*&Tableau[2] retourne le contenu du troisième élément de Tableau.
&Tableau[2] retourne l'adresse du troisième élément de Tableau.
&Tableau+X : Adresse augmentée X qui peut pointer n'importe où !

Rappel : Une adresse n'est pas un nombre entier. Si on désire l'afficher par exemple, il faut effectuer un "cast". *(Transformation en un entier)*

```
Serial.println((int)Tableau); // Cast en (int) car adresse x type int.
```

Remarque : Puisque l'identificateur d'un tableau se comporte comme un pointeur constant sur son premier élément, il est impossible de lui assigner une valeur quelconque, il y aura génération d'une alerte d'erreur.

Remarque : L'utilisation de l'opérateur sizeof qui, lorsqu'il est appliqué à un identificateur de type tableau, donne bien la taille de tout le tableau et non la taille d'un pointeur comme on pourrait le penser.

Chaines de caractères. *(**sizeof** indique la taille du texte sans le '\0')*

On peut les considérer comme des tableaux de caractères, sauf que ces tableaux sont implicitement de type char, avec la sentinelle '\0'.

Passer des paramètres par adresses :

Toutes les variables en langage C sont passées par valeurs aux paramètres des procédures ou des fonctions. *(Exception faite pour les valeurs des éléments constituant les tableaux)* C'est à dire que des copies de leurs valeurs sont effectuées dans la PILE au moment de l'appel de la procédure ou de la fonction. Toutes les modifications de ces variables effectuées durant la "subroutine" seront perdues lors du retour au code appelant. Il y a des cas où l'on désirerait pouvoir modifier une variable passée en paramètre et que ses modifications perdurent dans la procédure ou la fonction appelante. C'est l'utilisé fondamentale du passage des paramètres par adresse : Permettre la modification d'une ou plusieurs variables en visibilité du code appelant.

Passage de paramètres par adresses.

Exemple : Programme Passage_parametres_par_adresses.ino

```
/* Passage de paramètres par adresses. */
// La procédure permute le contenu des deux variables.

byte X = 0x73, Y = 0x88, A = 0xAA, B = 0x55;

void setup() {Serial.begin(115200);}

void loop() { Permute_Octets (&X, &Y);
  Serial.print(" X = $"); Serial.println(X, HEX);
  Serial.print(" Y = $"); Serial.println(Y, HEX);
   Permute_Octets (&A, &B);
  Serial.print(" A = $"); Serial.println(A, HEX);
  Serial.print(" B = $"); Serial.println(B, HEX);
 INFINI: goto INFINI; }

void Permute_Octets (byte *X, byte *Y) {
      byte TMP = *X; *X = *Y; *Y = TMP; }
```

Dans cet exemple, la procédure crée les pointeurs au moment de l'appel, dans le bloc de ses paramètres.

Fonctions retournant plusieurs paramètres.

Dans l'exemple de fonction retournant une valeur pointée donnée en page 9, l'instruction return ne peut renvoyer qu'une seule valeur. Si l'on désire renvoyer plusieurs valeurs, il faut procéder par l'utilisation des adresses comme montré dans l'exemple ci-dessus. Rien n'interdit, naturellement que return serve à retourner un résultat complémentaire issu d'un traitement spécifique effectué durant la fonction.

Passage par adresse de données non modifiées.

Passer des paramètres par adresse est également très utilisé pour optimiser la quantité de données qui doit transiter par la PILE, cette place disponible pour les données en RAM étant très restreinte sur les microcontrôleurs. Dans ce but, même si la variable ne doit pas être modifiée, on utilise quand même un passage par adresse pour éviter la copie implicite des variables autres que celles des tableaux. C'est particulièrement vrai avec les structures, sans pour autant trop nuire à la lisibilité, puisque celles-ci ont tendance à être assez imposantes.

Problème potentiel lors du changement d'une adresse.

Il importe de toujours avoir à l'esprit que l'affectation d'un pointeur à l'intérieur d'une procédure peut avoir un effet de bord si l'on y prend pas garde. Dans l'exemple donné ci-dessous, le corps de la procédure modifie la valeur de X même si on lui passe comme adresse celle de la variable Y comme c'est le cas en @.

```
byte X = 0x20, Y = 0x40;

void Procedure_qui_modifie_X(byte *PTR) {
  PTR = &X; // PTR pointe X comme cible.
  ++*PTR; } // La cible de PTR est incrémentée.

void loop() {
   Procedure_qui_modifie_X(&X);
   Procedure_qui_modifie_X(&Y);  @
   Suite du PGM ...
```

Si la lecture du contenu de la procédure n'est pas effectué, l'appel @ incite à penser que c'est la variable Y passée par adresse qui sera traitée alors, que c'est toujours X qui est incrémentée, puisque dans le code la valeur du pointeur est "détournée". Pour éviter ce risque, on pourrait déclarer le pointeur ou la cible comme étant constant :

```
void Procedure_avec_verification(const byte *PTR) {...} (1)
void Procedure_avec_verification(byte *const PTR) {...} (2)
```

(1) : On impose que la donné pointée sera une constante.
(2) : On impose que le pointeur soit d'adresse constante.
Avec cette précaution le compilateur va s'apercevoir que dans le corps de la procédure il y a tentative de modification soit de la valeur de la cible, soit de celle de PTR et générer un message d'erreur. Il est toutefois plus judicieux d'optimiser le code, car cette écriture n'est pas très lisible.

Gestion dynamique des tableaux :

L'application typique des pointeurs pour de l'allocation dynamique de mémoire consiste à pouvoir décider de la taille d'une variable au moment de l'exécution du programme, car elle n'est pas encore connue lors de son développement. Ainsi, pour allouer un tableau de N entiers, *(N étant déterminé durant l'exécution du programme)*, on déclare une variable de type pointeur sur entier avec lequel on alloue une zone mémoire correspondant à la taille nécessaire. Exemple :

```
byte NB_elements; // byte : Car taille prévue < 255.

void loop() {
  NB_elements = Valeur entière; (1)
  /* Allocation dynamique du tableau */
  long *TABLEAU = NULL;
  TABLEAU = (long*) malloc(NB_elements *sizeof(long));
  /* Utilisation du tableau */
  if (TABLEAU  != NULL) { (2)
     /* Sauvegarder l'adresse du début de TABLEAU. */
     long *DEBUT = TABLEAU;  (3)
      ... instructions pour utiliser le tableau.
     /* Restituer la RAM utilisée par le tableau */
     TABLEAU = DEBUT; // Restituer l'adresse du pointeur.
     free(TABLEAU); } // Libérer la place réservée. (4)
      ... suite du programme.
```

Commentaires sur ce programme :

(1) Le nombre d'éléments sera déterminé durant l'exécution du programme. Par exemple une saisie clavier sur la ligne série etc.

(2) Le pointeur TABLEAU reçoit l'adresse du premier élément au moment de la réservation de mémoire. Si la place n'est plus disponible en RAM, la réservation n'a pas lieu et malloc retourne la valeur NULL. Donc un else préviendrait que l'action n'est pas possible.

(3) Logiquement on peut considérer que l'exploitation du tableau va se faire par utilisation du pointeur TABLEAU. Hors l'instruction free() impose que le pointeur utilisé par malloc() contienne l'adresse du début de la zone allouée. Il importe de sauvegarder cette dernière.

(4) On ne libère la place que si l'utilisation du tableau n'est que temporaire. S'il sert durant tout le programme, free() ne sert à rien.

Exemple : Programme pointeurs_et_tableaux.ino

Passer un tableau en paramètre de fonction :

Pour désigner un tableau dans une fonction, il faut passer en argument son identificateur, et éventuellement en option sa taille. Procédant par adresses, il n'y a pas recopie du tableau et l'on modifie directement l'original. Pour "passer" un tableau à une fonction *(Ou une procédure si* **void** *remplace type)* on écrit une instruction du genre :

type_fonction NOM_FONCTION (type TABLEAU[], typeTaille)

NOTE : Inutile de préciser la Taille du tableau entre crochets car le compilateur l'ignorera. On peut toutefois la passer en paramètre si l'on utilise une boucle dans le traitement, *(Il ne faut pas déborder le tableau)* la procédure pouvant traiter des tableaux de tailles différentes.

Un tableau est converti en un pointeur sur son premier élément. C'est cette adresse qui est passée en paramètre à la procédure ou à la fonction. Du reste on peut désigner un tableau comme argument avec la forme :

type_fonction NOM_FONCTION (type *TABLEAU)

Les deux écritures sont strictement équivalentes. *(Dans les deux cas* type *indique le type des éléments du tableau)* Le choix de la forme utilisée est personnel et ne concerne que la présentation du programme.

Exemple : Programme Tableaux_de_pointeurs.ino

Tableau de pointeurs :

Parmi les applications possibles ils permettent de manipuler des éléments de tailles variables, des structures chaînées etc. Si nous avons besoin d'un ensemble de pointeurs du même type, nous pouvons les réunir dans un tableau de pointeurs :

type_cible * Nom_tableau [NB];

Déclare Nom_tableau de NB pointeurs de type_cible.

Par exemple double *A[10]; déclare un tableau A de 10 pointeurs sur des rationnels du type double dont les adresses et les valeurs ne sont pas encore définies. Le programme Tableaux_de_pointeurs.ino donne un exemple très complet d'un tableau de chaînes de caractères de dimensions différentes, avec des traitements dans lesquels :

PLANETES [n] retourne la chaîne entière de caractères.

>>> Ne fonctionne pas en affectation : ~~PLANETES [n] = "chaine";~~

*(PLANETES [n]) retourne le contenu *(Caractères)* de l'octet pointé.

(uint16_t) *(&PLANETES [n]) retourne l'adresse de la CIBLE : C'est l'adresse du premier caractère de la chaîne de rang (n+1).

(uint16_t) &PLANETES [n]) retourne l'adresse de l'élément de rang (n+1) du tableau de pointeurs. *(Adresse en RAM du pointeur)*

RÉSUMÉ sur l'utilisation basique des POINTEURS :

type *PTR = NULL; // Déclare PTR sur cible de taille type.

PTR est une nouvelle variable de nature pointeur. La variable ciblée sera du genre type. Bien que ce ne soit obligatoire on précise au compilateur qu'actuellement PTR **n'est pas affecté**.

Gérer une adresse dans PTR peut utiliser les opérations spécifiques aux pointeurs telles que l'incrémentation, mais ces dernières sont fonction de la taille des données ciblées, d'où type associé au pointeur.

PTR = &VARIABLE; // PTR pointe VARIABLE.

PTR reçoit l'adresse du premier octet de VARIABLE.

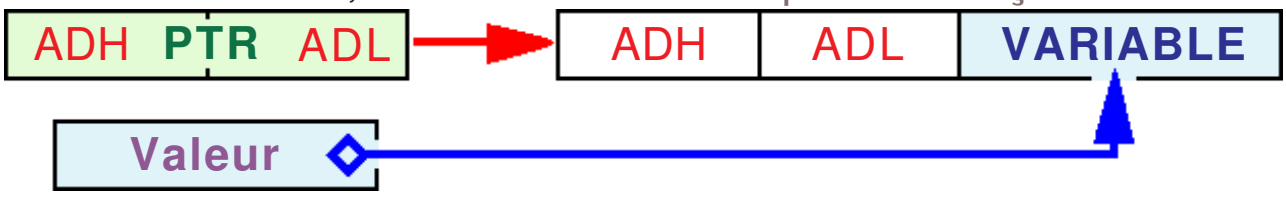
PTR2 = PTR1; // PTR2 reçoit le contenu de PTR1.

PTR1 et PTR2 sont **deux pointeurs de même** type.

type *PTR = (type *) ADR; // PTR pointera l'adresse ADR.

Affecte au pointeur PTR ciblant une variable de taille type la valeur d'un entier que l'on peut exprimer en décimal, octal ou hexadécimal.

*PTR = Valeur; // VARIABLE ciblée par PTR reçoit Valeur.



Affecte la Valeur **respectant le** type **pointé** à la zone mémoire dont l'adresse du premier octet est contenue dans PTR.

Retour = *PTR // Retourne le contenu de la variable ciblée.

Retourne la Valeur de VARIABLE actuellement ciblée par PTR.