- Introduction à l'algorithmique et au langage C -

Adresses et pointeurs

TD6

1re année ESIEA - Semestre 1

L. Beaudoin & R. Erra & A. Gademer & L. Avanthey

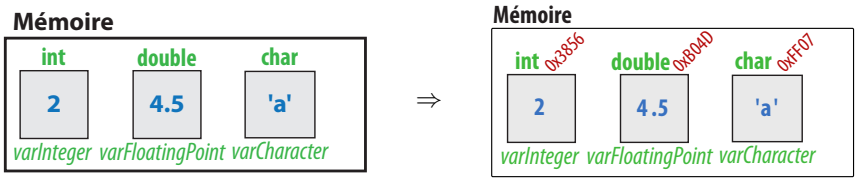
2015 - 2016

Avant propos

Nous savons que les variables sont des espaces de la mémoire dans lesquelles nous pouvons sauvegarder des informations. Nous allons dans ce TP nous intéresser d'un peu plus près à la manière dont cette mémoire est organisée et introduire la notion d'adresse. Nous aborderons ensuite les notions de pointeurs (variables qui peuvent contenir des adresses) et ce que cela change dans notre représentation des structures de données comme les tableaux.

1 Variables, noms et adresses

Une variable est un espace mémoire réservé pour un **type** d'information. Nous avons vu que nous manipulons généralement cet espace par le biais de son **nom**. Mais chaque espace mémoire possède aussi une position dans la mémoire de l'ordinateur. Cette dernière est identifiée par une **adresse** unique.



Adresse : Une adresse est un nombre entier non signé qui représente la position d'un espace mémoire dans la mémoire de l'ordinateur.

Remarque

Par soucis de compacité (visuelle), les entiers non signés utilisés pour représenter des adresses ne sont pas en base 10 (celle que nous utilisons généralement), mais en base 16 (hexadécimale, de 0 à F(15)). En langage C, nous pouvons écrire des nombres en hexadécimal en leur apposant le préfixe `0x`. Par exemple, écrire `0xbfa9a4dc` en hexadécimal équivaut à écrire 3215566044 en décimal [1]. Nous verrons bientôt plus en détail ces histoires de bases.

Comment obtenir l'adresse d'une variable ?

Avec l'opérateur `&` (esperluette [2]) ! Rappelez-vous, nous l'avons déjà utilisée lors des appels de la fonction `scanf`, quand nous voulions mémoriser l'information récupérée au clavier dans une variable. En réalité nous donnions à la fonction l'adresse de la variable.

Nous pouvons **afficher une adresse** avec le descripteur de type `"%p"`.

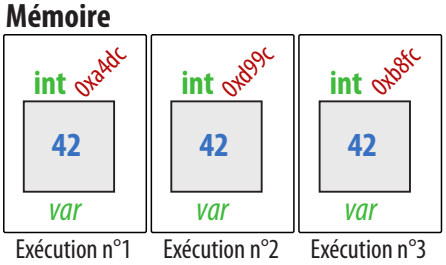
```
int var = 42;
printf("The value of var is %d.\n", var);
printf("The address of var is %p.\n", &var);
printf("The address of var is still %p.\n", &var);
```

```
$./myProg
The value of var is 42.
The address of var is 0xbfa9a4dc.
The address of var is still 0xbfa9a4dc.
$./myProg
The value of var is 42.
The address of var is 0xbfb1d99c.
The address of var is still 0xbfb1d99c.
$./myProg
The value of var is 42.
The address of var is 0xbfcdb8fc.
The address of var is still 0xbfcdb8fc.
```

Où se trouve ma variable dans la mémoire ?

Nous pouvons voir sur l'exemple précédent que la variable `var` change d'adresse à chaque exécution du programme (mais pas au sein du programme).

En effet, les espaces mémoires sont alloués **par le système d'exploitation** à chaque programme et chacun d'entre eux a son espace réservé propre : c'est ce qui vous permet de lancer plusieurs fois le même programme en parallèle sans conflit entre les variables.



1. Si vous êtes observateurs vous constaterez que les adresses des schémas ont été tronquées pour des raisons de place sur le dessin. :-p Mais cela ne change rien à l'explication théorique.
2. http://fr.wikipedia.org/wiki/Esperluette

Et le nom dans tout ça ?
Le nom des variables n'a de sens qu'au niveau du code source C, le binaire de son côté ne manipulant que des adresses. C'est cependant un outil inestimable pour le programmeur qui lui permet de rendre son programme intelligible pour les humains.

2 Stocker des adresses : les pointeurs

Nous venons de définir les adresses comme étant des entiers non signés, ce sont donc des valeurs qui peuvent être stockées dans des variables... non ?

Cependant les adresses ne peuvent pas être stockées dans n'importe quelle sorte de variables (sinon comment les distinguer des autres types ?), mais dans des variables spéciales appelées **pointeurs** (*pointer* en anglais).

Les pointeurs peuvent être définis selon plusieurs *types pointeurs* en fonction du type de la variable associée à l'adresse. Par exemple, si la variable dont nous stockons l'adresse est de type `int` alors le *type pointeur* du pointeur qui la contient sera `int *`. Voici les correspondances pour les types de variables que nous utilisons :

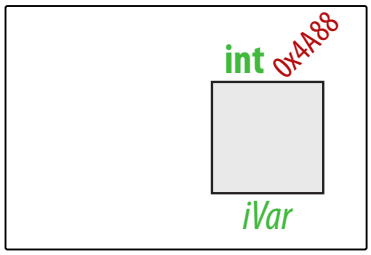
Type pointeur	Type variable
`int *`	`int`
`double *`	`double`
`char *`	`char`

Convention de nommage
Pour faciliter la lecture des codes, nous ajoutons généralement le suffixe `"p_"` à tous les noms de pointeurs.

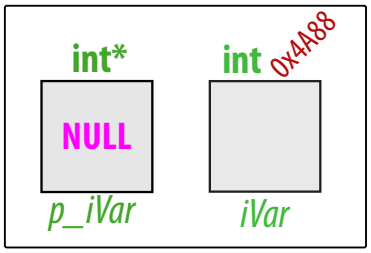
Prenons un exemple :

```c
// 1) Variable declaration
int iVar;

// 2) Pointer declaration
int *p_iVar = NULL; // pointer for int

// Printing of the address
printf("The value of &iVar is %p and p_iVar is %p.\n", &iVar, p_iVar);

// 3) Saving the adresse in the pointer
p_iVar = &iVar;

// Printing of the address
printf("The value of &iVar is %p and p_iVar is %p.\n", &iVar, p_iVar);
```

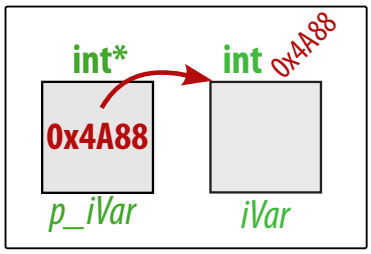
```
The value of &iVar is 0xbf954a88 and p_iVar is (nil).
The value of &iVar is 0xbf954a88 and p_iVar is 0xbf954a88.
```



1) Déclaration de la variable



2) Déclaration du pointeur



3) Nous affectons l'adresse de la variable au pointeur.

EXERCICE 1

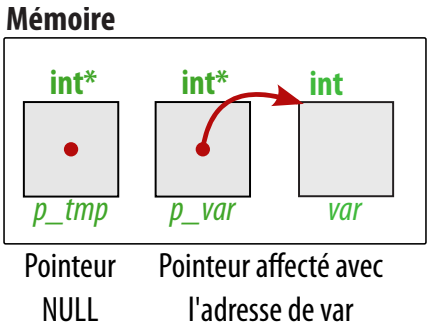
Recopiez ce code et exécutez-le.

QUESTION 1

- Écrivez les lignes nécessaires pour sauvegarder et afficher l'adresse d'un flottant (`double`).
- Écrivez les lignes nécessaires pour sauvegarder et afficher l'adresse d'un caractère (`char`).
- Rajoutez ces lignes dans votre code et exécutez-le.

Représentation graphique des pointeurs : la flèche

Vous aurez compris que les adresses représentées dans les schémas sont choisies arbitrairement. C'est pourquoi nous utilisons généralement un langage graphique pour représenter les pointeurs sous la forme d'une case marquée d'un point reliée le cas échéant par une flèche à l'espace mémoire correspondant à l'adresse stockée.



3 Accéder aux espaces mémoires stockés

Une fois initiés aux arcanes des adresses, nous pouvons avoir envie d'accéder directement aux contenus des espaces mémoires que nous avons stockés dans nos pointeurs (sans passer par les noms de variables) !

Cela est rendu possible par **l'opérateur * (étoile)** qui **permet d'accéder au contenu** d'un espace mémoire dont l'adresse a été stockée préalablement dans un pointeur.

Il se base sur **l'adresse** contenue et sur le *type pointeur* du pointeur pour déterminer le **type** de l'espace mémoire pointé.

3.1 Accès en lecture

En récupérant l'adresse d'une variable dans un pointeur, nous pouvons garder une référence sur celle-ci, cette référence est identique à la variable et suit l'évolution de sa valeur.

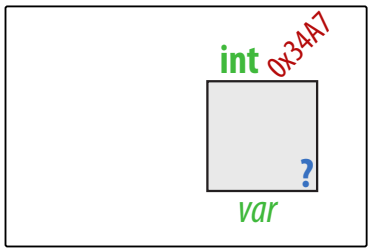
```c
// 1) Variable declaration
int var = 4;

// 2) Pointer declaration
int *p_var = NULL;

// 3) Saving the adresse in the pointer
p_var = &var; // (*p_var) shall be identical to var
printf("var = %d, (*p_var) = %d", var, (*p_var));

// 4) Changing the value of var
var++;
printf("var = %d, (*p_var) = %d", var, (*p_var));
```

EXERCICE 2

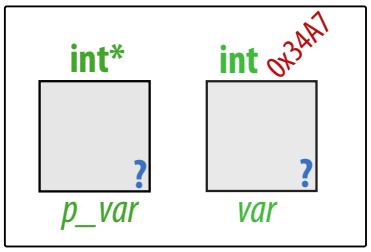
Recopiez ce code dans un nouveau fichier puis exécutez-le.

QUESTION 2

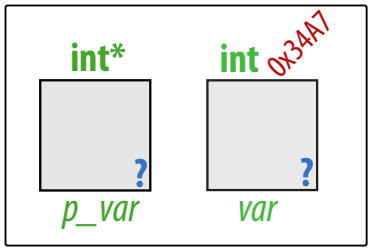
- Que voyez-vous à l'écran ?
- Complétez le schéma suivant pour expliquer ce résultat.



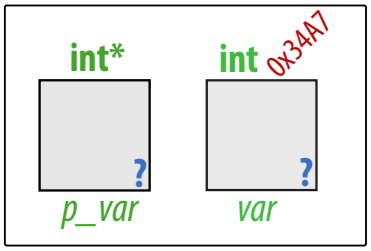
1) Déclaration de la variable



2) Déclaration du pointeur.



3)Le pointeur récupère l'adresse de `var`.

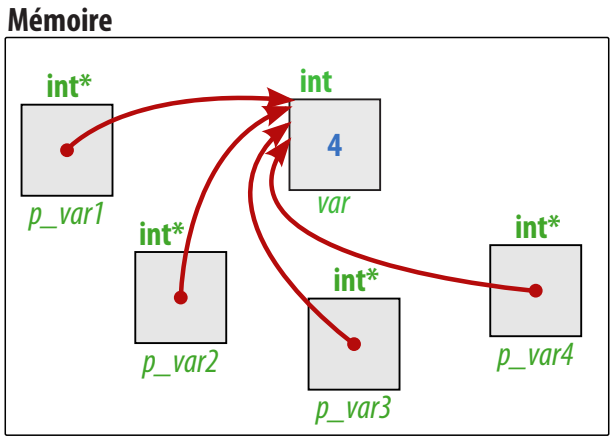


4) Nous incrémentons `var`.

À l'affichage, les contenus de `(*p_var)` et `var` sont identiques, ce qui est normal puisqu'elles correspondent au MÊME espace mémoire.

Premier super-pouvoir des pointeurs : les références multiples

Chaque espace mémoire possède un nom unique : il n'est pas possible d'avoir plusieurs noms de variable qui correspondent au même emplacement. Cependant, comme les pointeurs permettent d'accéder, par le biais des adresses, à des espaces mémoires, ils peuvent être considérés comme des références alternatives sur cet espace. Comme nous pouvons définir autant de pointeurs que nous désirons sur un même espace mémoire, nous parlons de références multiples.



3.2 Accès en écriture

Puisque nous avons accès, par le biais de l'opérateur *, à un espace mémoire équivalent à une variable, nous pouvons aussi bien l'utiliser en lecture qu'en écriture, comme le montre le petit exemple suivant :

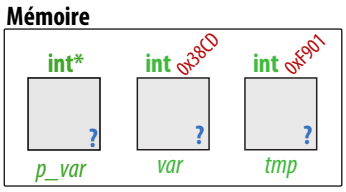
```c
/* 1) Declaration */
int var = 7;
int tmp;
int *p_var = &var; // (*p_var) is now identical to var

/* Show */
printf("var = %d, (*p_var) = %d", var, (*p_var));

/* 2) Read access */
tmp = (*p_var) + 5;
printf("var = %d, (*p_var) = %d, tmp = %d", var, (*p_var), tmp);

/* 3) Write access */
(*p_var) = tmp;
printf("var = %d, (*p_var) = %d, tmp = %d", var, (*p_var), tmp);
```

EXERCICE 3

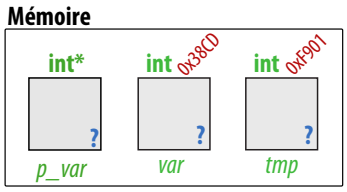
Recopiez ce code dans un nouveau fichier puis exécutez-le.

QUESTION 3

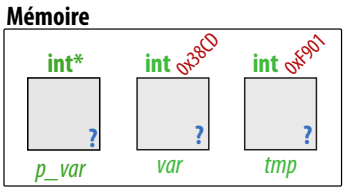
- Que voyez-vous à l'écran ?
- Complétez le schéma suivant pour expliquer ce résultat.



1) Déclaration et affectation.



2) Accès en lecture



3) Accès en écriture

Pourquoi plusieurs type de pointeurs ?
C'est pour pouvoir déterminer de manière non ambigüe le type de l'espace mémoire obtenu par l'opérateur * que les pointeurs possèdent autant de *types pointeurs* différents.

Déclaration	Type de la variable pointée
`int *p_iVar;`	`(*p_iVar) est de type int`
`double *p_dVar;`	`(*p_dVar) est de type double`
`char *p_cVar;`	`(*p_cVar) est de type char`

Espace mémoire autorisé

Nous avons vu dans les TD & TP sur les tableaux ce qui se passait lorsque nous débordions hors de la mémoire réservée (erreur de segmentation à l'exécution, Memory Violation (MV) sur le correcteur automatique, ...). Les possibilités offertes par l'opérateur * renforcent le risque de débordement, soyez donc extrêmement prudents.

Exemple de violation mémoire :

```
int* p_var; // Pointer declaration
p_var = NULL; // NULL or arbitrary address
(*p_var) = 10; // ???
```

EXERCICE 4

Testez ce code. Que se passe-t-il ?

Allocation dynamique de mémoire

Vous verrez l'année prochaine comment réserver directement des espaces mémoire sans passer par la déclaration de variable. Pour cette année, vous vous contenterez de manipuler des espaces mémoires préalablement déclarés de manière classique.

4 Pointeurs et fonctions : passage par adresse

Nous avons vu que les espaces mémoires sont cloisonnés. C'est-à-dire que quand nous sommes dans une fonction (`main` ou autre) les variables que nous y déclarons ne sont pas accessibles depuis l'extérieur. Donc quand nous voulons passer de l'information d'une fonction à l'autre nous sommes obligés de passer par les paramètres (entrées) et la valeur de retour (sortie).

```
int main() {
   int iArray[10] = {9, 6, 10, 12, 9, 3, 5, 12, -4, 8};
   int min;

   min = getMin(10, iArray);
}
```

Cette méthode a des limites : nous ne pouvons retourner qu'une seule et unique valeur. Si nous voulions par exemple une fonction qui identifie le max **et** le min, ce n'est pas possible de cette manière car nous ne pouvons pas retourner deux valeurs en même temps.

2ème super-pouvoir : rupture des cloisons mémoires

Les adresses vont nous permettre de percer ces cloisons entre les espaces mémoires des fonctions. Nous n'allons plus passer uniquement des valeurs en paramètres, mais aussi des adresses. Pour récupérer ces adresses dans la fonction, il faut déclarer des pointeurs en paramètre.

Si nous cherchions à récupérer simultanément le min et le max, nous écririons le prototype suivant (notez la présence des deux pointeurs) :

```
void getMinMax(int size, int iArray[], int *p_min, int *p_max)
```

Remarque

Notez-bien que nous n'avons plus besoin de retourner de valeur. Le type de retour de la fonction devient donc `void`.

Remarque sur la remarque

Souvent, les fonctions qui utilisent les pointeurs pour retourner des informations, utilisent leur valeur de retour comme statut du bon déroulement de la fonction, comme nous l'avons déjà vu avec la fonction `scanf`.

Dans la fonction ça donne quoi ? Nous considérons que l'utilisateur nous a passé des adresses valides en argument. Nous allons donc accéder à ces espaces mémoires grâce à l'opérateur `*`.

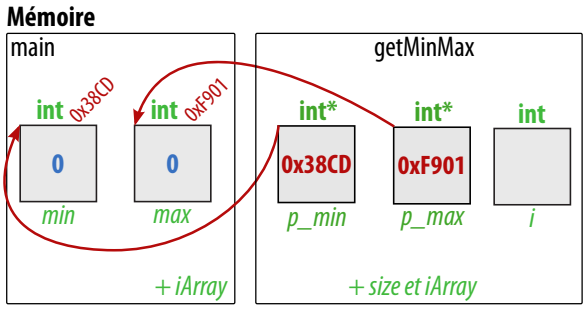
```
void getMinMax(int size, int iArray[], int *p_min, int *p_max) {

   // (*p_min) is identical to min
   // (*p_max) is identical to max

   int i;
   // Initialisation of min and max
   (*p_min) = iArray[0];
   (*p_max) = iArray[0];

   // For each element of iArray
   for( i = 1; i < size ; i++) {

      // Do we have a new min ?
      if( iArray[i] < (*p_min) ) {
         (*p_min) = iArray[i];
      } else {
         // Do we have a new max ?
         if( iArray[i] > (*p_max) ) {
            (*p_max) = iArray[i];
         }
      }
   }
}
```

Espace mémoire déclaré à l'extérieur de la fonction

Les pointeurs ne stockent que des **adresses**. Les variables qui contiendront les valeurs à renvoyer sont, pour leur part, déclarées à l'extérieur de la fonction. Si vous oubliez de les déclarer, cela provoquera une violation mémoire et une erreur de segmentation.

```c
int main() {
   int iArray[10] = {9, 6, 10, 12, 9, 3, 5, 12, -4, 8};

   // Declaring the variable
   int min = 0;
   int max = 0;

   // Showing the original values
   printf("Min = %d, Max = %d\n", min, max);

   // Passing the adresses to the function
   getMinMax(10, iArray, &min, &max);

   // Showing the modified values
   printf("Min = %d, Max = %d\n", min, max);

   return 0;
}
```

Pour vous aider, voici la représentation mémoire du programme au moment de l'appel de la fonction.



QUESTION 4

Déroulez ce code à la main. Que devrait-il afficher ?

EXERCICE 5

Recopiez-le code dans un nouveau fichier, puis exécutez-le pour valider votre réponse.

5 Mise en pratique

QUESTION 5

 En vous basant sur l'algorithme d'échange, écrivez une fonction `swap` dont vous déterminerez le prototype et qui permet d'échanger les contenus de deux variables de type `int`.

QUESTION 6

 Explicitez le fonctionnement de votre fonction avec un schéma mémoire (Avant l'appel, Pendant l'appel, Après l'appel).

6 Lecture de code

QUESTION 7

Déroulez le code suivant à la main ? Qu'affiche-t-il ?

```c
#include <stdio.h>

void addToInteger1(int integer1, int integer2) {
   printf("1/ integer1 = %d integer2 = %d\n", integer1, integer2);
   integer1 += integer2;
   printf("1/ integer1 = %d integer2 = %d\n", integer1, integer2);
}

void addToInteger2(int var1, int var2) {
   printf("2/ var1 = %d var2 = %d\n", var1, var2);
   var1 += var2;
   printf("2/ var1 = %d var2 = %d\n", var1, var2);
}

void addToInteger3(int *p_integer1, int *p_integer2) {
   printf("3/ (*p_integer1) = %d (*p_integer2) = %d\n", (*p_integer1), (*p_integer2));
   (*p_integer1) += (*p_integer2);
   printf("3/ (*p_integer1) = %d (*p_integer2) = %d\n", (*p_integer1), (*p_integer2));
}

int main() {
   int var1, var2;
   var1 = 2;
   var2 = 2;
   printf("m/ var1 = %d var2 = %d\n", var1, var2);

   addToInteger1(var1, var2);
   printf("m/ var1 = %d var2 = %d\n", var1, var2);

   addToInteger2(var1, var2);
   printf("m/ var1 = %d var2 = %d\n", var1, var2);

   addToInteger3(&var1, &var2);
   printf("m/ var1 = %d var2 = %d\n", var1, var2);

   return 0;
}
```

EXERCICE 6

Validez votre réponse en recopiant le code et en l'exécutant.

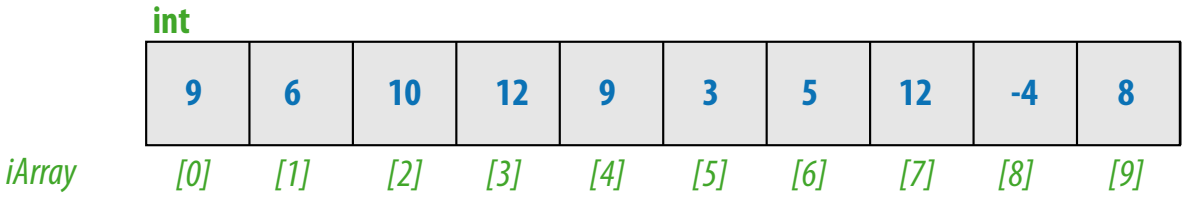
QUESTION 8

Quelles remarques pouvez-vous faire au sujet des trois fonctions ?

7 Pointeurs et tableaux

Lorsque nous avons défini les tableaux, nous vous avons indiqué que ces derniers définissent des espaces mémoires contigüs. Nous avons aussi précisé que nous accédions aux espaces mémoire par le biais du **nom** du tableau suivi de l'**indice** de la case entre crochets.

Ancienne représentation :



Dans les coulisses

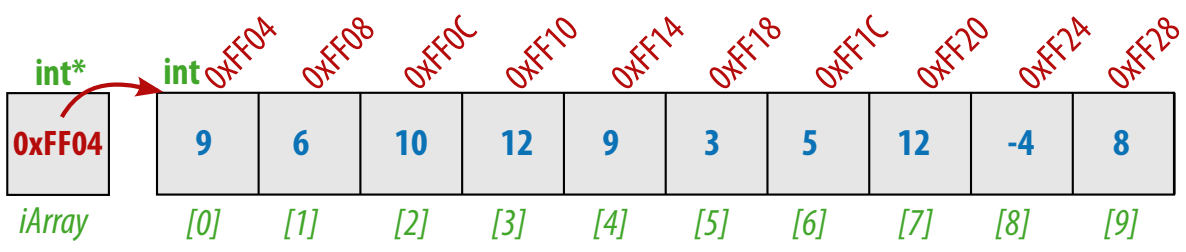
Nous avons vu qu'il nous suffisait de passer le nom d'un tableau à une fonction pour avoir accès à son contenu. Nous avons aussi vu que nous pouvions **modifier** le contenu du tableau dans la fonction et sans soucis apparent des cloisons mémoire !

Cela est lié évidement lié aux adresses et aux pointeurs. Dans le cas d'un tableau mono-dimensionnel, le nom correspond au pointeur sur la première case du tableau.

Notez bien

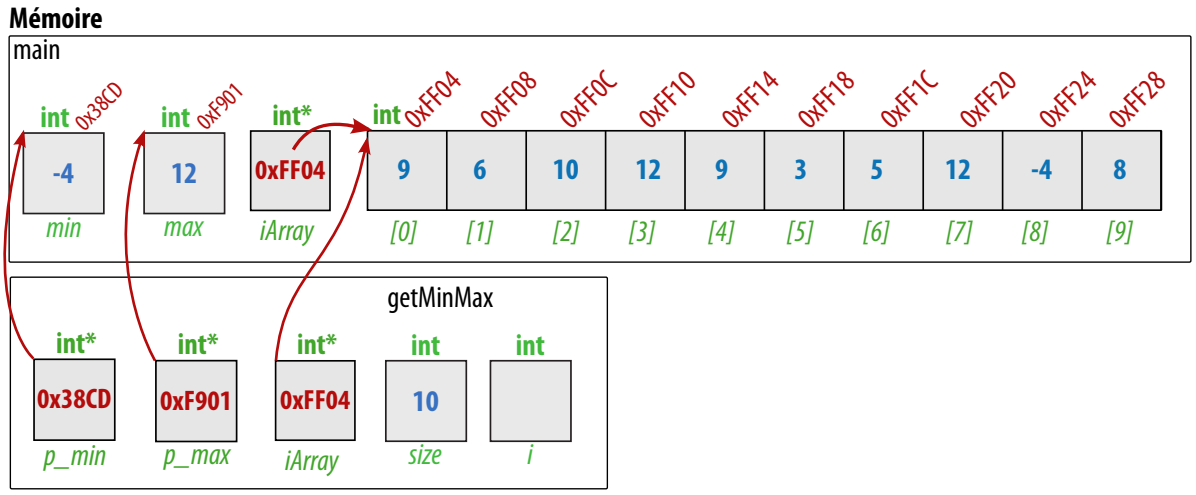
Lors de la déclaration d'un tableau de `n` cases dans un programme, le système d'exploitation réserve `n` espaces mémoire pour le contenu PLUS un espace mémoire **pour le pointeur qui contient l'adresse de la première case**.

Nouvelle représentation :



Reprenons l'exemple de la fonction `getMinMax` : nous voyons que nous passons l'adresse de la première case du tableau (contenue dans le pointeur `iArray` du `main`) en argument du paramètre `iArray` de la fonction. Cela nous permet d'accéder (en lecture et en écriture) aux cases du tableau déclaré dans le `main` depuis la fonction.

Comme `p_min` et `p_max` font aussi référence à des variables du `main`, la seule variable (non pointeur) déclarée dans la fonction est la variable `size` !



Comme monsieur Jourdain, nous utilisions les pointeurs sans le savoir !

QUESTION 9

En utilisant ce que vous avez vu dans ce TD réécrivez les prototypes des procédures suivantes (elles renvoient toutes `void`) :

- `getMean`,
- `getMinMax`
- `getMinMaxIndex`
- `getMode`
- `getMedian`
- `getMeanModeMedian`

EXERCICE 7

Codez les fonctions dans un même fichier :

- `getMinMax` qui affecte aux variables dont les adresses sont passées à la fonction, les **valeurs** des éléments minimum et maximum du tableau.
- `getMinMaxIndex` qui affecte aux variables dont les adresses sont passées à la fonction, les **indices** des éléments minimum et maximum du tableau.
- `int getFirstIndexOfValue(int size, int iArray[], int value, int* p_index)` qui assigne à la variable dont l'adresse `p_index` est passée à la fonction, l'indice correspondant à la première apparition de la valeur `value` dans le tableau. En cas de succès, la fonction renvoie 1 (OK), si la valeur n'est pas dans le tableau la variable **ne doit pas être modifiée** mais la fonction doit retourner 0 (KO).

Remarque : Votre `main` vous servira à tester ces fonctions.
Vous soumettrez vos fonctions au correcteur automatique.

Mémo

« Chaque espace mémoire est identifié par une adresse. »
« Les variables sont identifiées par un nom, un type et une adresse. »
« Nous obtenons une adresse avec l'opérateur & (esperluette). »
« Un pointeur est une variable qui contient une adresse. »
*« Nous accédons au contenu d'une adresse avec l'opérateur * (étoile). »*
« Deux manières de passer des arguments en paramètre à une fonction : par valeur ou par adresse. »
« Le nom d'un tableau correspond à l'adresse de la première case du tableau. » « Deux manières de passer des arguments en paramètre à une fonction : par valeur ou par adresse. »