Mémoire et pointeurs
Initiation à la magie noire !

Julien Baste

IUT de Lille – Université de Lille

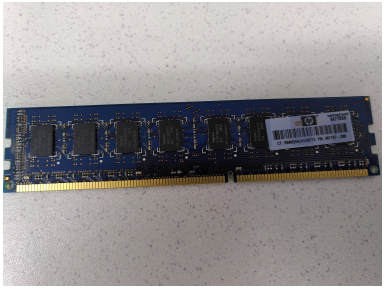
Séance 02

2022/2023

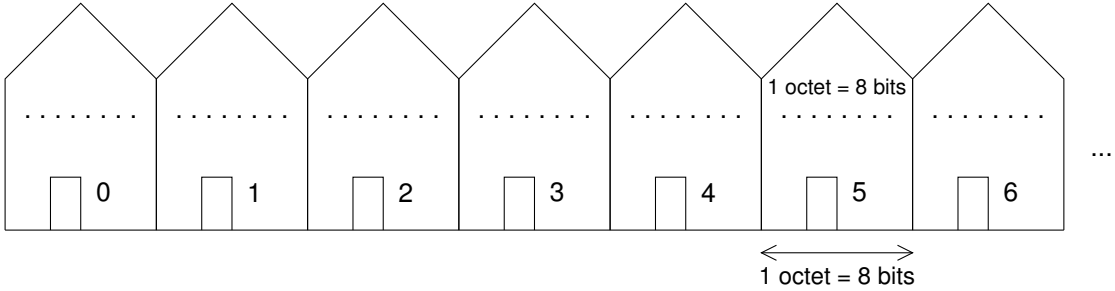
Que fait-on aujourd'hui ?

Le plan aujourd'hui est le suivant :

- Compréhension de la mémoire
- Découverte des pointeurs



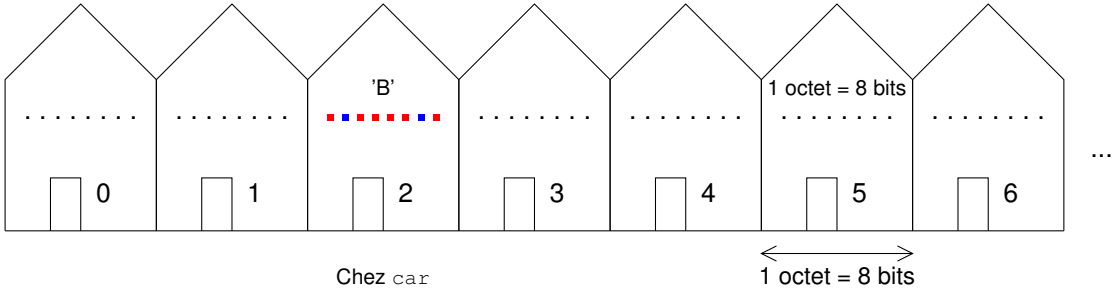
La grande rue de la mémoire



La mémoire est comme une grande rue :

- chaque maison (ou case mémoire) contient 8 bits de données (soit un octet)
- chaque maison (ou case mémoire) à une adresse dans la rue (ici de 0 à 6)

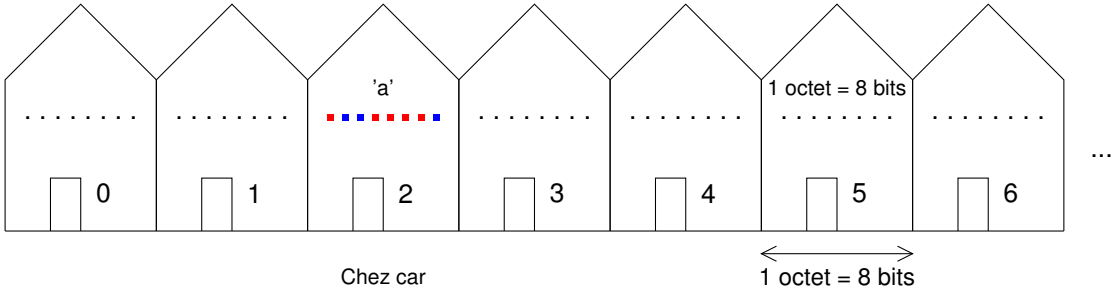
Que stocker dans ces maisons ?



Intuitivement, on peut stocker une variable.
ex : `char car = 'B';` (rappel : le code ascii de 'B' est 0x42)

- le contenu de car se trouve à l'adresse 2. (adresse exprimée en général en hexadécimal.)
- le contenu s'obtient en écrivant `car`.
- l'adresse de car s'obtient, en C, en écrivant `&car`.

Que stocker dans ces maisons ?



Si je modifie `car` :

```
char car = 'B';
car = 'a';
```

- l'adresse de `car` ne change pas, Il habite toujours à l'adresse 2.
- son contenu change (Il redécore chez lui).

Un exemple

```c
#include <stdio.h>

int main (void)
{
  char car = 'B';

  printf("car vaut %c\n",car);
  printf("adresse de car : %p\n",&car);

  printf("\nJe modifie car\n\n");
  car = 'a';

  printf("car vaut %c\n",car);
  printf("adresse de car : %p\n",&car);

  return 0;
}
```

```
car vaut B
adresse de car : 0x7ffc063697b7

Je modifie car

car vaut a
adresse de car : 0x7ffc063697b7
```

De combien de maisons a besoin ma variable ?

Une seule maison = seulement 8 bits (entre 0 et 255).
Si l'on veut stocker des valeurs plus grandes, il nous faut plus de maisons.

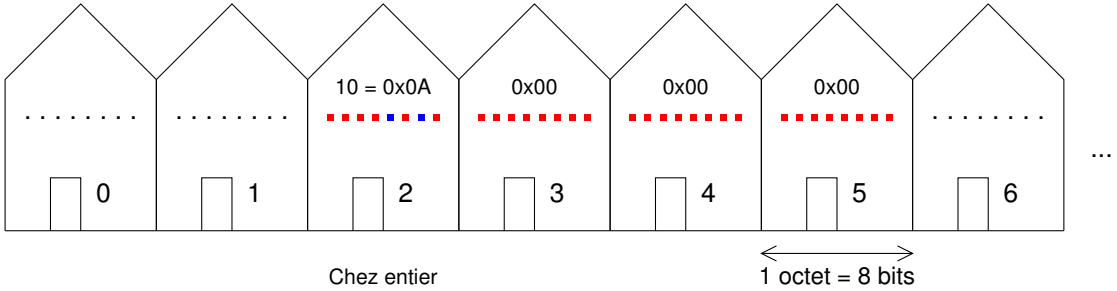
La fonction qui permet de savoir combien d'octets nécessitent une variable est **`sizeof`**.
ATTENTION : **`sizeof`** retourne un "long unsigned int" que l'on affiche avec `%ld`.

```c
#include <stdio.h>

int main (void)
{
  printf("size of char : %ld octet\n", sizeof(char));
  printf("size of int : %ld octets\n", sizeof(int));
  printf("size of float : %ld octets\n", sizeof(float));

  return 0;
}
```

```
size of char : 1 octet
size of int : 4 octets
size of float : 4 octets
```

Note : Nous verrons plus tard pour la taille des tableaux et chaîne de caractères.

Que stocker dans ces maisons ?

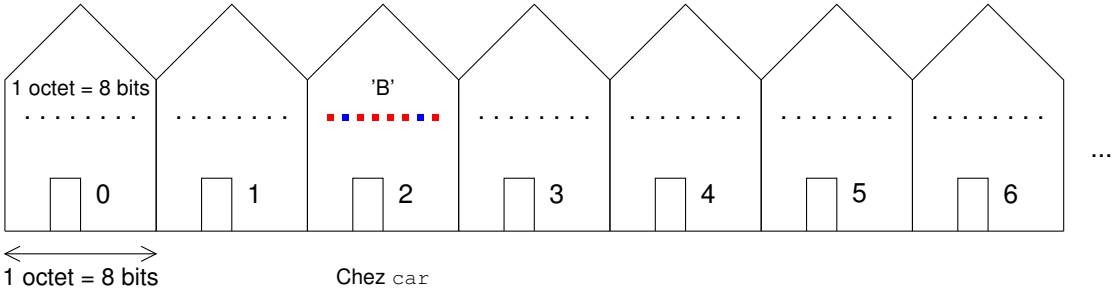


ex : `int entier = 10;`

- L'adresse d'`entier` est 2, la première adresse des 4 maisons (2, 3, 4, et 5) qu'il occupe.
- Un entier nécessite 4 cases mémoires (donc 4 maisons occupées).
- L'entier est écrit en little-endian. (Les octets de poids faible en premier cf. R1.03.)
  0x0000000A est écrit 0x0A 0x00 0x00 0x00

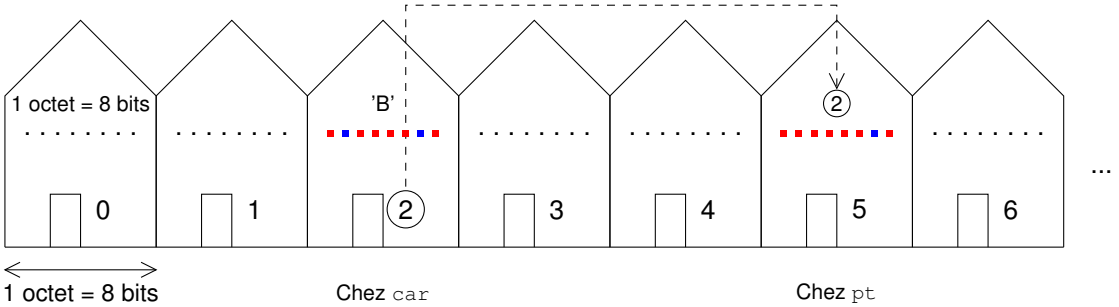
Note : Modifier le contenu de la case mémoire 3, modifie aussi le contenu de `entier`.

Le principe des pointeurs



- le contenu de `car` (donc 'B') se trouve à l'adresse 2.
- l'adresse de `car` s'obtient, en C, en écrivant `&car`.

Le principe des pointeurs



- le contenu de `car` (donc 'B') se trouve à l'adresse 2.
- l'adresse de `car` s'obtient, en C, en écrivant `&car`.

Idée : créer une variable (ex : `pt`) qui contient l'adresse de `car` !

Une telle variable s'appelle un pointeur.

- `pt` contient l'adresse de `car` (donc l'adresse 2)
- `*pt` correspond au contenu de `car` (donc 'B')
  (On parle de déréférencer un pointeur)

Déclarations de pointeurs

Un pointeur est déclaré en fonction de la cible du pointeur.

Si mon pointeur `pt` cible (contient l'adresse d') :

- un **`char`** alors je le déclare ainsi : **`char`**`* pt;`
- un **`int`** alors je le déclare ainsi : **`int`**`* pt;`
- un **`float`** alors je le déclare ainsi : **`float`**`* pt;`

Globalement : `type* pt;` permets de déclarer un pointeur vers une case mémoire contenant un objet de type `type`.

ATTENTION : Il y a deux utilisations différentes de `*` :

- Le "`*`" de la déclaration de pointeur.
- L'opérateur de déréférencement "`*`".

NOTE : Nous utiliserons **`int`**`* pt;` dans ce cours.
L'utilisation de **`int`** `*pt` est aussi correcte et très courante.

Un exemple de pointeur

```c
#include <stdio.h>

int main (void)
{
  int entier = 10;
  int* pointeur;

  pointeur = &entier;

  printf("adresse de entier      : %p\n", &entier);
  printf("pointeur vaut          : %p\n\n", pointeur);
  printf("entier vaut                          : %d\n", entier);
  printf("Me déréférençement de pointeur est : %d\n", *pointeur);
  *pointeur = 12;
  printf("\n*pointeur = 12;\n");
  printf("entier vaut                          : %d\n", entier);
  printf("Me déréférençement de pointeur est : %d\n", *pointeur);
  return 0;
}
```

```
adresse de entier      : 0x7ffe1972724c
pointeur vaut          : 0x7ffe1972724c

entier vaut                          : 10
Me déréférençement de pointeur est : 10

*pointeur = 12;
entier vaut                          : 12
Me déréférençement de pointeur est : 12
```

Retour sur les fonctions

Concernant les arguments d'une fonction.

- Une fonction ne modifie pas ses arguments.
  (elle travaille sur une copie de la case mémoire/maison donnée comme argument.)
- On dit que les arguments sont passés par valeur.
- Les arguments d'une fonction doivent être considérés comme des constantes.

Concernant la valeur de retour :

- On ne peut retourner qu'une seule valeur.

ASTUCE : Vous ne pouvez pas modifier un argument,
Mais vous pouvez toujours modifier la case mémoire ciblée par un pointeur !

Pointeurs et fonctions

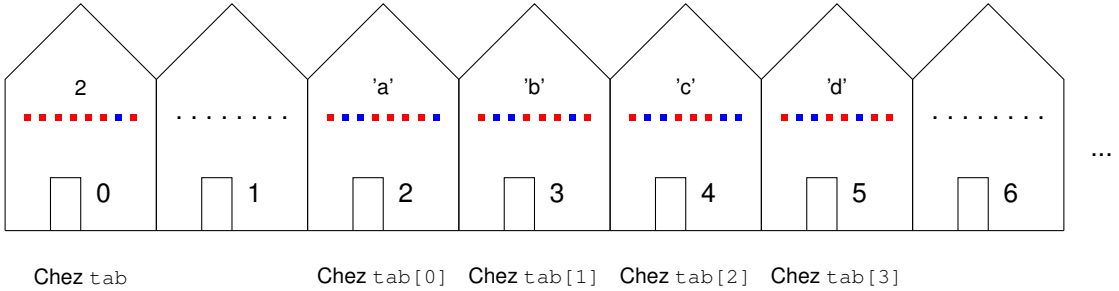
Pour utiliser des pointeurs dans une fonction :

- On déclare que l'argument attendu est un pointeur :
  ```
  int fonction(int* pointeur);
  ```
- On modifie ce qu'il y a à l'adresse indiquée par le pointeur :
  ```
  *pointeur = 4;
  ```
- La valeur de retour ne nous intéresse pas ici.
- Notez que l'on n'a pas modifié l'argument, `pointeur` mais uniquement la case mémoire/maison ciblée par le pointeur.
- Désormais si l'on appelle `fonction` avec en argument un pointeur, le contenu de la case mémoire ciblée par le pointeur vaudra 4.

Exemple : Pointeurs et fonctions

```c
#include <stdio.h>

int par_valeur (int a);
int par_pointeur (int* pt_b);

int main (void)
{
  int a = 0;
  int b = 0;
  int* pt_b = &b;
  par_valeur(a);
  par_pointeur(pt_b);
  printf("Dans le main,          a vaut : %d\n",a); // Valeur affichée : 0
  printf("Dans le main,          b vaut : %d\n",b); // Valeur affichée : 1
  return 0;
}

int par_valeur (int a)
{
  a = a+1;
  printf("Dans la fonction,      a vaut : %d\n",a); // Valeur affichée : 1
  return 0;
}

int par_pointeur (int* pt_b)
{
  *pt_b = *pt_b+1;
  printf("Dans la fonction, *pt_b vaut : %d\n",*pt_b); // Valeur affichée : 1
  return 0;
}
```

Les tableaux



Déclaration d'un tableau de taille 4 : `char tab[4] = {'a','b','c','d'};`

- En pratique `tab` est simplement un pointeur contenant l'adresse de `tab[0]`.
- Les cases d'un tableau sont placées à la suite dans la mémoire.
- L'adresse de `tab[2]` correspond à l'adresse de `tab[0]` plus 2.
- L'adresse de `tab[i]` est donc `tab+i`.
- Le contenu de `tab[i]` peut être accédé via `*(tab+i)`.

Exemple : Tableaux et pointeurs

```c
#include <stdio.h>

int main (void)
{
  int i;
  char tab[4] = {'a','b','c','d'};
  char* pt = &tab[0];

  *(pt+2) = 'e';
  *(tab+3) = 'f';

    for (i=0; i<4; i++)
    {
      printf("tab[%d] = %c\n", i, tab[i]);
    }

  return 0;
}
```

```
tab[0] = a
tab[1] = b
tab[2] = e
tab[3] = f
```

Tableaux et arguments

Comment passer un tableau en argument d'une fonction ?

- Vous devez donner comme argument le pointeur vers le début du tableau.
- Il n'y a pas de fonction length en C,
  vous aurez donc souvent besoin de passer en deuxième argument la taille du tableau.

Comment utiliser mon tableau passé en argument ?

- Dans la fonction vous pouvez vous en servir comme vous utiliseriez un tableau normalement.
- Ou alors avec la notation en pointeur.
- Attention à toujours savoir où est la fin de votre tableau.

NOTE : Si vous passez un tableau à deux dimensions (ou plus) en argument, vous perdez son aspect bidimensionnel !

Exemple : Tableaux et arguments

```c
#include <stdio.h>

int affiche_tab(int* tableau, int taille);

int main (void)
{
  int tab[4] = {10,11,12,13};

  affiche_tab(tab,4);

  return 0;
}

int affiche_tab(int* tableau, int taille)
{
  int i;
  for (i=0; i<taille; i++)
    {
      printf("tableau[%d] = %d\n", i, tableau[i]);
    }
  return 0;
}
```

```
tableau[0] = 10
tableau[1] = 11
tableau[2] = 12
tableau[3] = 13
```

C’est tout pour aujourd’hui