ECE PARIS • LYON
ÉCOLE D'INGÉNIEURS

# Algorithmique et programmation structurée en C

Cours n°3

# Langage C
# Mémoire, adresses, pointeurs

#  Plan du cours

1. **Mémoire centrale**
2. **Adresses mémoire**
3. **Pointeurs**

# Mémoire Centrale

**Rappel :** dans un programme, les variables sont stockées en mémoire centrale (mémoire vive / RAM) afin d'être utilisables par le processeur.

## La mémoire centrale est composée de plusieurs mots binaires.

De nos jours, un mot binaire fait **un octet** (8 bits). On dit alors que **l'octet est la plus petite unité de mémoire adressable**.

## Chaque mot binaire/octet possède une **adresse**.

Une adresse tous les 8 bits.

## Les adresses sont écrites au format **Hexadécimal** (alphabet : `0-9,a-f`).

**Exemple :** `ef3c4a68`, noté `0xef3c4a68` (`0x` précise le format **hexadécimal**).

# Le format Hexadécimal

**Base : 16**

**Alphabet : 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.**

***NB :*** *possibilité d'écrire les lettres en majuscules ou en minuscules.*

## Comptons en Hexadécimal :

```
  0,   1,   2,   3,   4,   5,   6,   7,   8,   9,   A,   B,   C,   D,   E,   F,
 10,  11,  12,  13,  14,  15,  16,  17,  18,  19,  1A,  1B,  1C,  1D,  1E,  1F,
 20,  21,  22,  23,  24,  25,  26,  27,  28,  29,  2A,  2B,  2C,  2D,  2E,  2F,
 3…,       4…,       5…,       6…,       7…,       8…,
 90,  91,  92,  93,  94,  95,  96,  97,  98,  99,  9A,  9B,  9C,  9D,  9E,  9F,
 A0,  A1,  A2,  …,                                          FC,  FD,  FE,  FF,
100, 101, 102, 103, …
```

# Le format Hexadécimal

## Exemples d'affichage

On peut afficher une adresse au format hexadécimal dans un **`printf`** en utilisant le *placeholder* **`%x`**.

```
printf("Affichage d'un entier relatif (%%d) : %d %d\n", 15, 0x15); // Affiche :
// Affichage d'un entier relatif (%d) : 15 21


printf("Affichage d'un entier exprimé en hexadécimal (%%x) : %x %x\n", 15, 0x15); // Affiche
: // Affichage d'un entier exprimé en hexadécimal (%x) : f 15
```

💡 **`%%d`**! permet d'afficher **`%d`** au lieu de le remplacer par une valeur.

# Mémoire

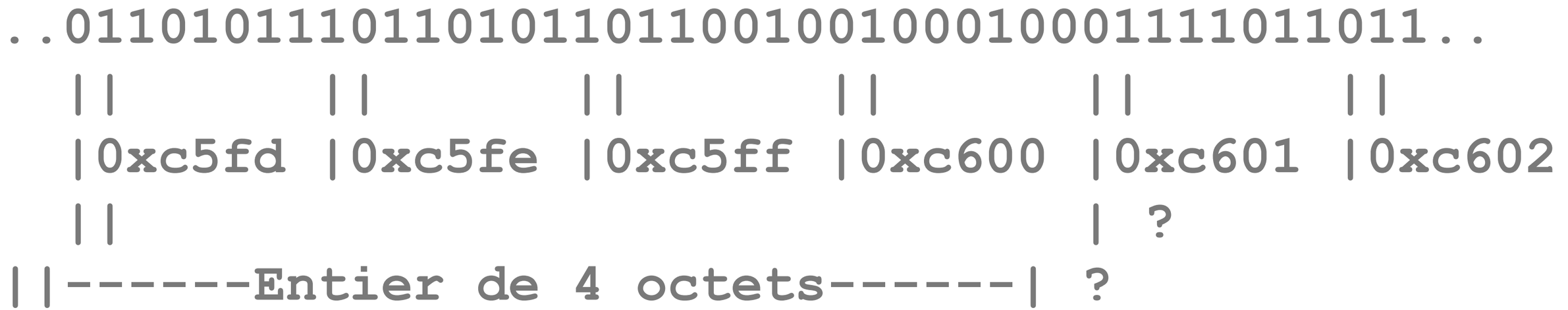
- Chaque variable occupe 1 ou plusieurs mots mémoires (octets).
- Le type de la variable détermine sa taille en mots mémoires (octets).

  Un **`int`** occupe 4 octets, un **`char`** occupe 1 seul octet…
- Sans bidouillage, un mot mémoire ne peut pas être partagé par plusieurs variables.

  Ainsi, pour représenter un simple booléen (1 bit), on est obligé d'utiliser un un mot mémoire entier (1 octet : plus petite unité adressable en mémoire).
- L'adresse d'une variable occupant plusieurs mots mémoires est celle de son premier mot mémoire.
- Les mots mémoire d'une même variable sont contigus dans la mémoire.

  On ne peut pas avoir le début à l'adresse **`0x4a68`**, d'autres variables au milieu, puis la suite à l'adresse **`0xff08`**.

# Mémoire - Exemple d'un entier en mémoire centrale




Chaque octet a sa propre adresse.

L'entier, stocké sur 4 octets, a pour adresse : **0xc5fd**

| Adresse | Bits en mémoire | | | | | | | | Valeur |
|---|---|---|---|---|---|---|---|---|---|
| .... | ... | | | | | | | | ... |
| 0xef3c4a68 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | Quelque chose |
| 0xef3c4a69 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | Quelque chose |
| 0xef3c4a6a | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Un entier<br>- d'adresse : 0xef3c4a6a<br>- de valeur : 11 |
| 0xef3c4a6b | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 0xef3c4a6c | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 0xef3c4a6d | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | |
| 0xef3c4a6e | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Quelque chose |
| ... | ... | | | | | | | | ... |

Chaque mot mémoire (ici d'un octet) possède une adresse. Un entier occupe généralement 4 octets.
Il occupera donc ici 4 mots mémoires (donc 4 octets).

# Adresses

On obtient l'**adresse** d'une variable avec l'opérateur unaire[1] "**&**" (esperluette)

```
int a = 11;

printf("%d", a);  // affiche la valeur de a

printf("%x", &a); // affiche l'adresse de a (&a) au format hexadécimal
```

Nous avons déjà utilisé l'esperluette — et donc les adresses — avec **`scanf()`**.

**À votre avis, pourquoi la fonction scanf a-t-elle besoin de l'adresse de la variable du (sous-)programme qui l'appelle ?**

```c
int main() {
    int a = 0;
    scanf("%d", &a); // en appelant scanf, la fonction main
}                    // lui fournit l'adresse de sa variable a
```

⊙ Imaginons que l'on crée une version simplifiée de **`scanf`** sans utiliser d'adresse.

```
void scanfMaison(/* … */, int variableAModifier) { // On reçoit une copie de la
                                                   // valeur 0 dans variableAModifier
    // code compliqué
    variableAModifier = 5; // On suppose que l'utilisateur a tapé 5, la copie est modifiée
}

int main(void) {
    int a = 0;              // a vaut 0
    scanfMaison("%d", a);
    printf("%d", a);        // a vaut toujours 0 (c'est une copie qui a pris la valeur 5), affiche 0
    return 0;
}
```

**Dans le main, pourquoi `a` ne vaut pas `5` après l'appel de `scanfMaison` ?**

⊙ Car **`a`** est **passé par valeur**.

On copie sa valeur dans la variable de la fonction appelée (**`scanfMaison`**).

- Le sous-programme **`scanfMaison`** travaille donc sur une **copie de la valeur** de la variable **`a`** du main (une copie de la valeur **`0`**, qui s'appelle **`variableAModifier`** à l'intérieur de **`scanfMaison`**).

Pour que la procédure **`scanfMaison`** puisse modifier directement la valeur de la variable **`a`** du **`main`**, ce dernier ne va non plus lui transmettre une copie de la valeur de sa variable **`a`** (0) mais une **copie de son adresse en mémoire** (ex : **`0x3ba1`**). Ainsi, la procédure **`scanfMaison`** pourra modifier directement l'espace mémoire de la variable **`a`** du **`main`** :

```c
void scanfMaison(/* … */, int* adrVarAModifier) { // adrVarAModifier reçoit comme valeur l'adresse 0x3ba1"%#$ !&
                                                  // code compliqué
    *variableAModifier = 5; // On suppose que l'utilisateur a tapé 5,
                            // on écrit la valeur 5 à l'adresse contenue dans adrVarAModifier (en 0x3ba1)
                            // (l'adresse de la variable a du main)
}

int main(void) {
    int a = 0;              //  a vaut 0, on suppose qu'elle est à l'adresse
    0x3ba1     scanfMaison("%d", &a);  //  Nous passons 0x3ba1 en paramètre
    printf("%d", a);        //  a vaut maintenant 5  le printf affiche 5
    return 0;
}
```

Revenons à la fonction **scanf**.

Le **main** lui fournit bien l'adresse de sa variable **a** (et non pas sa valeur), afin que le **scanf** puisse modifier directement sa valeur.

```c
scanf("%d", &a); // ici, %d indique à scanf que l'on souhaite écrire
      un                  // entier (4 octets), à partir de l'octet
                                                  d'adresse &a
```

Sans ce mécanisme, une fonction ne peut modifier que ses propres variables (celles déclarées dans son **bloc** *(ou de façon globale (pas bien))*).

# Pointeurs

## Définition :

**Un pointeur est une variable qui stocke une adresse mémoire.**

La valeur d'un pointeur est une adresse. Un pointeur dispose comme toute variable de sa propre adresse.

```
int n = 10;    // n est un entier (un int), sa valeur est un entier : 10,
               // son adresse mémoire est – par exemple - 0x9da3 (elle occupe 4 octets à partir de cette adresse).

int* pN = &n; // pN est un pointeur sur un entier (un int*), sa valeur est l'adresse d'un entier (l'adresse de n) : 0x9da3,
              // son adresse mémoire est - par exemple - 0xa58c (chaque variable a sa propre adresse).
```

## Vocabulaire :

On dit qu'un pointeur pointe sur $X$ s'il a pour valeur l'adresse de $X$. Ici, `pN` pointe sur `n`.

# Pointeurs

Pour qu'une variable ne stocke non plus une valeur, mais une adresse, on ajoute un **astérisque (*)** à droite du type de ce dont on veut stocker l'adresse. C'est alors un **pointeur**.

| | Type | Valeur/Contenu |
|---|---|---|
| ***Pointeur sur* un entier** (sur un `int`) | `int*` | Adresse d'un `int` |
| ***Pointeur sur* un nombre décimal** (sur un `float`) | `float*` | Adresse d'un `float` |
| ***Pointeur sur* un pointeur sur un entier** (sur un `int*`) | `int**` | Adresse d'un `int*` |

**NB :** Le type d'un pointeur a un astérisque de plus que le type de la variable sur laquelle il pointe.

# Pointeurs - Déclaration

La position de l'astérisque n'a pas d'importance :

```
int* pMonPointeur;  // (conseillé)
// --- OU ---
int *pMonPointeur;  // (très répandu)
// --- OU ---
int * pMonPointeur; // (rare)
// --- OU --int*pMonPointeur;
// (rare)
```

Une bonne pratique consiste à nommer les pointeurs en commençant par un "`p`" (notation hongroise).

🚨 **ATTENTION :** il faut toujours initialiser ses variables, surtout les pointeurs !

# Pointeurs - Initialisation

😌 Pensez à toujours initialiser vos variables, surtout les pointeurs ! Un pointeur peut ne pointer sur rien. On lui donne alors la valeur **NULL**.

```
int a = 11;
int* pMonPointeur = &a; // pointeur qui pointe sur a

ou int* pMonPointeur = NULL;// pointeur qui ne pointe encore sur rien (0x0)

if (pMonPointeur != NULL) { … } // il faut alors tester si notre
```

Au cours du programme, un pointeur peut pointer sur plusieurs variables.

Sauf s'il est constant (`int* const pInt`).

Un `int*` ne peut pointer que sur un `int`.

```
int a = 5, b = 10; int* pInt = NULL; // pInt ne pointe sur rien

if (pInt == NULL) { // exemple de test (facultatif)
     pInt = &a; // pInt pointe maintenant sur a
}

pInt = &b; // pInt pointe maintenant sur b
```

# **Pointeurs -** Accès à la valeur à l'adresse pointée

On accède à la valeur se trouvant à l'adresse **pointée** avec l'opérateur unaire ***** :

```
int entier = 5; int* pEntier = &entier; // pEntier pointe sur entier

printf("%x\n", pEntier); // affiche la valeur de pEntier, donc l'adresse d'entier

printf("%d\n", *pEntier); // affiche 5 (la valeur pointée par pEntier)

printf("%x\n", &pEntier); // affiche l'adresse du pointeur pEntier (&pEntier)
```

🚨 **ATTENTION :** ne pas confondre avec la déclaration du pointeur (**`int* pEntier;`**), ni avec la multiplication (**`a*b`**).

# Opérateur * - Trois utilisations à distinguer

```
// On crée un pointeur qui pointe sur l'entier a
```

- Pour déclarer un pointeur : `int* p = &a;`

- Pour accéder à la valeur pointée :

```
*p = 5; // a vaut maintenant 5
```

- Pour multiplier :

```
3**p // Donne comme résultat 15
```

# Pointeurs - Exemple

## Comment accéder à la valeur/l'adresse ?

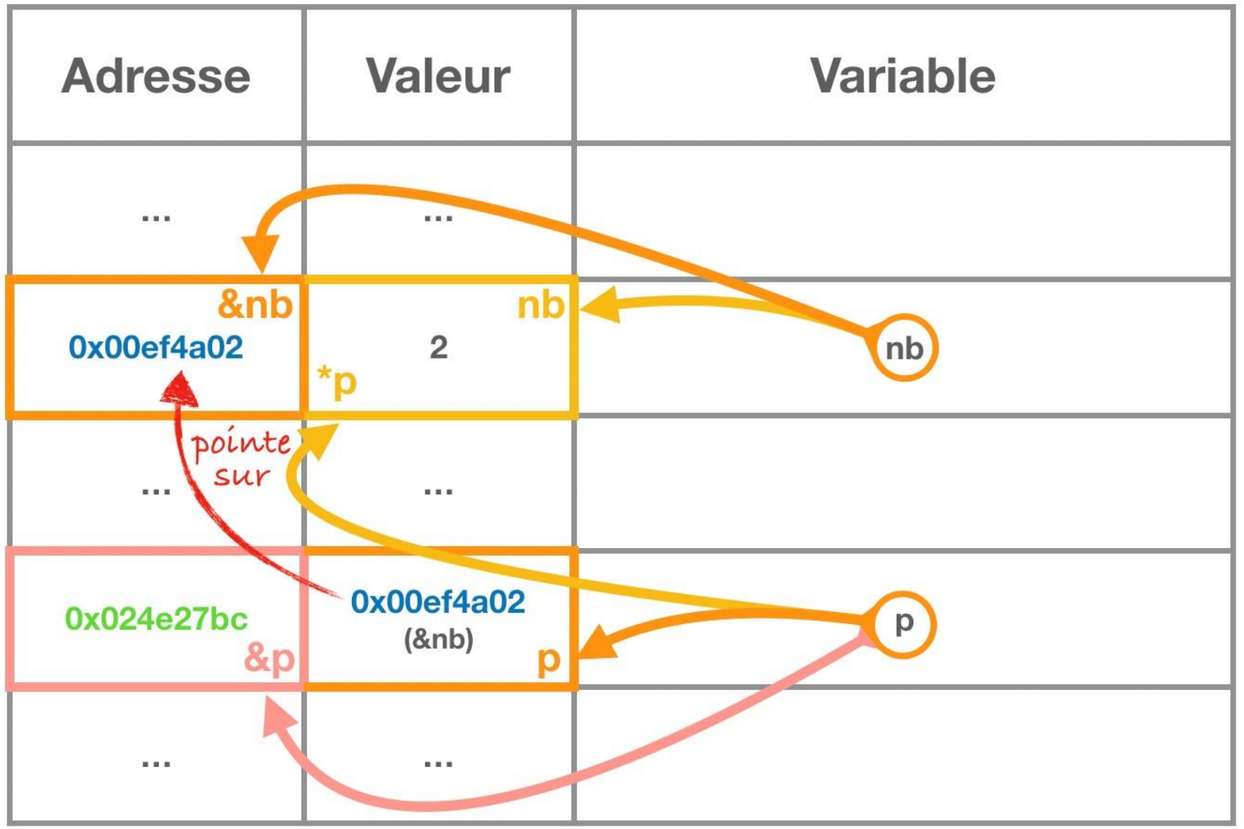
```c
#include <stdio.h>

int main(void) {

    int nb = 2;

    int* p = &nb;

    return 0;
}
```



# Pointeurs sur pointeurs sur …

```
int entier = 5;
int* pEntier = &entier;      // pointeur sur un entier
int** ppEntier = &pEntier;   // pointeur sur un pointeur sur un entier

printf("%x\n", pEntier);    // affiche l'adresse de entier

printf("%d\n", *pEntier);    // affiche 5 (la valeur à l'adresse

pointée) printf("%x\n", ppEntier);    // affiche l'adresse de pEntier

printf("%x\n", *ppEntier);   // affiche l'adresse de entier
                             // (la valeur de pEntier)
printf("%d\n", **ppEntier); // affiche 5
```

Compliqué... On reviendra dessus avec les listes chaînées ! Pour bien comprendre, **faites des schémas**.

# Pointeurs - Résumé

- La valeur d'un pointeur est une adresse.
- Si `p` pointe sur `x`, alors la valeur de `p` correspond à l'adresse de `x` (`&x`).
- Si `p` pointe sur `x`, alors toute modification de `*p` modifie `x` et toute modification de `x` modifie `*p` puisque `*p` et `x` correspondent au même espace mémoire.

# Pointeurs - Récapitulatif sur un court programme

```
int nb = 5;
int* p = &nb;
```

| | Valeur | Exemple(s) d'utilisation |
|---|---|---|
| `nb` | La valeur de la variable **nb** : **5**. | `nb = 5;`<br>`printf("%d", nb);` |
| `&nb` | L'adresse en mémoire de la variable **nb**. | `int* p = &nb;`<br>`scanf("%d", &nb);` |
| `p` | La valeur du pointeur **p**, donc l'adresse d'un entier, celle de l'entier **nb** (car `p = &nb`). | `p = &nb;` |
| `*p` | La valeur en mémoire là où pointe **p**, donc **5**. | `*p = 6;` (**nb** vaut maintenant **6**)<br>`scanf("%d", p);`<br>`printf("%d", *p);` |
| `&p` | L'adresse du pointeur **p**. | `int** pp = &p;` (pointeur sur un pointeur) |

# Pointeurs - Remarque

```c
int n = 0;
int pN = &n;
scanf("%d", &n);
// est équivalent à :
scanf("%d", pN); // pas d'esperluette car contient l'adresse souhaitée
// est équivalent à :
scanf("%d", &*pN);
// est équivalent à :
scanf("%d", &*&*&*&*&*&*&*&*&*pN);
// ...
```

Les combinaisons **&*** et ***&** s'annulent ! Et ça semble logique, l'un accède à l'adresse de la valeur, et l'autre à la valeur de l'adresse.