INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES CENTRE VAL DE LOIRE

# Algorithmique et Langage C
# Mémoire et Pointeurs

Camille BAUDOIN, Moncef HIDANE
{camille.baudoin@insa-cvl.fr, moncef.hidane@insa-cvl.fr}

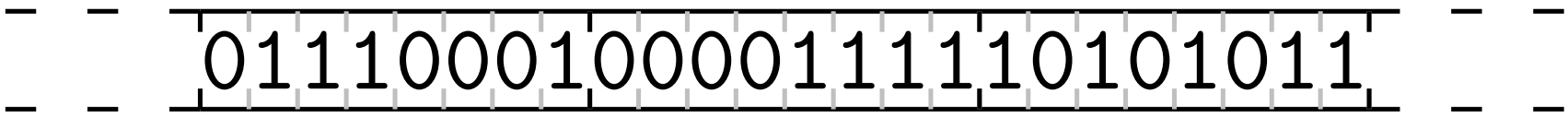
2022-2023

# Rappel sur la mémoire

# Rappel sur la mémoire

## Bande mémoire

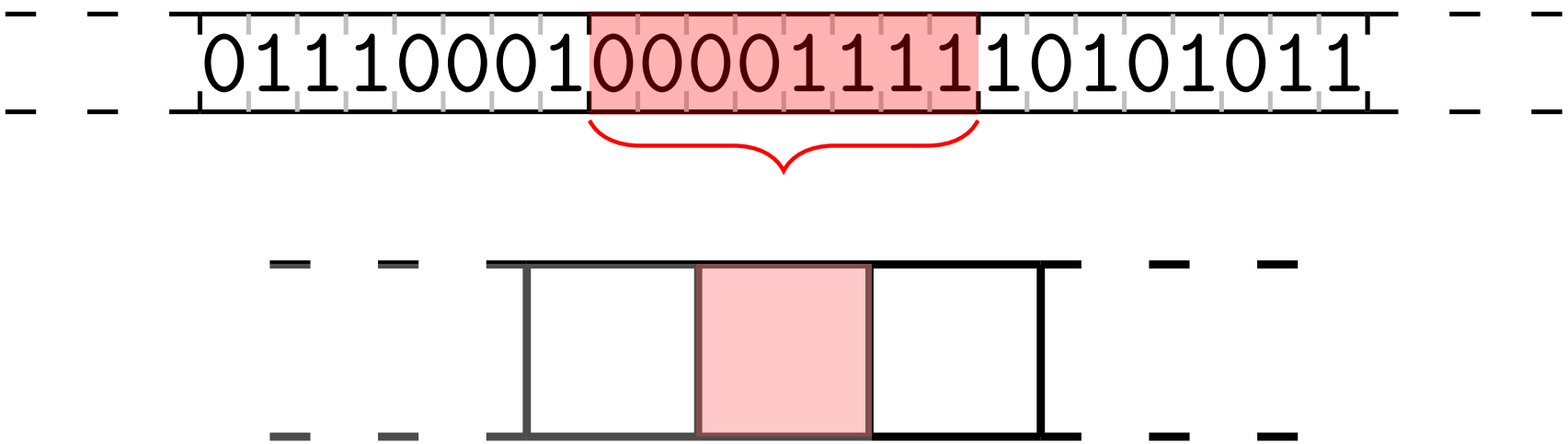
- Mémoire = ensemble de bits (unité la plus petite de mémoire)



# Rappel sur la mémoire

## Bande mémoire

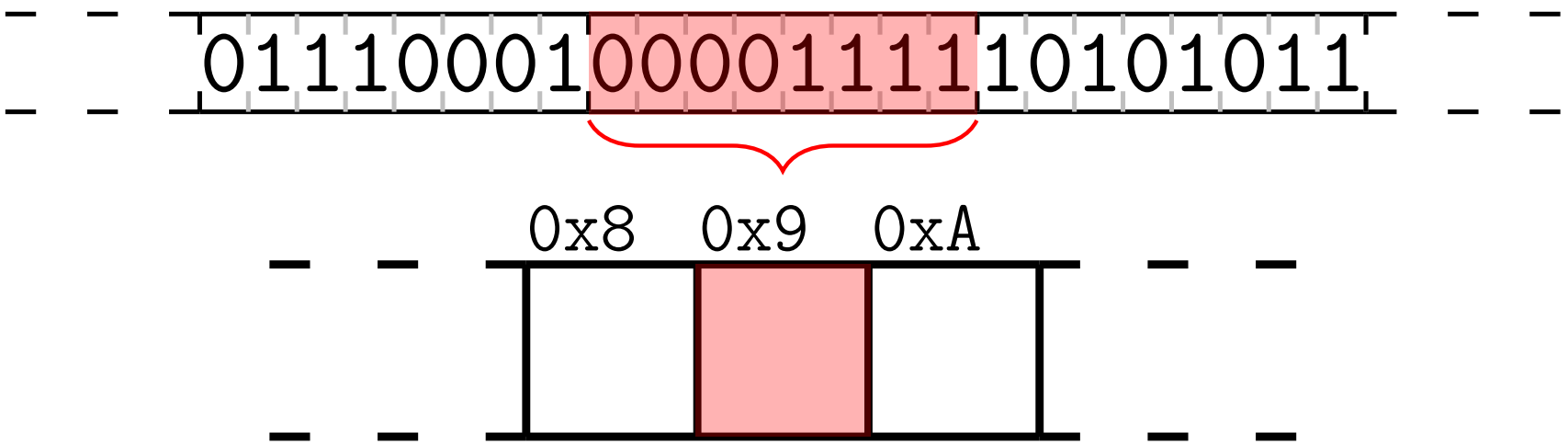
- Mémoire = ensemble de bits (unité la plus petite de mémoire)
- Les bits sont assemblés en octet (8 bits)



# Rappel sur la mémoire

## Bande mémoire

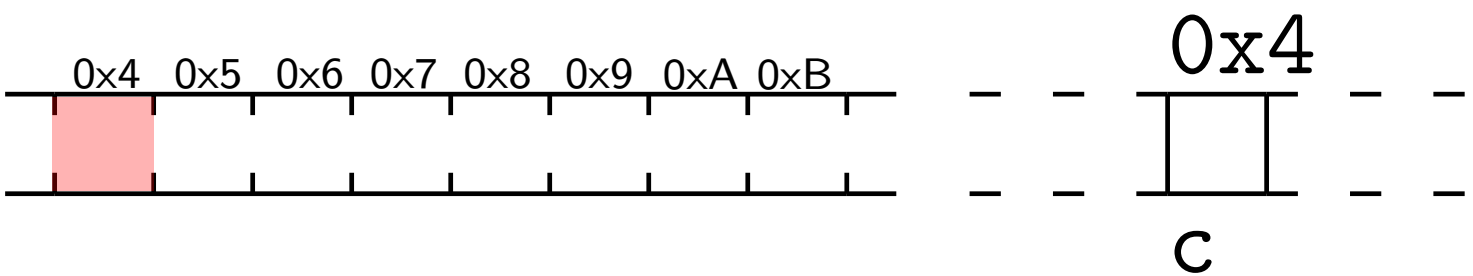
- Mémoire = ensemble de bits (unité la plus petite de mémoire)
- Les bits sont assemblés en octet (8 bits)
- Ces emplacements mémoire (octet) sont identifiés par une adresse unique
- Une adresse mémoire est écrite par convention en hexadécimal (16 valeurs de 0 à F) indiqué par le prefix 0x



# Fonctionnement de la mémoire

## Variables

- Toute déclaration d'une variable entraîne l'allocation d'une ou plusieurs case(s) mémoire
- Le nombre de cases occupé dépend du type de la variable
  - un char occupe 1 octet
- Le compilateur traduit les noms de variables en adresses (celle du premier octet de l'emplacement mémoire consacré à la variable)
- Lorsqu'une variable est lue ou écrite, à l'exécution du programme, le processeur va lire ou écrire à l'adresse correspondante

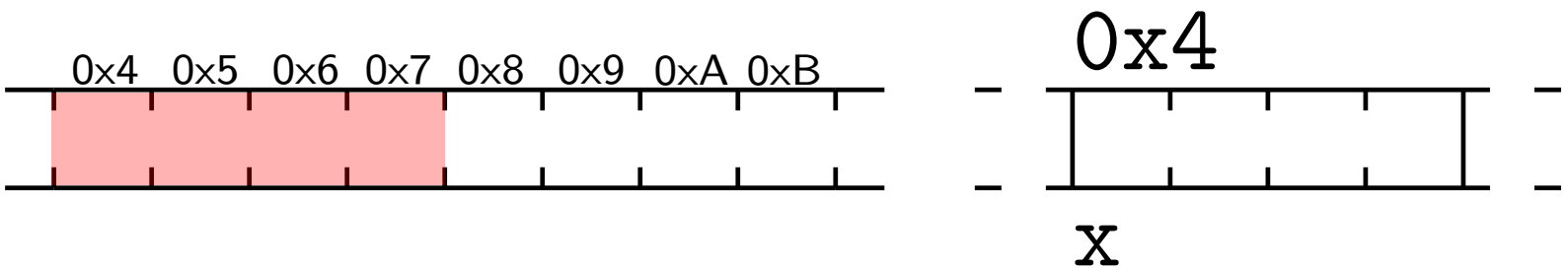


# Fonctionnement de la mémoire

## Variables

- Toute déclaration d'une variable entraîne l'allocation d'une ou plusieurs case(s) mémoire
- Le nombre de cases occupé dépend du type de la variable
  - un char occupe 1 octet
  - un int occupe généralement 4 octets

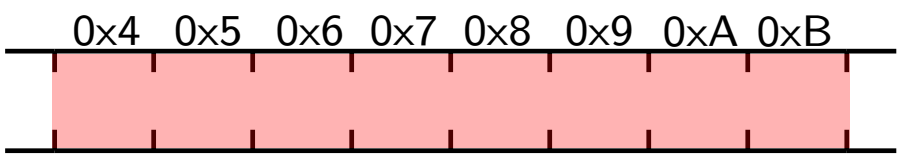
- Le compilateur traduit les noms de variables en adresses (celle du premier octet de l'emplacement mémoire consacré à la variable)
- Lorsqu'une variable est lue ou écrite, à l'exécution du programme, le processeur va lire ou écrire à l'adresse correspondante

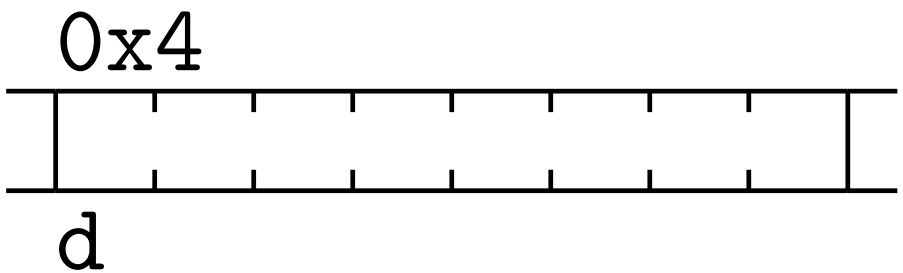


# Fonctionnement de la mémoire

## Variables

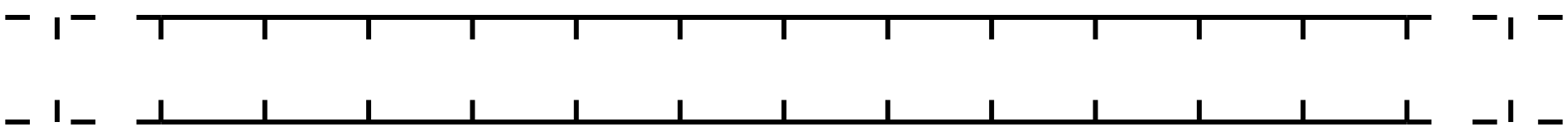
- Toute déclaration d'une variable entraîne l'allocation d'une ou plusieurs case(s) mémoire
- Le nombre de cases occupé dépend du type de la variable
  - un char occupe 1 octet
  - un int occupe généralement 4 octets
  - un double occupe 8 octets
- Le compilateur traduit les noms de variables en adresses (celle du premier octet de l'emplacement mémoire consacré à la variable)
- Lorsqu'une variable est lue ou écrite, à l'exécution du programme, le processeur va lire ou écrire à l'adresse correspondante



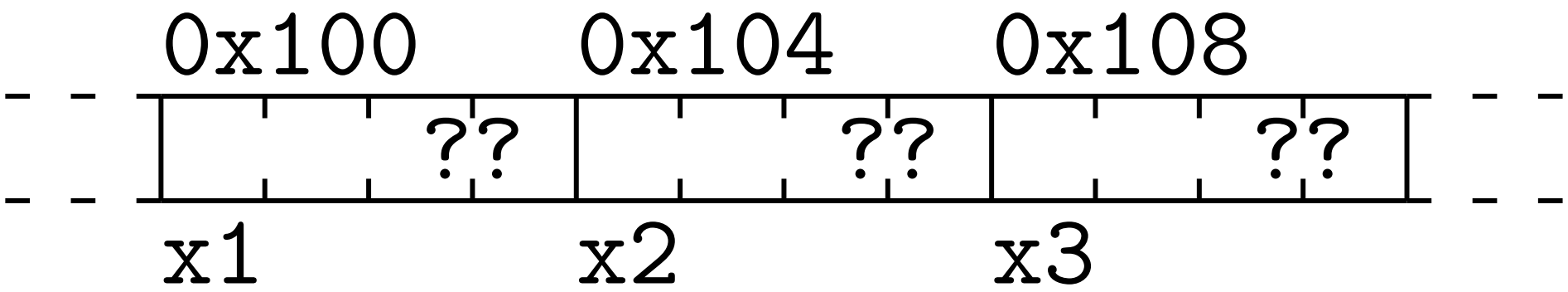
# Exemple

```
int main()
{
 -->
    int x1;
    int x2;
    int x3;
    x1 = 10;
    x2 = 15;
    x3 = x1+x2;
    return 0;
}
```



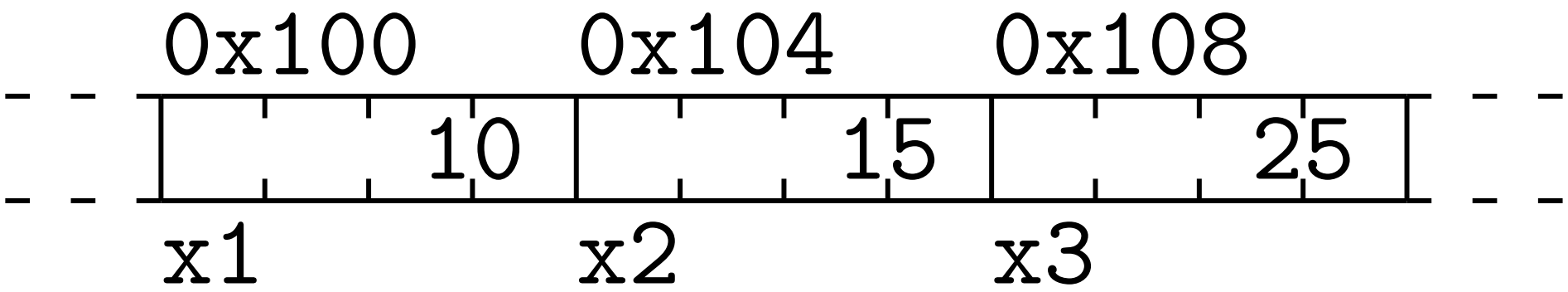
# Exemple

```
int main()
{

    int x1;
    int x2;
 -->int x3;
    x1 = 10;
    x2 = 15;
    x3 = x1+x2;
    return 0;
}
```



# Exemple

```
int main()
{

    int x1;
    int x2;
    int x3;
    x1 = 10;
    x2 = 15;
 -->x3 = x1+x2;
    return 0;
}
```

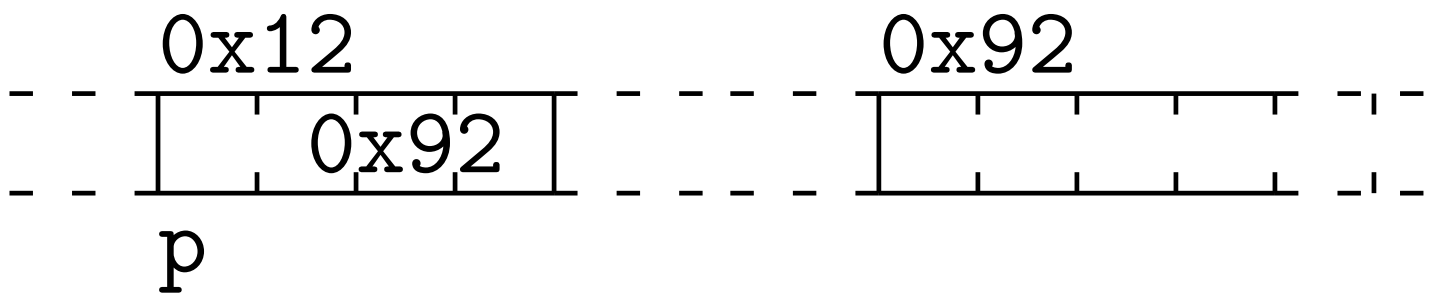


# Les pointeurs

# Manipulation d'adresses mémoires par pointeurs

## Variables

- On peut aussi accéder à un emplacement mémoire directement par son adresse (au lieu d'utiliser la variable assiociée à cet emplacement)
- Pour faire cela, on utilise une variable qui contient l'adresse mémoire d'un autre objet. On appelle cela un *pointeur*
- On dit alors qu'elle "pointe" sur cet emplacement mémoire
- Un pointeur est une variable comme une autre
  - qui possède une adresse (ici `0x12`)
  - à ne pas confondre avec sa valeur : adresse de l'objet pointé (ici `0x92`)
  - taille : 4 ou 8 octets en fonction de l'architecture de l'ordinateur



# Déclaration d'un pointeur

- En C, un pointeur pointe sur un type donné

Syntaxe: `type* id_ptr`

- Syntaxes équivalentes : `type *id_ptr`, `type * id_ptr`
- Le nom de la variable est `id_ptr` et son type `type*`
- Un pointeur contient l'adresse du premier octet de l'emplacement mémoire pointé

```
/* ptr_i (resp. ptr_f) est une variable prévue pour contenir
l'adresse  d'un emplacement mémoire contenant un entier
(resp. un flottant).*/
int* ptr_i; /* Déclaration d'un pointeur sur entier */
float* ptr_f; /* Déclaration d'un pointeur sur float */
```

Lors d'une déclaration multiple de pointeur, un « * » doit être ajouter avant toutes les nouvelles variables

```
int* p1, p2; /* Déclaration d'un pointeur p1 et d'un entier p2 */
int* p3,* p4; /* Déclaration de deux pointeurs */
```

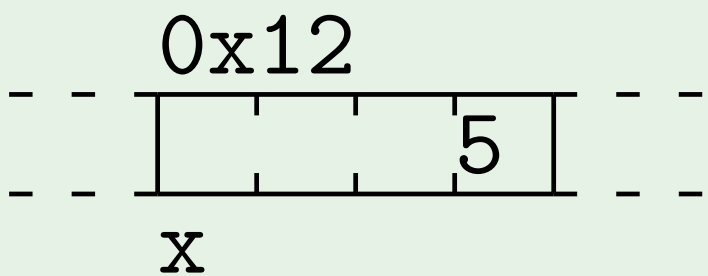
# Affectation d'une valeur à un pointeur

## Opérateur de référencement

- Opérateur permettant de passer d'une variable à son adresse (du premier octet de l'emplacement) opérateur de référencement (ou d'adressage) « & »
- On peut utiliser cette valeur pour l'affecter à un pointeur

## Exemple 1

```
-->int x=5;
   int* p1;
   p1 = &x; /* p1 pointe maintenant sur l'emplacement mémoire de x *\
```



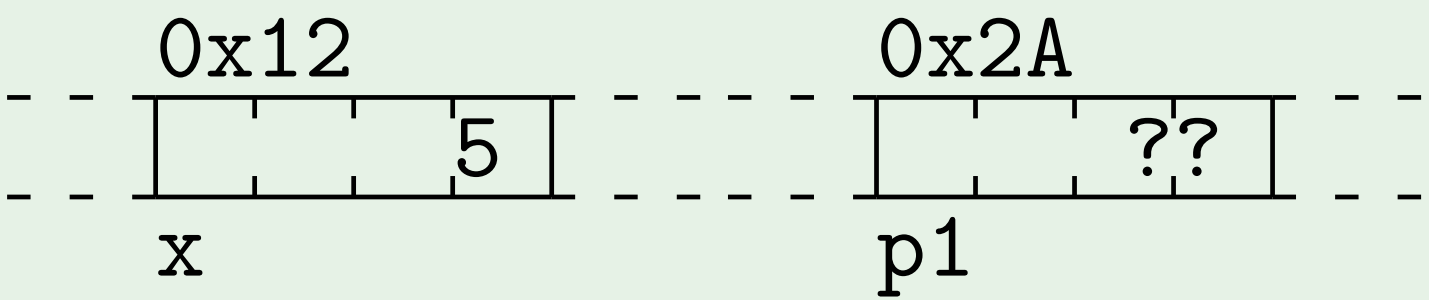
# Affectation d'une valeur à un pointeur

## Opérateur de référencement

- Opérateur permettant de passer d'une variable à son adresse (du premier octet de l'emplacement) opérateur de référencement (ou d'adressage) « & »
- On peut utiliser cette valeur pour l'assigner à un pointeur

## Exemple 1

```
   int x = 5;
-->int* p1;
   p1 = &x;/ /* p1 pointe maintenant sur l'emplacement mémoire de x *\
```



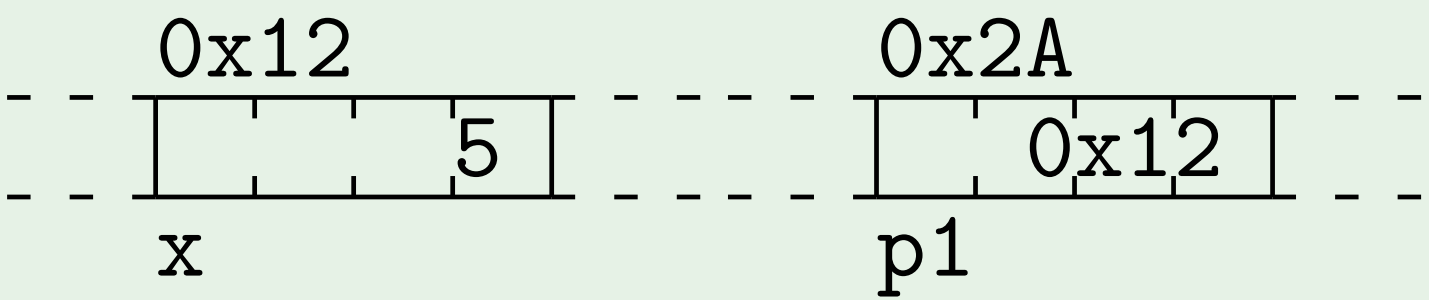
# Affectation d'une valeur à un pointeur

## Opérateur de référencement

- Opérateur permettant de passer d'une variable à son adresse (du premier octet de l'emplacement) opérateur de référencement (ou d'adressage) « & »
- On peut utiliser cette valeur pour l'affecter à un pointeur

## Exemple 1

```
    int x = 5;
    int* p1;
-->p1 = &x; /* p1 pointe maintenant sur l'emplacement mémoire de x *\
```



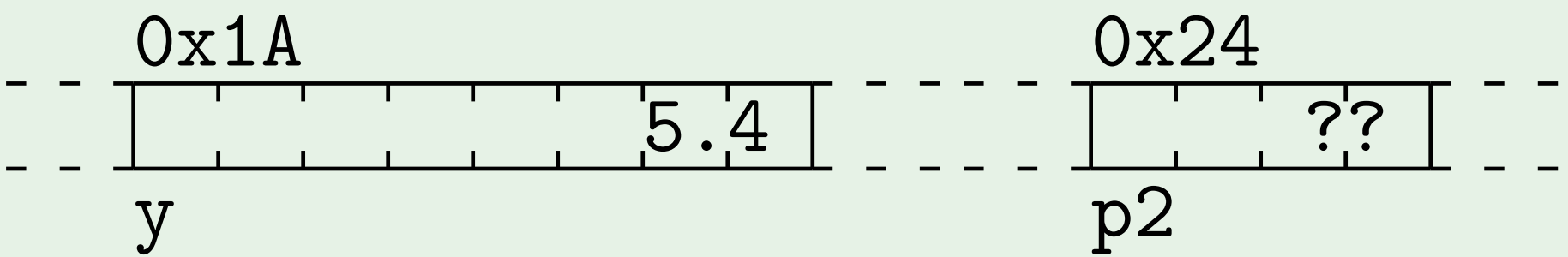
# Manipulation d'adresses mémoires par pointeurs

## Affectation d'une valeur à un pointeur

- Opérateur permettant de passer d'une variable à son adresse (du premier octet de l'emplacement) opérateur de référencement (ou d'adressage) « & »
- On peut utiliser cette valeur pour l'assigner à un pointeur

## Exemple 2

```
   double y = 5.4;
-->double* p2;
   p2 = &y; /* p2 pointe maintenant sur l'emplacement mémoire de y *\
```



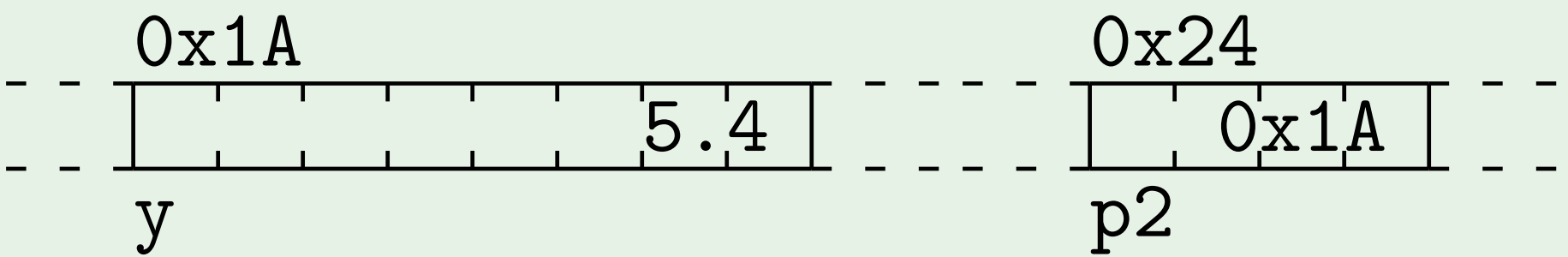
# Manipulation d'adresses mémoires par pointeurs

## Affectation d'une valeur à un pointeur

- Opérateur permettant de passer d'une variable à son adresse (du premier octet de l'emplacement) opérateur de référencement (ou d'adressage) « & »
- On peut utiliser cette valeur pour l'assigner à un pointeur

## Exemple 2

```
    double y = 5.4;
    double* p2;
-->p2 = &y; /* p2 pointe maintenant sur l'emplacement mémoire de y *\
```



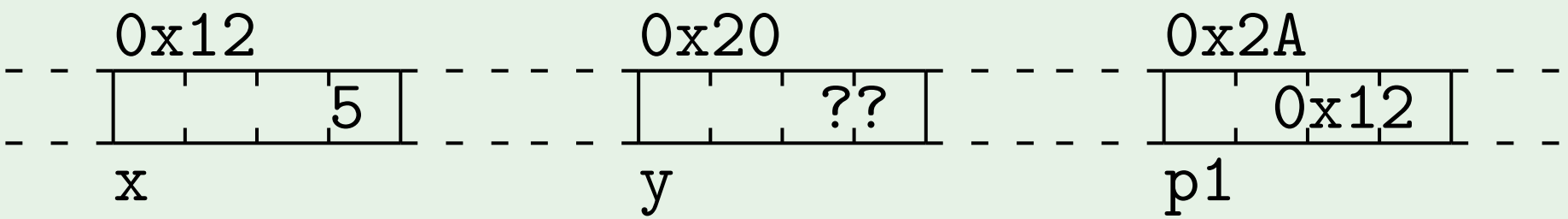
# Accès à la valeur contenue dans un emplacement pointé

## Opérateur de déréférencement

- On a besoin d'un opérateur permettant de passer d'une adresse mémoire à la valeur contenue à cette adresse.
- Opérateur de déréférencement (ou d'indirection) : « * »
- On peut l'utiliser pour lire la valeur d'une variable pointée (accès en lecture) ou pour la modifier (accès en écriture)

## Exemple d'accès en lecture

```
   int x = 5, y;
   int* p1; /* ici * se réfère au type de p1 */
-->p1 = &x;
   /* on affecte la valeur de x à y en utilisant p1 */
   y = *p1; /* ici * opérateur de déréférencement*/
```



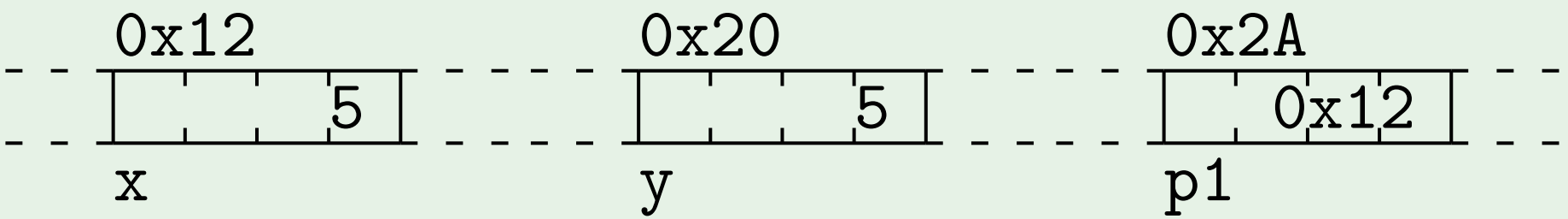
# Accès à la valeur contenue dans un emplacement pointé

## Opérateur de déréférencement

- On a besoin d'un opérateur permettant de passer d'une adresse mémoire à la valeur contenue à cette adresse.
- Opérateur de déréférencement (ou d'indirection) : « * »
- On peut l'utiliser pour lire la valeur d'une variable pointée (accès en lecture) ou pour la modifier (accès en écriture)

## Exemple d'accès en lecture

```
    int x = 5, y;
    int* p1; /* ici * se réfère au type de p1 */
    p1 = &x;
    /* on affecte la valeur de x à y en utilisant p1 */
-->y = *p1; /* ici * opérateur de déréférencement*/
```



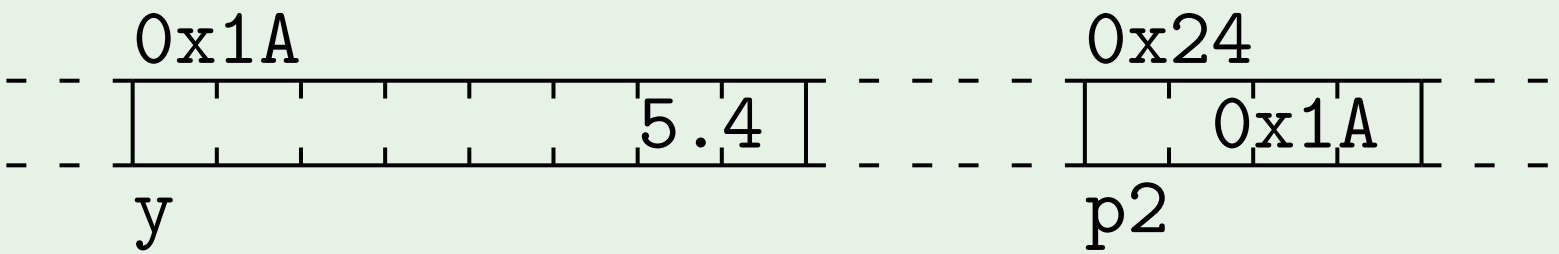
# Accès à la valeur contenue dans un emplacement pointé

## Opérateur de déréférencement

- On a besoin d'un opérateur permettant de passer d'une adresse mémoire à la valeur contenue à cette adresse.
- Opérateur de déréférencement (ou d'indirection) : « * »
- On peut l'utiliser pour lire la valeur d'une variable pointée (accès en lecture) ou pour la modifier (accès en écriture)

## Exemple d'accès en écriture

```
   double y = 5.4;
   double* p2; /* ici * se réfère au type de p2 */
-->p2 = &y;
   /* on modifie y en utilisant p2 */
   *p2 = 0.2; /* ici * opérateur de déréférencement*/
```



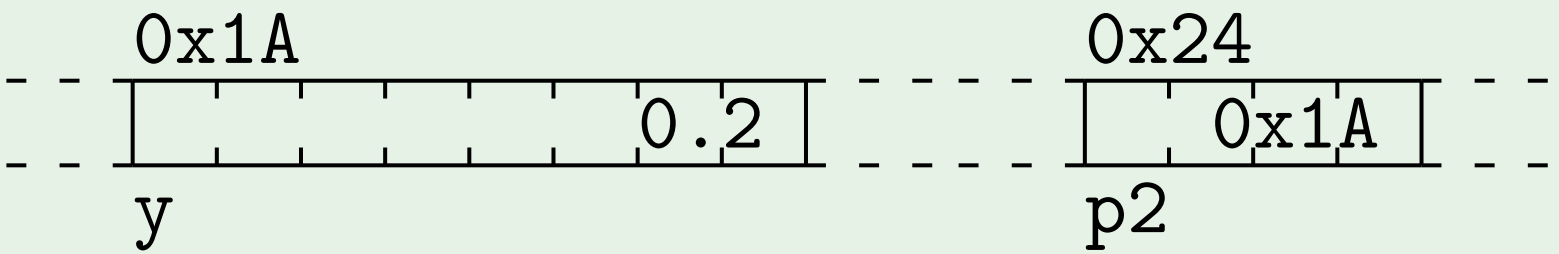
# Accès à la valeur contenue dans un emplacement pointé

## Opérateur de déréférencement

- On a besoin d'un opérateur permettant de passer d'une adresse mémoire à la valeur contenue à cette adresse.
- Opérateur de déréférencement (ou d'indirection) : « * »
- On peut l'utiliser pour lire la valeur d'une variable pointée (accès en lecture) ou pour la modifier (accès en écriture)

## Exemple d'accès en écriture

```
    double y = 5;
    double* p2; /* ici * se réfère au type de p2 */
    p2 = &y;
    /* on modifie y en utilisant p2 */
-->*p2 = 0.2; /* ici * opérateur de déréférencement*/
```

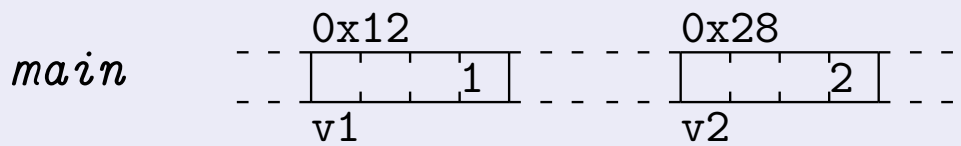


# Pointeur comme paramètre de fonction

# Passage par valeur des paramètres d'une fonction

- Rappel le passage par argument est le mécanisme lors d'un appel de fonction
- Dans le langage C, le passage d'argument se fait toujours par valeur :
  - les paramètres sont des copies indépendantes des arguments d'entrée (initialisés avec la valeur des arguments assiciés)
  - en particulier, les arguments de la fonction ne sont pas modifiés par cette fonction

```
void swap(int x1, int x2)
{
    int tmp = x1;
    x1 = x2;
    x2 = tmp;
}

int main ()
{
 -->int v1 = 1, v2 = 2;
    swap(v1,v2);
    return 0;
}
```

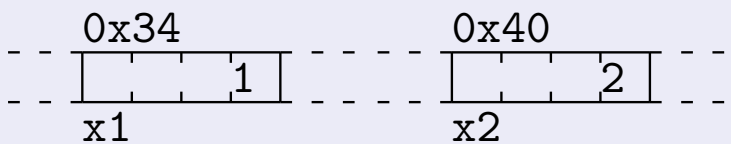


# Passage par valeur des paramètres d'une fonction

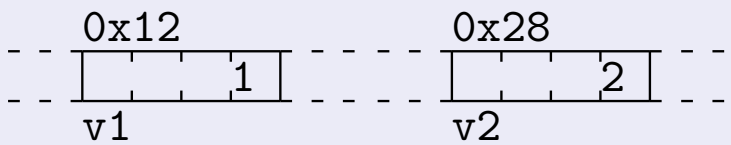
- Rappel le passage par argument est le mécanisme lors d'un appel de fonction
- Dans le langage C, le passage d'argument se fait toujours par valeur :
  - les paramètres sont des copies indépendantes des arguments d'entrée (initialisés avec la valeur des arguments assiciés)
  - en particulier, les arguments de la fonction ne sont pas modifiés par cette fonction

```
void swap(int x1, int x2)
{
    int tmp = x1;
    x1 = x2;
    x2 = tmp;
}

int main ()
{
    int v1 = 1, v2 = 2;
 -->swap(v1,v2);
    return 0;
}
```

*swap*




*main*

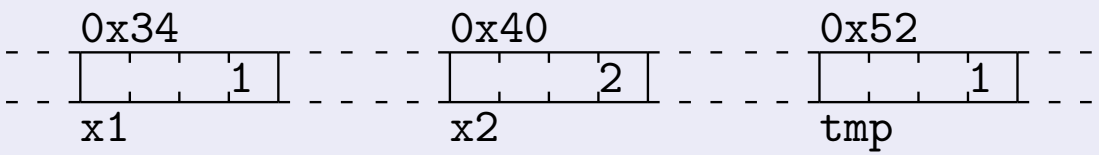


# Passage par valeur des paramètres d'une fonction

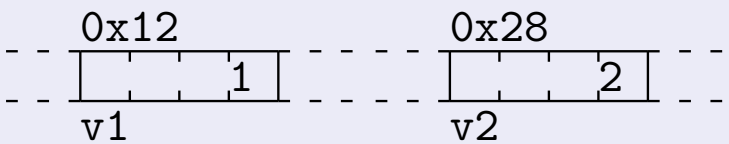
- Rappel le passage par argument est le mécanisme lors d'un appel de fonction
- Dans le langage C, le passage d'argument se fait toujours par valeur :
  - les paramètres sont des copies indépendantes des arguments d'entrée (initialisés avec la valeur des arguments assiciés)
  - en particulier, les arguments de la fonction ne sont pas modifiés par cette fonction

```
void swap(int x1, int x2)
{
 -->int tmp = x1;
    x1 = x2;
    x2 = tmp;
}

int main ()
{
    int v1 = 1, v2 = 2;
 -->swap(v1,v2);
    return 0;
}
```

*swap*




*main*

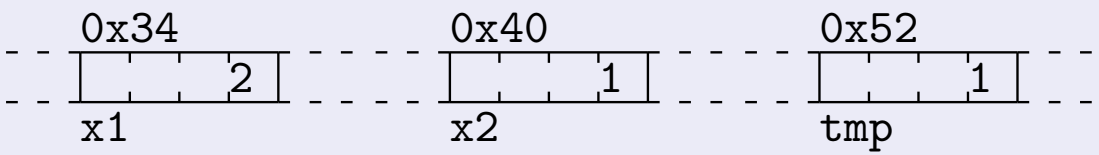


# Passage par valeur des paramètres d'une fonction

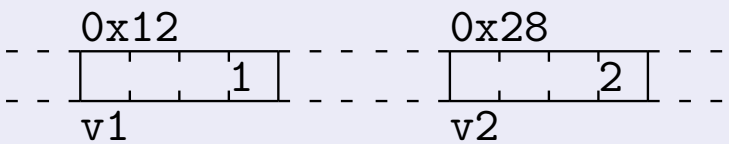
- Rappel le passage par argument est le mécanisme lors d'un appel de fonction
- Dans le langage C, le passage d'argument se fait toujours par valeur :
  - les paramètres sont des copies indépendantes des arguments d'entrée (initialisés avec la valeur des arguments assiciés)
  - en particulier, les arguments de la fonction ne sont pas modifiés par cette fonction

```
void swap(int x1, int x2)
{
    int tmp = x1;
    x1 = x2;
 -->x2 = tmp;
}

int main ()
{
    int v1 = 1, v2 = 2;
 -->swap(v1,v2);
    return 0;
}
```

*swap*



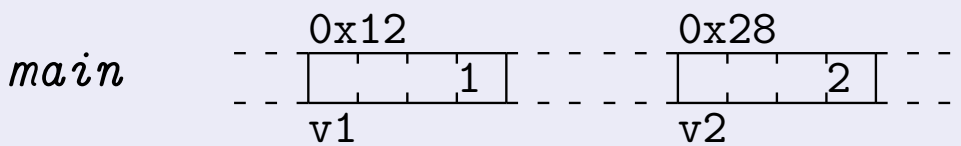

*main*



# Passage par valeur des paramètres d'une fonction

- Rappel le passage par argument est le mécanisme lors d'un appel de fonction
- Dans le langage C, le passage d'argument se fait toujours par valeur :
  - les paramètres sont des copies indépendantes des arguments d'entrée (initialisés avec la valeur des arguments assiciés)
  - en particulier, les arguments de la fonction ne sont pas modifiés par cette fonction

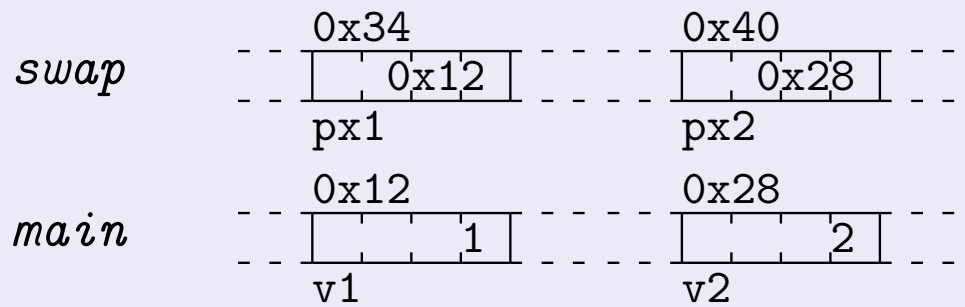
```
void swap(int x1, int x2)
{
    int tmp = x1;
    x1 = x2;
    x2 = tmp;
}

int main ()
{
    int v1 = 1, v2 = 2;
    swap(v1,v2);
 -->return 0;
}
```



# Passage par pointeur des paramètres d'une fonction

- En langage C, on peut utiliser des pointeurs comme paramètres de fonction
- Le même mécanisme a lieu:
  - les paramètres sont des copies des arguments d'entrée
  - la fonction a connaissance du ou des emplacements mémoires à modifier
  - car les pointeurs paramètres et arguments pointent sur le même emplacement mémoire
- On parle de passage par adresse ou passage par pointeur

```
void swap (int* px1, int* px2)
{-->
    int tmp = *px1;
    *px1 = *px2;
    *px2 = tmp;
}

int main ()
{
    int v1 = 1, v2 = 2;
 -->swap(&v1,&v2);
    return 0;
}
```

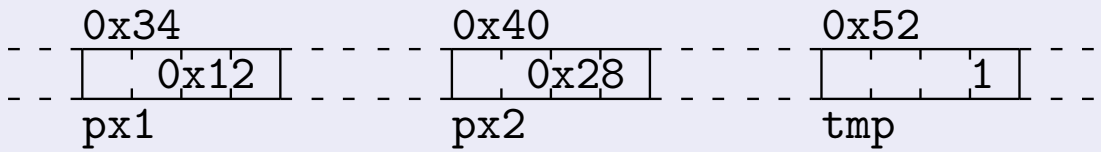


# Passage par pointeur des paramètres d'une fonction

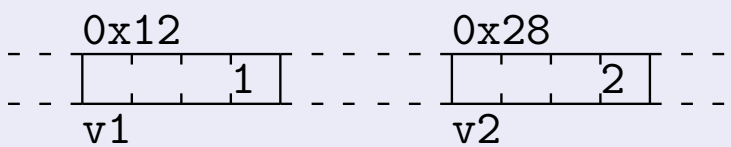
- En langage C, on peut utiliser des pointeurs comme paramètres de fonction
- Le même mécanisme a lieu:
  - les paramètres sont des copies des arguments d'entrée
  - la fonction a connaissance du ou des emplacements mémoires à modifier
  - car les pointeurs paramètres et arguments pointent sur le même emplacement mémoire
- On parle de passage par adresse ou passage par pointeur

```
void swap (int* px1, int* px2)
{
 -->int tmp = *px1;
    *px1 = *px2;
    *px2 = tmp;
}

int main ()
{
    int v1 = 1, v2 = 2;
 -->swap(&v1,&v2);
    return 0;
}
```

*swap*



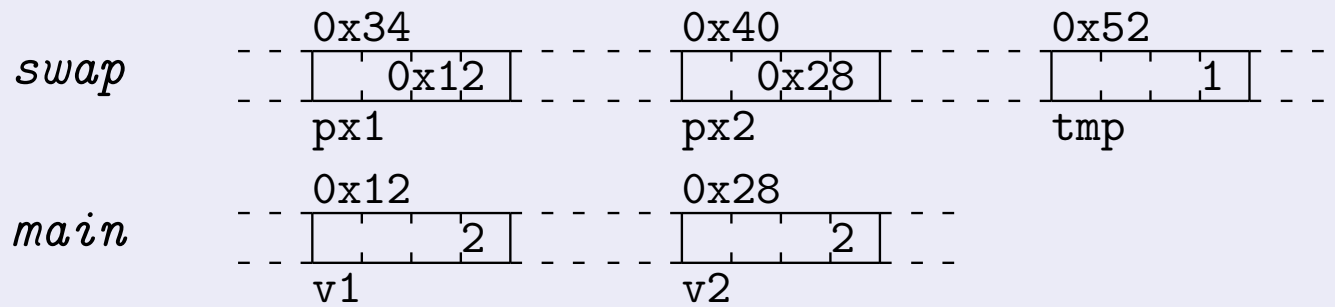

*main*



# Passage par pointeur des paramètres d'une fonction

- En langage C, on peut utiliser des pointeurs comme paramètres de fonction
- Le même mécanisme a lieu:
  - les paramètres sont des copies des arguments d'entrée
  - la fonction a connaissance du ou des emplacements mémoires à modifier
  - car les pointeurs paramètres et arguments pointent sur le même emplacement mémoire
- On parle de passage par adresse ou passage par pointeur

```
void swap (int* px1, int* px2)
{
    int tmp = *px1;
 -->*px1 = *px2;
    *px2 = tmp;
}

int main ()
{
    int v1 = 1, v2 = 2;
 -->swap(&v1,&v2);
    return 0;
}
```

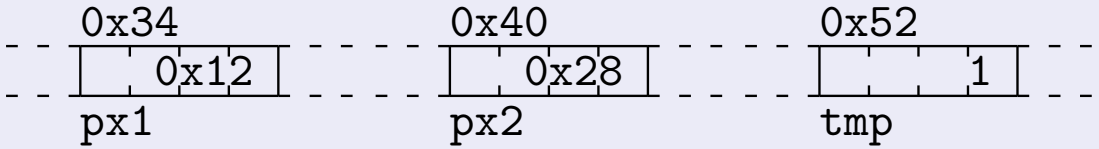


# Passage par pointeur des paramètres d'une fonction

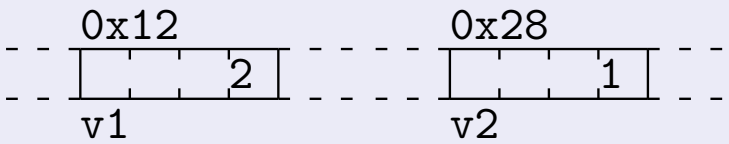
- En langage C, on peut aussi utiliser des pointeurs comme paramètre de fonction
- Le même mécanisme a lieu:
  - les paramètres sont des copies des arguments d'entrée
  - la fonction a connaissance du ou des emplacements mémoires à modifier
  - car les pointeurs paramètres et arguments pointent sur le même emplacement mémoire
- On parle de passage par adresse ou passage par pointeur

```
void swap (int* px1, int* px2)
{
    int tmp = *px1;
    *px1 = *px2;
 -->*px2 = tmp;
}

int main ()
{
    int v1 = 1, v2 = 2;
 -->swap(&v1,&v2);
    return 0;
}
```

*swap*




*main*



# Pointeur en sortie

## Exemple

- Une fonction peut retourner un pointeur en sortie
- Ceci est très commun

## Exemple

```
int* max_pos(int* arr, int n)
{
    int* pos_max = arr;
    int val_max = *pos_max;
    int i;
    for (i = 1; i < n; ++i){
        if (*(arr+i) > val_max) {
            pos_max = pos_max + i;
            val_max = *pos_max;
        }
    }
    return pos_max;
}
```

# Les entrées/sorties de la bibliothèque standard

# Les entrées/sorties de la bibliothèque standard

- Jusqu'à présent, nous utilisions une bibliothèque interne à l'INSA (`"insaio.h"`) introduite dans vous aider à vous familiariser avec le langage C.
- Le langage C possède une bibliothèque spécifique pour gérer les entrées-sorties la bibliothèque standard, le C possède un fichier d'en-tête (`stdio.h`) fournissant deux fonctions d'entrées/sorties :
- Elle fournit deux fonctions :

  `printf` pour l'affichage,
  `scanf` pour la saisie.
- Avant d'utiliser la fonction printf il faudra écrire `#include <stdio.h>` dans le code source.

# Affichage d'expressions constantes avec `printf`

- `printf` et `scanf` acceptent un nombre variable d'arguments.
- Le premier argument est appelé le format
- Le format est le seul argument obligatoire lors d'un appel à la fonction `printf`.
- Le format correspond à une expression (ici constante) que l'on souhaite afficher.

## First Examples

```
printf("Hello world!\n");
printf("Une citation: \"Le diable est dans les détails \"\n");
printf("A\tB\nC\tD\n");
```

# Affichage d'expressions numériques avec `printf`

- On peut utiliser `printf` pour afficher la valeur d'expressions (et donc de variables).
- Dans ce cas, le format doit spécifier les conversions à appliquer à chaque expression dont on souhaite afficher la valeur.
- Les expressions à afficher doivent être des arguments supplémentaire précisés lors de l'appel.
- L'ordre de ces arguments supplémentaires doit être le même que celui des spécifications de conversion correspondantes dans le format.
- Il doit y avoir autant d'arguments supplémentaires que de spécifications de conversion (sinon risque de plantage ou d'affichage incohérent).
- Au sein d'un même format, expressions constantes et spécifications de conversion peuvent cohabiter.

# Affichage d'expressions numériques avec `printf`

- Une spécification de conversion commence par le caractère.
- Elle apparaît dans le format à l'endroit ou l'expression correspondante doit être affichée.

| | |
|---|---|
| `%i` or `%d` | affiche un `int` en base 10 |
| `%f` | affiche un `float`/ `double` en base 10 |
| `%c` | affiche un `char` |
| `%s` | affiche une chaîne de caractères |
| `%p` | affiche une adresse contenue dans un pointeur en hexadécimal |

# Exemples

```c
#include <stdio.h>

int main()
{
    int a = 100;
    float x = 5.3;
    double y = -10.2;
    char sGtr[] = "toto";

    printf("Entier en base 10 : %d",a);
    printf(" ou %i\n",a);
    printf("float en base 10 : %f", x);
    printf("double en base 10 : %f \n", y);
    return 0;
}
```

# Afficher `char`

- Lors de l'affichage d'une expression de type char, la conversion spécifiée permet de choisir ce qui apparaîtra à l'écran :
  - %d or %i affichera le code de l'expression dans la table de correspondance
  - %c affichera le caractère correspondant au code dans la table.

```c
#include <stdio.h>
int main()
{
    char c = 'A';
    printf("%c a pour code %d",c, c);

    return 0;
}
```

# Attention au format !

- En mémoire, entiers et réels ne sont pas codés de la même manière.
- Ainsi, afficher une expression réelle avec un code de conversion entier (ou inversement) entraîne un affichage faux.
- On peut facilement croire que l'on a fait une erreur de calcul alors que le problème vient de la spécification de conversion.
- Toujours penser à vérifier l'affichage en premier !

```c
int a = 10;
float x = 5.3;
printf("%f\n", a); /* Format incorrect! */
printf("%d\n", x); /* Format incorrect! */
```

# Saisie d'expressions numériques

- La fonction `scanf` est utilisée pour la saisie de données.
- La syntaxe est la suivante

```
scanf(format, &variable_1, &variable_2, ...)
```

- On comprend pourquoi on doit nécessairement appeler `scanf` avec `&` (sinon aucune modification possible de l'argument d'entrée)
- La fonction scanf
  - lit des données depuis l'entrée standard (le clavier s'il n'y a pas de redirection)
  - les interprète selon le format spécifié
  - les stocke dans les variables indiquées

| | |
|---|---|
| `%d`, `%i` | saisie d'un entier `int` |
| `%f` | saisie d'un float `float` |
| `%lf` | saisie d'un float `double` |
| `%c` | saisie d'un caractère `char` |
| `%s` | saisie d'une chaîne de caractère |

# Saisie d'expressions numériques

- Le format ne doit contenir que les spécifications de conversion (pas de chaîne constante ni de séquence d'échappement).
- Il doit y avoir autant d'arguments supplémentaires que de spécifications de conversion dans le format.
- Les arguments supplémentaires doivent être fournis dans l'ordre dans lequel ils apparaissent dans le format

# Exemples

```
int a, b, c;
scanf("%i", &a);
scanf("%i%i", &b, &c);
printf("Le carre de %i est %i\n", a, a*a);
printf("La somme de %i et %i est %i\n", b, c, b+c);
```

```
#define PI 3.1415
double r, circ, area;

scanf("%lf",&r);
circ = 2*PI*r;
printf("La circonference d'un cercle de rayon %f"
"est %f\n", r, circ);
printf("L'aire d'un cercle de rayon %f est "
"%f\n",r, PI*r*r);
```

# Lire des chaînes de caractères avec scanf

- La lecture des chaînes de caractères en C est quelque chose d'assez difficile. Il est facile de commettre de erreurs et d'introduire des bugs !
- La spécification de conversion `%s` permet d'utiliser `scanf` pour lire des chaînes. Par exemple `scanf("%s", str);`
- Notez que dans l'exemple précédent nous n'avons pas écrit `&str`. L'explication vous sera donnée dès le prochain cours.

## Appel de scanf pour les chaînes de caractères

- Tous les caractères d'espacement (espace, tabulation, retour à la ligne) au début sont ignorés ;
- Ensuite, scanf lit les caractères et les stocke dans str jusqu'à ce qu'elle rencontre un caractère d'espacement.
- Enfin, elle insère le caractère fin de chaîne après le dernier caractère lu.
- `scanf` n'a aucun moyen de savoir si une chaîne de caractère est pleine. Si la longueur de la chaîne cible est insuffisante, les caractères sont stockés dans une zone mémoire non destinée à la chaîne (provoque souvent un arrêt brutal de l'exécution du programme).

# Exercice

Donnez la sortie à l'écran du programme suivant

```c
#include <stdio.h>


int main()
{
    int var1 = 5, var2 = 10;
    int* p;
    p = &var1;
    printf("var1 = %d, valeur pointée = %d\n", var1, *p);
    var1 = 20;
    printf("valeur pointée = %d\n", *p);
    *p = 2; /* ici * : opérateur de déréférencement*/
    printf("var1 = %d\n", var1);
    p = &var2;
    printf("valeur pointée = %d\n",*p);
    return 0;
}
```

# Arithmétique des pointeurs

# Arithmétique des pointeurs

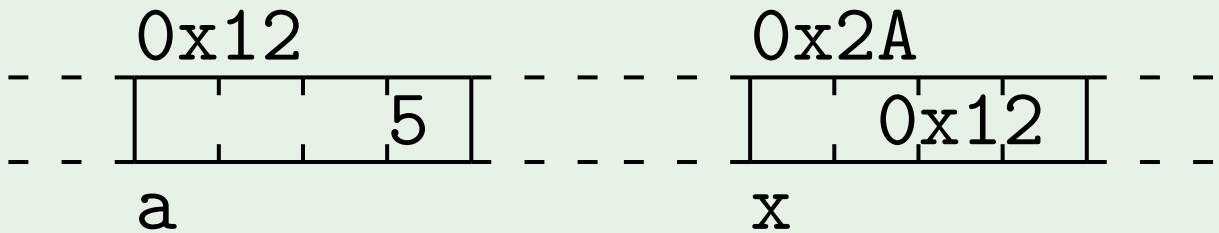
## Règle de base

Quand on ajoute une valeur entière `i` à un pointeur, il est décalé d'un nombre d'octets égal à la taille du type multipliée par `i`.

- Valeur positive : décalage vers la droite
- Valeur négative : décalage vers la gauche
- *Attention le décalage dépend donc du type du pointeur*

On suppose que les `int` occupent 4 octets. Alors, le code suivant décale le pointeur `x` de 4 octets.

```
    int a = 5;
-->int* x = &a;
    x = x + 1;
```



# Arithmétique des pointeurs

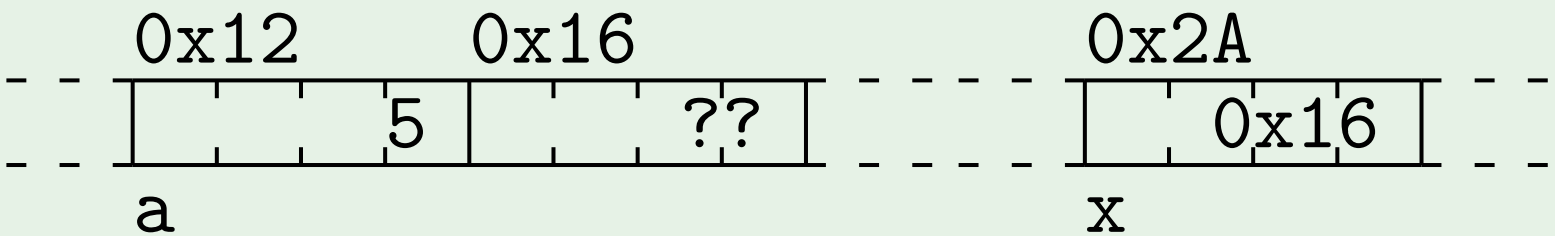
## Règle de base

Quand on ajoute une valeur entière `i` à un pointeur, il est décalé d'un nombre d'octets égal à la taille du type multipliée par `i`.

- Valeur positive : décalage vers la droite
- Valeur négative : décalage vers la gauche
- *Attention le décalage dépend donc du type du pointeur*

On suppose que les `int` occupent 4 octets. Alors, le code suivant décale le pointeur `x` de 4 octets.

```
    int a = 5;
    int* x = &a;
-->x = x + 1;
```



# Arithmétique des pointeurs

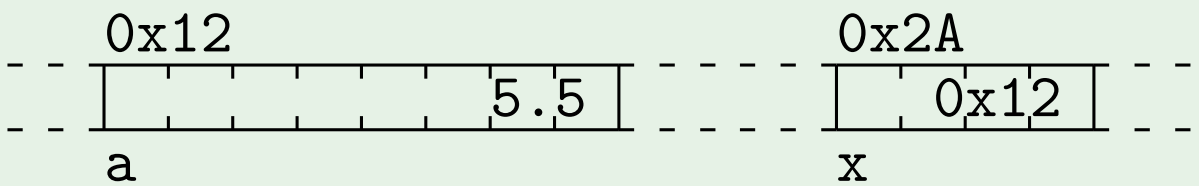
## Règle de base

Quand on ajoute une valeur entière `i` à un pointeur, il est décalé d'un nombre d'octets égal à la taille du type multipliée par `i`.

- Valeur positive : décalage vers la droite
- Valeur négative : décalage vers la gauche
- *Attention le décalage dépend donc du type du pointeur*

On suppose que les `double` occupent 8 octets. Alors, le code suivant décale le pointeur `x` de 8 octets.

```
    double a = 5.5;
-->double* x = &a;
    x = x + 1;
```



# Arithmétique des pointeurs

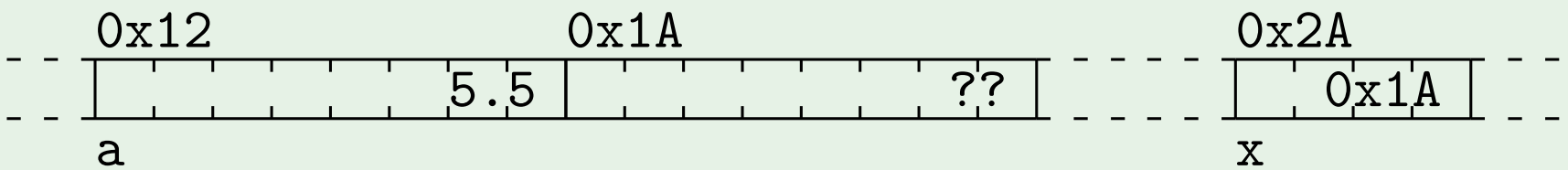
## Règle de base

Quand on ajoute une valeur entière `i` à un pointeur, il est décalé d'un nombre d'octets égal à la taille du type multipliée par `i`.

- Valeur positive : décalage vers la droite
- Valeur négative : décalage vers la gauche
- *Attention le décalage dépend donc du type du pointeur*

On suppose que les doubles occupent 8 octets. Alors, le code suivant décale le pointeur x de 8 octets.

```
    double a = 5.5;
    double* x = &a;
-->x = x + 1;
```

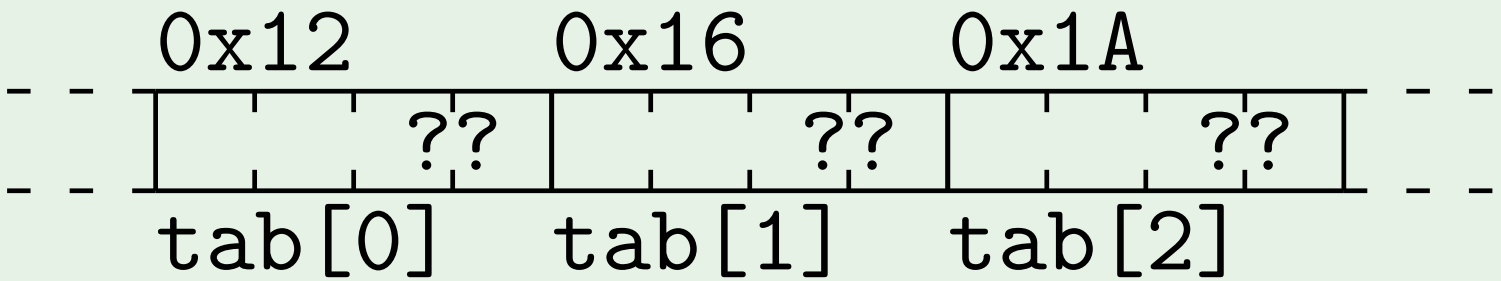


# Rappel tableaux statiques

- Un tableau correspond à un certain nombre d'octets contigus en mémoire et à une variable associée
- L'opérateur d'indexation « [] » est utilisé pour accéder aux éléments spécifiques à l'intérieur du tableau en utilisant son indice.
- `tab[i]` : le `i+1`$^{\text{ème}}$ élément du tableau `tab` (d'incide `i`)

## Exemple

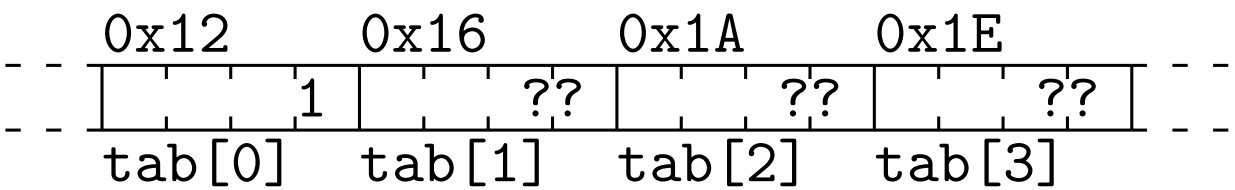
```
int tab[3];
```



# Lien entre tableaux et pointeurs

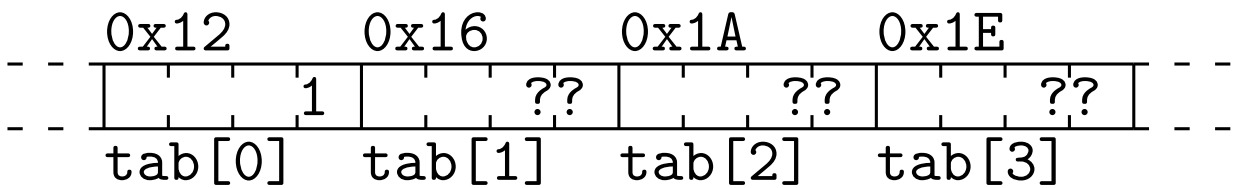
- En langage C, il y a un lien étroit entre pointeurs et tableaux.
- La variable associée à un tableau `tab` peut se manipuler comme un pointeur sur l'adresse du premier octet du tableau
  - `tab` : pointe sur le premier élément du tableau
  - `tab+i` : pointe sur l'`i+1`$^{\text{ème}}$ élément du tableau
- De manière générale, on peut écrire les deux égalités (mathématiques) suivantes :
  `tab[i] = *(tab+i)` et `tab+i = &tab[i]`

```c
int tab[4]; /* ici, tab se réfère à l'adresse 0x12 */
/* tab + 1 pointe sur tab[1] */
/* tab + 2 pointe sur tab[2] */
```



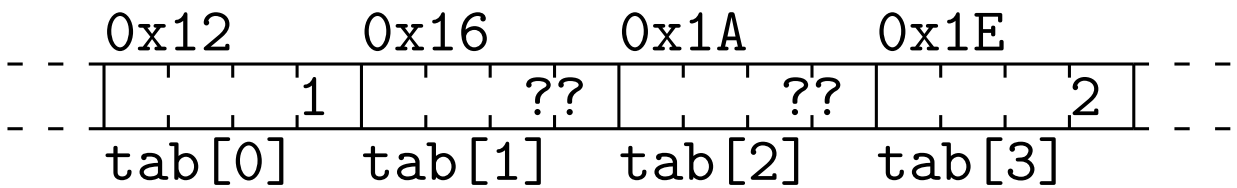
# Exemple

```
    int tab[4]; /* ici, tab se réfère à l'adresse 0x12 */
    int i=0;
    /* modifie la valeur d'int pointé par tab (0x12), tab[0]=1 */
-->*tab = 1;
    /* modifie la valeur d'int pointé par tab+3 (0x1E), tab[3]=2 */
    *(tab + 3) = 2;
    for(i=0; i<4; i++){
        /* modifie la valeur d'int pointé par tab+i, tab[i]=0 */
        *(tab + i) = 0;
    }
```



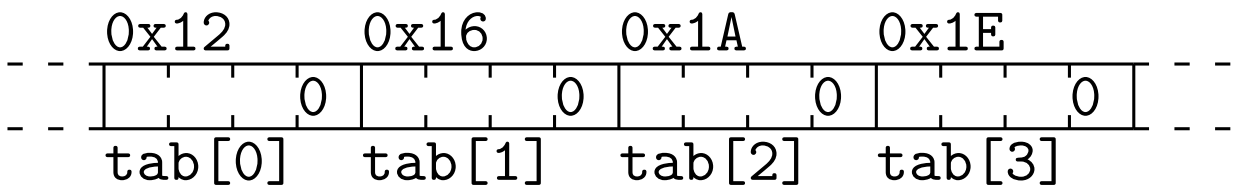
# Exemple

```
    int tab[4]; /* ici, tab se réfère à l'adresse 0x12 */
    int i=0;
    /* modifie la valeur d'int pointé par tab (0x12), tab[0]=1 */
    *tab = 1;
    /* modifie la valeur d'int pointé par tab+3 (0x1E), tab[3]=2 */
-->*(tab + 3) = 2;
    for(i=0; i<4; i++){
        /* modifie la valeur d'int pointé par tab+i, tab[i]=0 */
        *(tab + i) = 0;
    }
```



# Exemple

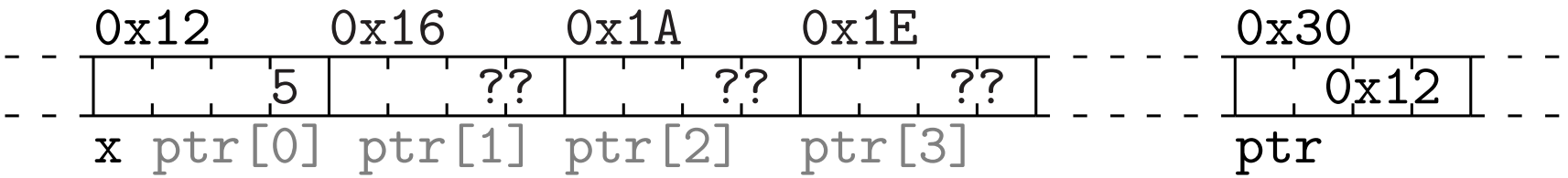
```
    int tab[4]; /* ici, tab se réfère à l'adresse 0x12 */
    int i=0;
    /* modifie la valeur d'int pointé par tab (0x12), tab[0]=1 */
    *tab = 1;
    /* modifie la valeur d'int pointé par tab+3 (0x1E), tab[3]=2 */
    *(tab + 3) = 2;
    for(i=0; i<4; i++){
        /* modifie la valeur d'int pointé par tab+i, tab[i]=0 */
        *(tab + i) = 0;
-->}
```



# Lien entre tableaux et pointeurs

- A l'inverse tout pointeur peut se manipuler comme un tableau
- L'opérateur d'indexation « `[]` » peut s'utiliser avec les pointeurs et respecte les mêmes égalités (mathématiques) :
  `ptr[i] = *(ptr+i)` et `ptr+i = &ptr[i]`
- Tout se passe comme si on manipule un tableau dont la première case possède l'adresse contenue dans le pointeur

```
int x = 5;
int* ptr = &x;
```




- Attention, rien n'assure que les emplacements mémoire dans des cases `ptr[1]`, `ptr[2]`, `ptr[3]` soient disponibles
- L'instruction `ptr[3] = 12;` est syntaxiquement correcte mais peut provoquer des erreurs.

# Exercice

Soit la déclaration suivante :

```
int tab[]= {12, 23, 34, 45, 56, 67, 78, 89, 90};
```

Quelles valeurs ou adresses fournissent ces expressions ? (on considère que tab pointe sur l'adresse 0x500)

```
tab+1
*tab+2
*(tab+2)
&tab[4]-3
tab+2
&tab[0]
tab+(*tab-10)
```

# Allocation dynamique

# Tableaux statiques

- Jusque maintenant les tableaux que nous avons manipulé étaient des tableaux dit statiques
- Ce type de tableau est stocké dans une zone mémoire spécifique : la *pile*
  - Cette zone mémoire est celle d'accès le plus rapide
  - Mais elle a une capacité restreinte
  - Lorsqu'un tableau de trop grande taille est allouée sur la pile le programme s'arrête
- La taille d'un tableau statique doit être connue à la compilation
  - Obligation d'utiliser un tableau de grande taille lors de la déclaration lorsque l'on ne connait pas la taille exacte (espace inutilisé en mémoire)

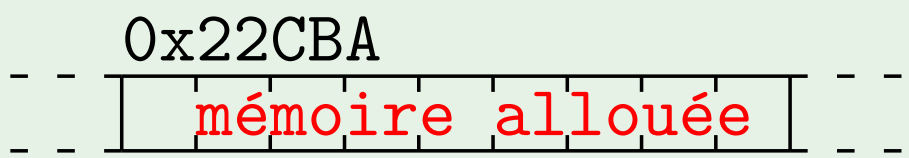
```c
int main()
{
    int tab[50];
    int taille = 0;
    printf("Saisir la taille du tableau: ");
    scanf("%i", &taille); /* doit être <50*/
    ...
}
```

# Allocation dynamique : la fonction malloc

## Fonction malloc

- L'allocation dynamique utilise la fonction `malloc` définie dans `stdlib.h`
- La mémoire allouée va dans une zone mémoire différente : le *tas*.
  - Moins rapide d'accès
  - De capacité plus grande que la pile
- `malloc` alloue un certain nombre d'octets (passé en paramètre) contigus en mémoire et retourne l'adresse du début de cette zone.

```
malloc(9);
```




- Cet appel alloue 9 octets en mémoire
- Cette expression vaut l'adresse du premier octet réservé (`0x22CBA` sur la représentation de droite)

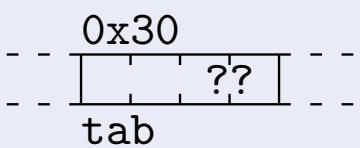
# Tableaux dynamiques

Pour créer un tableau alloué dynamiquement :

1. On alloue une zone mémoire suffisante pour un nombre `n` d'élements
2. On récupère l'adresse du premier octet

- Taille de la zone à allouer : `n` fois le nombre d'éléments
- L'opérateur `sizeof` renvoie la taille en octet d'un objet ou d'un type :
  - `sizeof(int)` vaut (généralement) 4
  - `sizeof(char)` vaut 1
- L'expression `malloc(n*sizeof(type));` :
  - Alloue la taille correpondant à `n` éléments du type voulu,
  - L'adresse de son premier octet se manipule comme un tableau de taille `n`

```
-->int* tab;
   tab = malloc(3*sizeof(int));
```



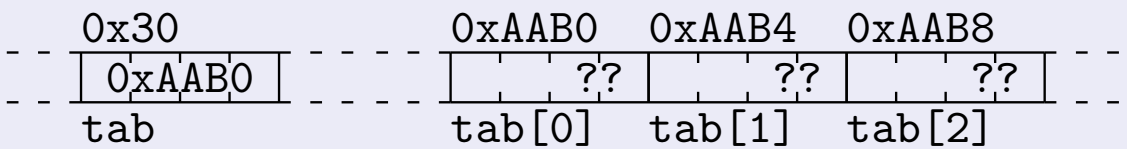
# Tableaux dynamiques

Pour créer un tableau alloué dynamiquement :

1. On alloue une zone mémoire suffisante pour un nombre `n` d'élements
2. On récupère l'adresse du premier octet

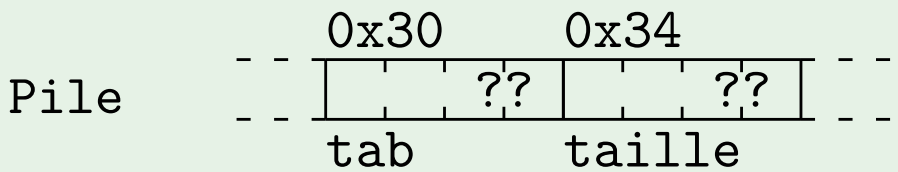
- Taille de la zone à allouer : `n` fois le nombre d'éléments
- L'opérateur `sizeof` renvoie la taille en octet d'un objet ou d'un type :
  - `sizeof(int)` vaut (généralement) 4
  - `sizeof(char)` vaut 1
- L'expression `malloc(n*sizeof(type));` :
  - Alloue la taille correpondant à `n` éléments du type voulu,
  - L'adresse de son premier octet se manipule comme un tableau de taille `n`

```
-->int* tab;
   tab = malloc(3*sizeof(int));
```



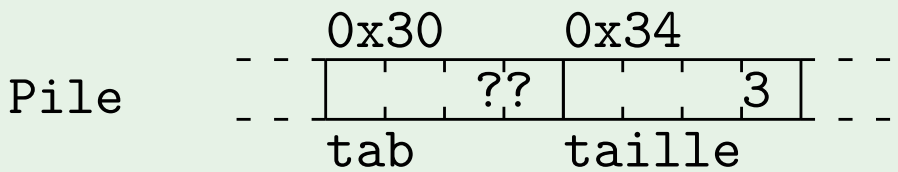
# Exemple

```c
#include <stdlib.h>
#include <stdio.h>
int main()
{
    double* tab;
 -->int taille;
    printf("Saisir la taille du tableau: ");
    scanf("%i ",&taille);
    tab = malloc(taille*sizeof(double));
    tab[0] = 2.5;
    tab[1] = 4.6;
    return 0;
}
```



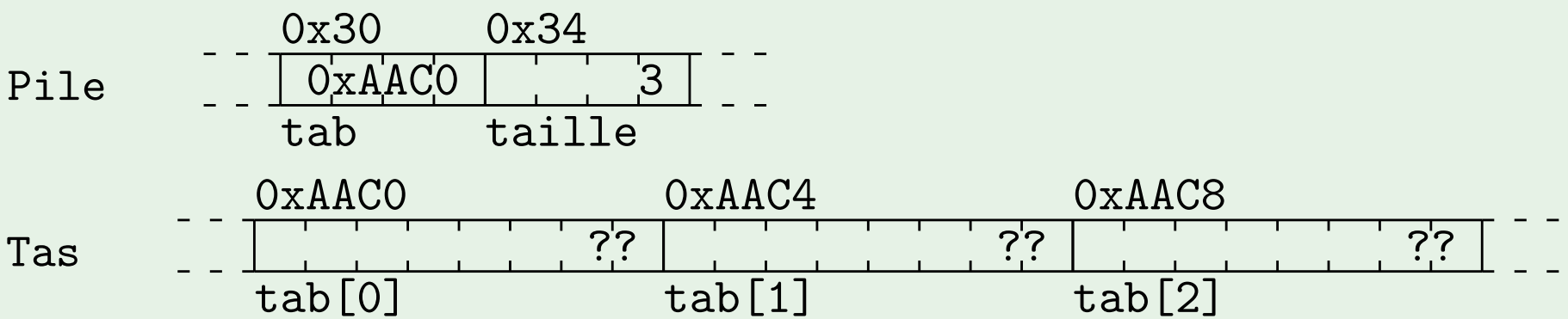
# Exemple

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
    double* tab;
    int taille;
    printf("Saisir la taille du tableau: ");
 -->scanf("%i ",&taille);
    tab = malloc(taille*sizeof(double));
    tab[0] = 2.5;
    tab[1] = 4.6;
    return 0;
}
```



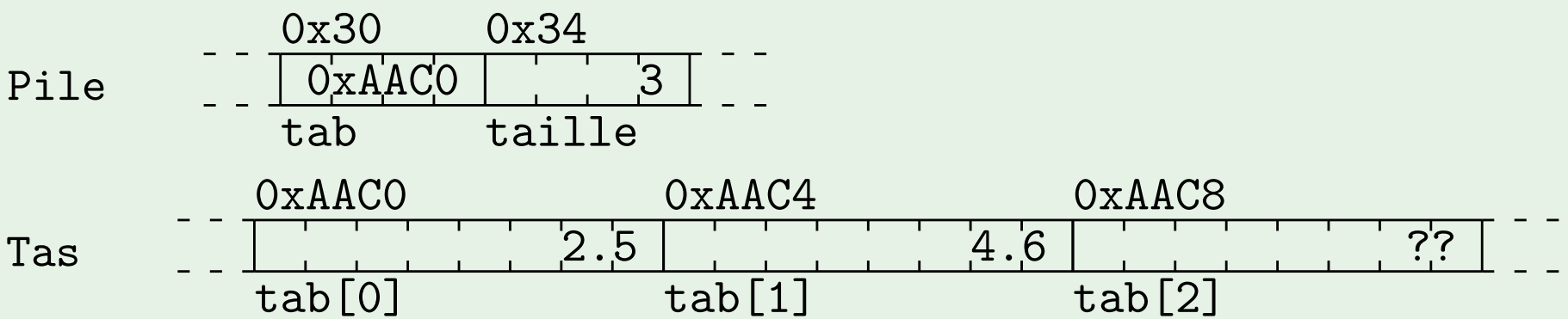
# Exemple

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
    double* tab;
    int taille;
    printf("Saisir la taille du tableau: ");
    scanf("%i ",&taille);
 -->tab = malloc(taille*sizeof(double));
    tab[0] = 2.5;
    tab[1] = 4.6;
    return 0;
}
```



# Exemple

```c
#include <stdlib.h>
#include <stdio.h>
int main()
{
    double* tab;
    int taille;
    printf("Saisir la taille du tableau: ");
    scanf("%i ",&taille);
    tab = malloc(taille*sizeof(double));
    tab[0] = 2.5;
 -->tab[1] = 4.6;
    return 0;
}
```



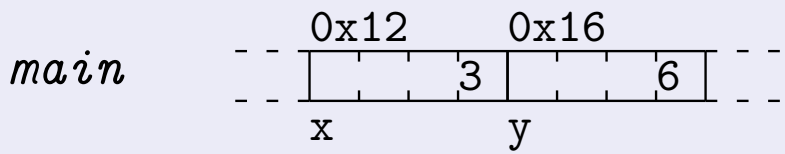
# Persistance de la mémoire

## Différence entre les tableaux statiques et dynamiques

# Durée de vie d'une variable

- Ces variables locales sont libérées à la sortie de leur porté (la porté d'une variable est le bloc de code dans laquelle elle est déclarée).
- A ce moment, l'emplacement mémoire associé devient libre et peut être attribué à une autre variable.
- Ceci est valide pour toutes les variables (notamment les pointeurs)

```c
#include <stdio.h>
int mul(int x, int y)
{
    int z = x*y;
    printf("z = %i", z);
    return 0;
}

int main()
{
 -->int x = 3, y = 6;
    mul(x, y);
    return 0;
}
```

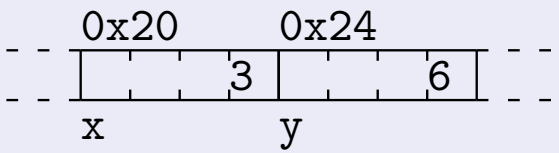


# Durée de vie d'une variable

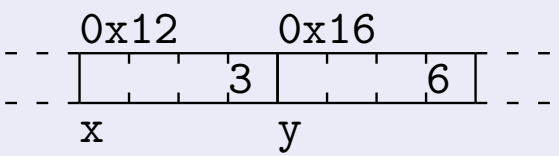
- Ces variables locales sont libérées à la sortie de leur porté (la porté d'une variable est le bloc de code dans laquelle elle est déclarée).
- A ce moment, l'emplacement mémoire associé devient libre et peut être attribué à une autre variable.
- Ceci est valide pour toutes les variables (notamment les pointeurs)

```c
#include <stdio.h>
int mul(int x, int y)
{-->
    int z = x*y;
    printf("z = %i", z);
    return 0;
}

int main()
{
    int x = 3, y = 6;
 -->mul(x, y);
    return 0;
}
```

*mul*



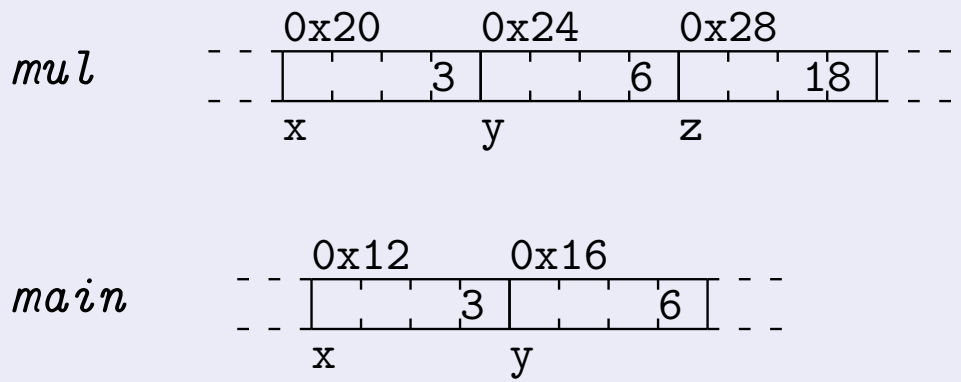

*main*



# Durée de vie d'une variable

- Ces variables locales sont libérées à la sortie de leur porté (la porté d'une variable est le bloc de code dans laquelle elle est déclarée).
- A ce moment, l'emplacement mémoire associé devient libre et peut être attribué à une autre variable.
- Ceci est valide pour toutes les variables (notamment les pointeurs)

```c
#include <stdio.h>
int mul(int x, int y)
{
 -->int z = x*y;
    printf("z = %i", z);
    return 0;
}

int main()
{
    int x = 3, y = 6;
 -->mul(x, y);
    return 0;
}
```

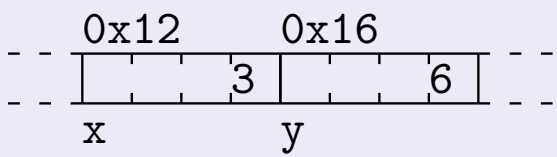


# Durée de vie d'une variable

- Ces variables locales sont libérées à la sortie de leur porté (la porté d'une variable est le bloc de code dans laquelle elle est déclarée).
- A ce moment, l'emplacement mémoire associé devient libre et peut être attribué à une autre variable.
- Ceci est valide pour toutes les variables (notamment les pointeurs)

```
#include <stdio.h>
int mul(int x, int y)
{
    int z = x*y;
    printf("z = %i", z);
    return 0;
}

int main()
{
    int x = 3, y = 6;
    mul(x, y);
 -->return 0;
}
```

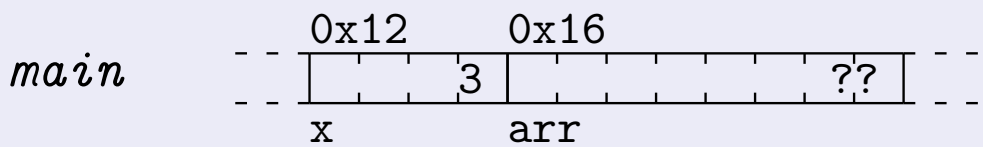
*main*



# Durée de vie des tableaux statiques

- Les tableaux statiques sont des variables locales comme les autres
- Leur durée de vie est égale à leur porté
- Lorsqu'on crée un tableau statique dans une fonction, la mémoire alloué pour celui-ci est libéré à la sortie de la fonction

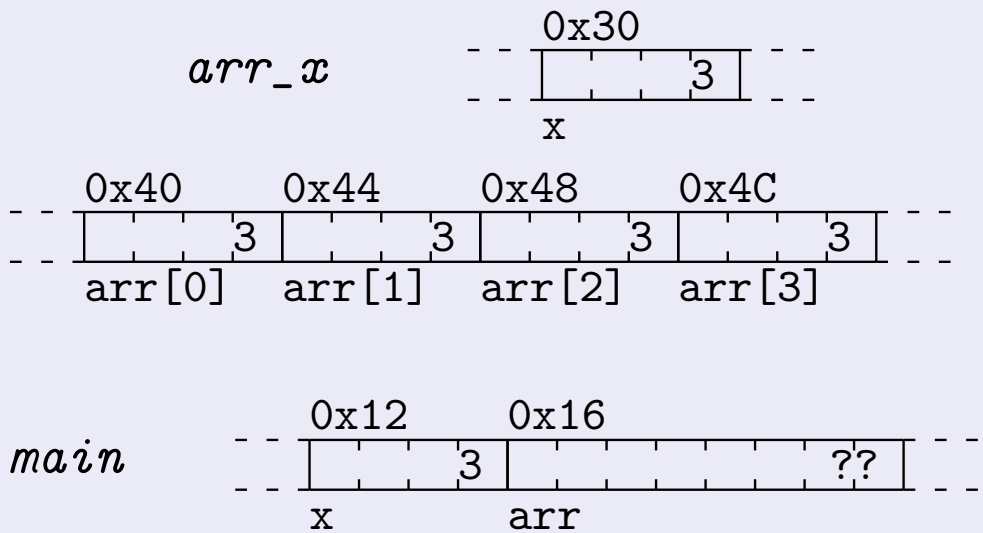
```
#include <stdio.h>
int* arr_x(int x)
{
    int arr[] = {x, x, x, x};
    printf("%i", arr[0]);
    return arr;
}

int main()
{
    int x = 3;
 -->int* arr;
    arr = arr_x(x);
    return 0;
}
```



# Durée de vie des tableaux statiques

- Les tableaux statiques sont des variables locales comme les autres
- Leur durée de vie est égal à leur porté
- Lorsqu'on crée un tableau statique dans une fonction, la mémoire alloué pour celui-ci est libéré à la sortie de la fonction

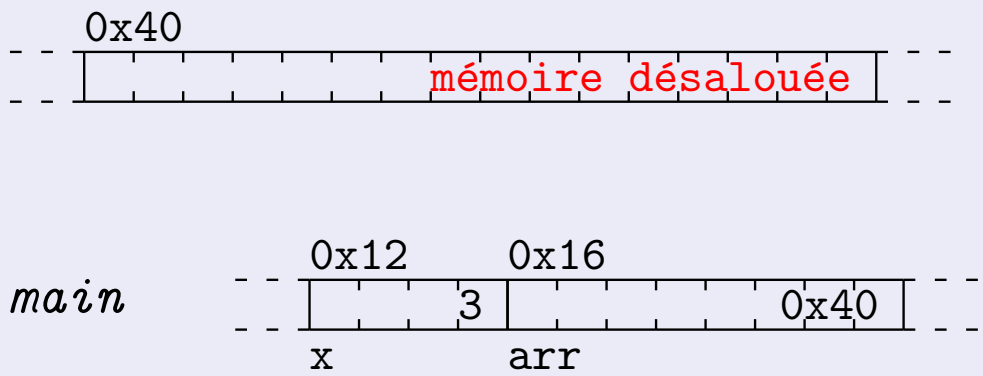
```
#include <stdio.h>
int* arr_x(int x)
{
 -->int arr[] = {x, x, x, x};
    printf("%i", arr[0]);
    return arr;
}

int main()
{
    int x = 3;
    int* arr;
 -->arr = arr_x(x);
    return 0;
}
```



# Durée de vie des tableaux statiques

- Les tableaux statiques sont des variables locales comme les autres
- Leur durée de vie est égal à leur porté
- Lorsqu'on crée un tableau statique dans une fonction, la mémoire alloué pour celui-ci est libéré à la sortie de la fonction

```c
#include <stdio.h>
int* arr_x(int x)
{
    int arr[] = {x, x, x, x};
    printf("%i", arr[0]);
    return arr;
}

int main()
{
    int x = 3;
    int* arr;
    arr = arr_x(x);
 -->return 0;
}
```

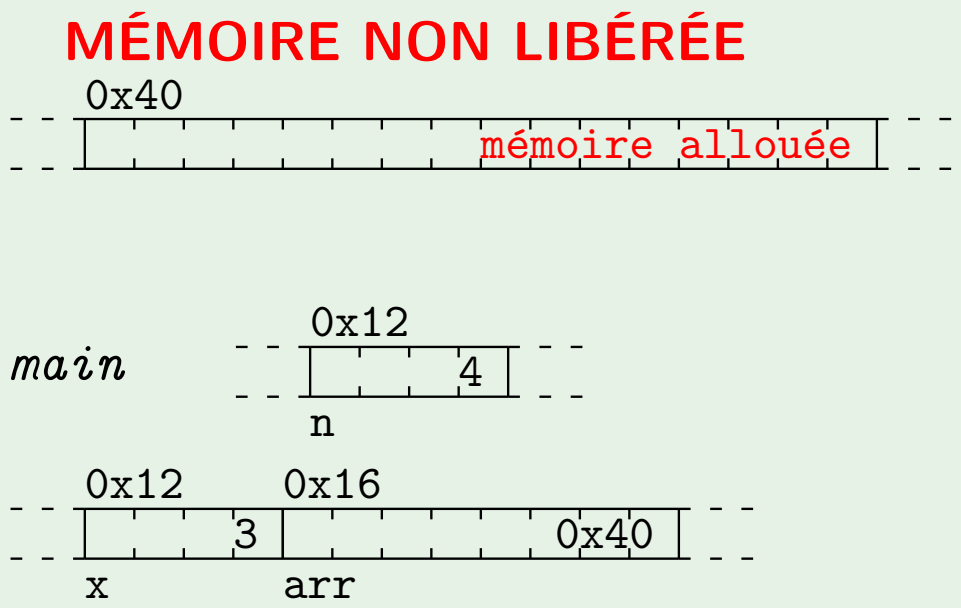


# Persistance de la mémoire dynamique

- Une zone mémoire allouée avec `malloc` n'a pas la même durée de vie : elle n'est pas libérée à la sortie du bloc où elle a été allouée.

## Tableau dynamique alloué dans une fonction

```
#include <stdlib.h>
int* allocate(int n)
{
    int* res = malloc(n*sizeof(int));
    return res;
}

int main()
{
    int n = 4;
    int* arr;
    arr = allocate(n);
 -->...
    return 0;
}
```

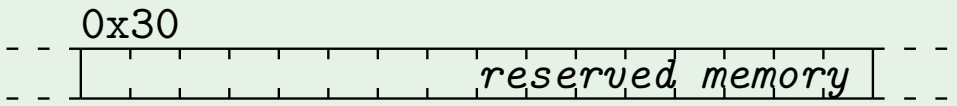


# Persistance de la mémoire dynamique

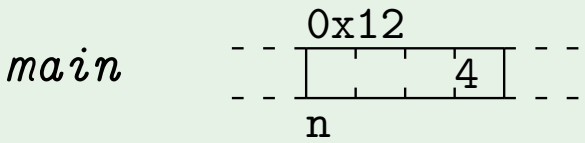
- Le pointeur qui contient l'adresse du tableau dynamique sera libéré à la sortie de la fonction
- Celui-ci doit être renvoyé en sortie de fonction pour pouvoir accéder à la mémoire alloué.

## Tableau dynamique alloué dans une fonction

```c
#include <stdio.h>
#include <stdlib.h>
void allocate(int n)
{
    int* res = malloc(n*sizeof(int));
}

int main()
{
    int n = 4;
    allocate(n);
 -->...
    return 0;
}
```




**On a perdu l'information sur le pointeur : cette mémoire ne pourra plus être libérée**



# Nécessité d'une fonction `free`

- Un tableau dynamique n'est pas désallouer à la sortie du bloc où il a été défini
- Il est donc nécessaire de la désallouer manuellement lorsqu'il n'est plus utilisé
- On utilise la fonction `free` (de `stdlib.h`) qui prend en argument un pointeur sur une zone mémoire à libérer.
- Ne jamais utilisée pour des emplacements mémoires non alloué avec `malloc`.

## Exemple

```c
#include <stdio.h>
#include <stdlib.h> // Nécessaire pour malloc
int main()
{
    int* tab;
    int taille;
    printf("Donnez la taille du tableau : ");
    scanf("%d ",&taille);
    tab = malloc(taille*sizeof(int));
    //... etc
    free(tab);
    return 0;
}
```

```c
#include <stdlib.h> // Nécessaire pour malloc
#include <stdio.h>

float moyenne(float* tab, int taille)
{
    float somme = 0.;
    int i;
    for (i=0; i<taille; i++) {somme+=tab[i];}
    return somme/taille;
}

int main()
{
    float* tab;
    int taille = 3;
    float m;
    tab = malloc(taille*sizeof(float));
    tab[0] = 1.2;
    tab[1] = 1;
    tab[2] = 2.5;
    m = moyenne(tab, taille);
    printf("\n moyenne = %f\n'', m);
    free(tab);
    return 0;
}
```

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES CENTRE VAL DE LOIRE

# Algorithmique et Langage C
# Types de données avancées

Camille BAUDOIN, Moncef HIDANE
camille.baudoin@insa-cvl.fr, moncef.hidane@insa-cvl.fr

2022-2023

# Les types avancés

## Intêret

- Pour donner du sens au code (faciliter la lecture du programme)
- Pour regrouper des données hétérogènes (`int`, `char *`, )
- Pour simplifier la programmation

Il est logique de représenter ainsi une réalité complexe. C'est la première approche de l'objet.

## Types de données avancés de ce cours

- Les synonymes
- Les énumérations
- Les structures

# Synonyme

# Synonyme

- Définition d'un nouveau type
- Possibilité de déclarer un identifiant en tant qu'alias de type, à utiliser pour remplacer un nom de type éventuellement complexe
- Comme le type de base équivalent
- Donner un sens particulier à une série de variables de même nature logique
- Pour des tableaux ou matrices de taille fixe

# Synonyme

## Syntaxe

- Définition au début du programme (après les `#include`)
- Mot clé `typedef`
- Syntaxe identique à la déclaration d'une variable
- Les types ainsi déclarer s'utilise comme les types classiques : déclaration de variables, paramètres de fonction, création de nouveau type.

```
/* On définit un nouveau type "coordonnee" synonyme de int */
typedef int coordonnee;
/* On définit un nouveau type "vecteur" synonyme de tableau de
3 coordonnées */
typedef coordonnee vecteur[3];
```

# Exemple

```c
#include <stdio.h>
typedef int coordonnee;
typedef coordonnee vecteur[3];

void affiche(vecteur v)
{
    printf("%i %i %i", v[0], v[1], v[2]);
}

int main()
{
    /* équivalent à int x1 = 10, y1 = 7, z1 = 12;*/
    coordonnee x1 = 10, y1 = 7, z1 = 12;
    vecteur v1; /* equivalent à coordonee v1[3];*/
    v1[0] = x1;
    v1[1] = y1;
    v1[2] = z1;
    affiche(v1);
    return 0;
}
```

# Énumération

# Énumération

## Utilité

- Pour un type pouvant prendre un nombre fini de valeurs
- Elles sont manipulées par le compilateur comme des entiers
- Introduire du sens (donc de la lisibilité) dans le programme

## Syntaxe

```
typedef enum{v1, v2,..., vn} nomType;
```

- Une valeur entière est associée à chaque élément de l'énumération
- Par défaut, la première valeur (v1) vaut 0, et les suivantes sont incrémentées (v2 vaut 1, ... vn vaut n-1)
- Possibilité de spécifier la première valeur (les suivantes seront incrémentées)

```
/* v1 = 4, v2 = 5, ... vn = n+3 */
typedef enum{v1 = 4, v2,..., vn} nomType;
```

- Possibilité des spécifier toutes les valeurs

```
typedef enum{v1 = 3, v2 = 10, v3 = 1, v4 = 7} nomType;
```

# Énumération : déclaration de variable

## Exemple : les jours de la semaine

```
/* LUNDI vaut 0, MARDI vaut 1, ... DIMANCHE vaut 6*/
typedef enum{LUNDI,MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI,
             DIMANCHE} jour;

/* LUNDI vaut 1, MARDI vaut 2, ... DIMANCHE vaut 7*/
typedef enum{LUNDI=1,MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI,
             DIMANCHE} jour;

/* LUNDI vaut 1, MARDI vaut 2, ... DIMANCHE vaut 7*/
typedef enum{MARDI=2, VENDREDI=5, DIMANCHE=7, MERCREDI=3, LUNDI=1,
             JEUDI=4, SAMEDI=6} jour;

/* Déclaration de variable de type jour */
jour j1, j2;
j1 = MARDI;
j2 = DIMANCHE;
```

# Enumération : comparaison

- Les opérateurs de comparaison `< <= > >= == !=` s'applique sur les énumérations
- Un boucle `for` peut être contrôlée par une variable de type énumération

```
/* LUNDI vaut 0, MARDI vaut 1, ... DIMANCHE vaut 6*/
typedef enum{LUNDI,MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI,
             DIMANCHE} jour;

/* Utilisation */
jour j1 = MARDI, j2 = DIMANCHE;

/* Condition vrai */
if (j1<j2){
    printf("le jour 1 arrive avant le jour 2 dans la semaine");
}

for(j1 = LUNDI, j1<=DIMANCHE; j1++){
    /* j1 parcours les jours de la semaine */
}
```

# Énumération : saisie et affichage

- On peut ni lire, ni afficher directement la valeur d'une variable énumérée
- Utilisation de sa valeur entière associée

## Saisie

```
void saisir(jour *j)
{
    printf("Saisie d'un jour :\n"
    printf("0-lundi\n" );
    printf("1-mardi\n" );
    printf("2-mercredi\n" );
    printf("3-jeudi\n" );
    printf("4-vendredi\n" );
    printf("5-samedi\n" );
    printf("6-dimanche\n" );
    int i;
    scanf("%i",&i);
    *j = i;
}
```

# Énumération : saisie et affichage

- On peut ni lire, ni afficher directement la valeur d'une variable énumérée
- Utilisation de sa valeur entière associée

## Affichage

Pour « `j1 = MARDI;` » l'instruction `printf("jour = %i", j1)` affichera à l'écran « `jour = 1` ». Il faut donc implémenter un affichage spécifique.

```
/* Solution 1 : affichage conditionnel */
void affiche(jour j)
{
    if(j == LUNDI){
        printf("lundi");
    }
    if(j == MARDI){
        printf("mardi");
    }
    ...
    if(j == DIMANCHE){
        printf("dimanche");
    }
```

# Énumération : saisie et affichage

- On peut ni lire, ni afficher directement la valeur d'une variable énumérée
- Utilisation de sa valeur entière associée

## Affichage

Pour « `j1 = MARDI;` » l'instruction `printf("jour = %i", j1)` affichera à l'écran « `jour = 1` ». Il faut donc implémenter un affichage spécifique.

```
/* Solution 2 : utilisation d'un tableau*/
void affiche(jour j)
{
    /* nomJour[i] pour i entier entre 0 et 6 est une chaîne de
    caractère qui contient le nom du jour à afficher */
    char nomJour[7][9] = {"lundi", "mardi", "mercredi", "jeudi",
    "vendredi", "samedi", "dimanche"};

    printf("jour = %s", nomJour[j]);
}
```

# Structure

# Les structures

## Les types déjà abordés

- Pour des données isolées : les types simples
- Pour regrouper des ensembles finis d'éléments de même type : les tableaux

## Intérêt des structures

- Permet de grouper des variables de types différents ayant un lien au sein d'une même entité.
- Permet de représenter des objets ou concepts de la vie réelle

## Champs d'une structure

- Chaque élément de la structure, appelé *champ*, est référencé par un identificateur, appelé *nom de champ*.
- Deux champs ne peuvent pas avoir le même nom
- Un champ ne peut pas être du type que la structure [a]
- Les champs d'une structure sont consécutifs en mémoire

a. mais un pointeur vers une structure de même type est possible

# Définition d'un type structuré

**Syntaxe**

```
struct nomType
{
    type1 nomChamp1;
    type2 nomChamp2;
...
};
```

**Exemple**

```
struct velo
{
    int nbVitesses;
    float hauteur;
    char marque[250];
};
```

- Ici on a défini un nouveau type : `struct velo`.
- Toute variable de type `struct velo` contiendra automatiquement une valeur pour chacun de ses champs (`nbVitesses`, `hauteur`, `marque`).

**Attention**

La définition d'un type structuré ne crée pas de variable en mémoire. Seule la déclaration d'une variable de ce type allouera de la mémoire.

# Déclaration d'une variable structurée

## Syntaxe

- Déclaration de variable

```
struct nomType nomVariable;
```

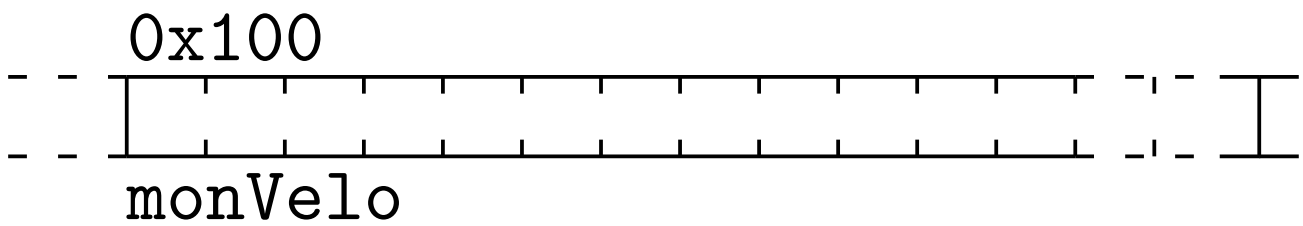
- Opérateur d'accés au champ, « . »: permet accéder aux champs d'une variable structurée

```
maVariable.monChamp1 = 9;
```

- La priorité de l'opérateur d'accès au champ est très élevée, il est donc inutile d'utiliser des parenthèses.

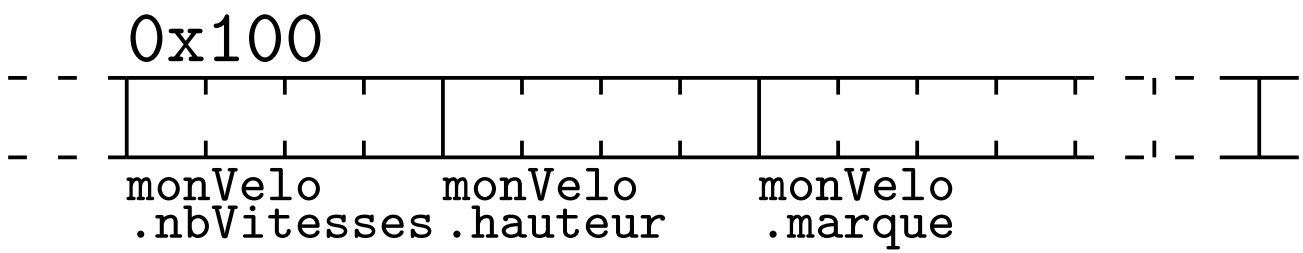
# Exemple

```c
struct velo{
    int nbVitesses;
    float hauteur;
    char marque[250];
};

int main()
{
    /* monVelo est une variable de type struct velo et possède
    des valeurs pour les champs nbVitesses, hauteur, et marque */
 -->struct velo monVelo;
    maVelo.nbVitesses = 9;
    maVelo.hauteur = 1.2;
    strcpy(maVelo.marque, "Fahrradmanufaktur");
    return 0;
}
```



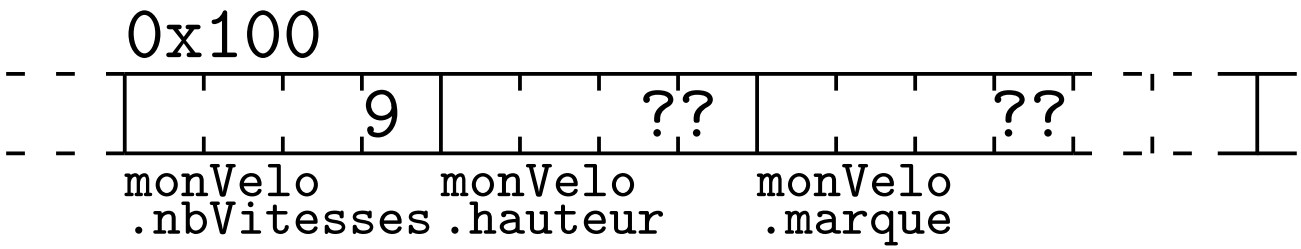
# Exemple

```c
struct velo{
    int nbVitesses;
    float hauteur;
    char marque[250];
};

int main()
{
    /* monVelo est une variable de type struct velo et possède
    des valeurs pour les champs nbVitesses, hauteur, et marque */
 -->struct velo monVelo;
    maVelo.nbVitesses = 9;
    maVelo.hauteur = 1.2;
    strcpy(maVelo.marque, "Fahrradmanufaktur");
    return 0;
}
```



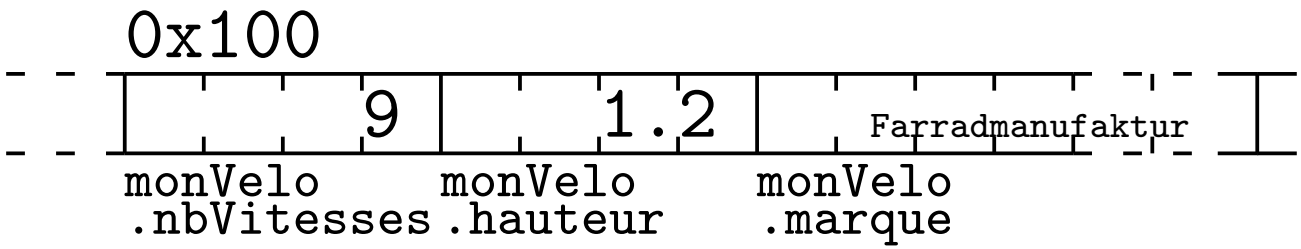
# Exemple

```c
struct velo{
    int nbVitesses;
    float hauteur;
    char marque[250];
};

int main()
{
    /* monVelo est une variable de type struct velo et possède
    des valeurs pour les champs nbVitesses, hauteur, et marque */
    struct velo monVelo;
 -->maVelo.nbVitesses = 9;
    maVelo.hauteur = 1.2;
    strcpy(maVelo.marque, "Fahrradmanufaktur");
    return 0;
}
```



# Exemple

```c
struct velo{
    int nbVitesses;
    float hauteur;
    char marque[250];
};

int main()
{
    /* monVelo est une variable de type struct velo et possède
    des valeurs pour les champs nbVitesses, hauteur, et marque */
    struct velo monVelo;
    maVelo.nbVitesses = 9;
    maVelo.hauteur = 1.2;
 -->strcpy(maVelo.marque, "Fahrradmanufaktur");
    return 0;
}
```



# Synonyme pour les structures

- Pour éviter l'utilisation du mot clé `struct` à chaque déclaration, on déclare un synonyme une fois le type structuré défini

```
struct nomStructure {
    type1 nomChamp1;
    type2 nomChamp2;
};
typedef struct nomStucture nomType;
```

- Remplace toute occurence de `nomType` par `struct nomStructure`

```
struct s_velo {
    ...
};
typedef struct s_velo velo;

int main()
{
    velo monVelo; /* déclaration "classique" : type identifiant;*/
    ...
}
```

# Déclaration condensée : structure et synonyme

```
typedef struct {
    type1 nomChamp1;
    type2 nomChamp2;
} nomType;

/* Au lieu de */
struct nomStruc {
    type1 nomChamp1;
    type2 nomChamp2;
};
typedef struct nomStruc nomType;
```

- Format très utilisé
- Attention : il n'est pas possible lorsqu'un des champs de la structure est un pointeur vers une variable du même type structuré

# Exemple

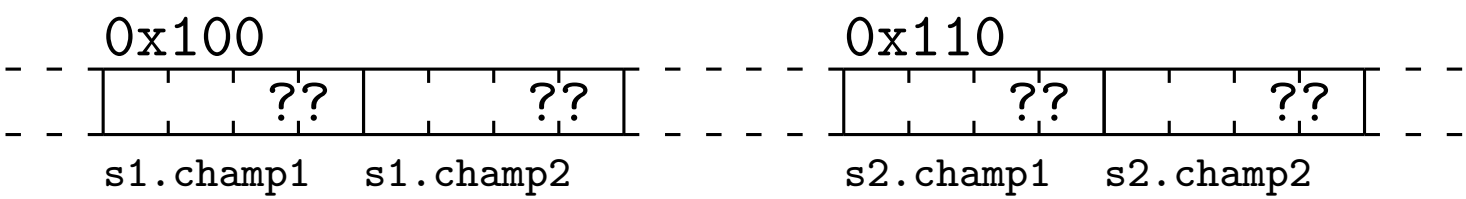
- Remarque : deux types structurés différents peuvent avoir un même nom de champ. La désignation avec l'opérateur d'accès permet de les différencier.

```
typedef struct {
    int nbVitesses;
    float hauteur;
    char marque[250];
} velo;
typedef struct{
    int hauteur;
    char marque[250];
} trotinette;

int main()
{
    velo monVelo;
    trotinette maTrot;
    monVelo.hauteur = 1.2; /* hauteur de monVelo (float) */
    maTrot.hauteur = 4; /* hauteur de maTrot (int) */
    return 0;
}
```

# Copie de structure

- L'opérateur d'affectation « = » entre deux structures réalise une copie champ à champ.

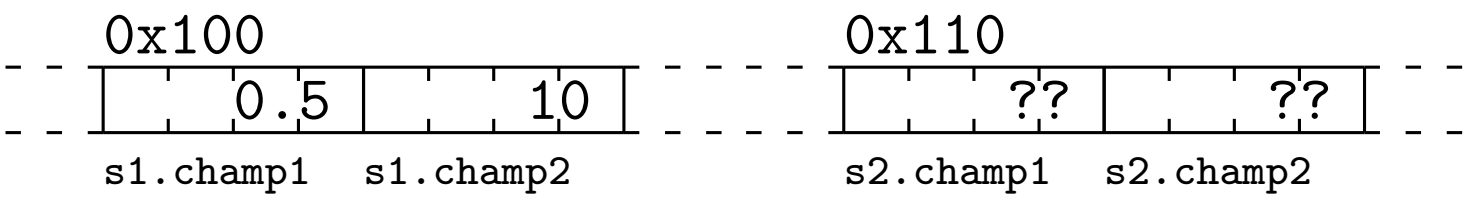
```
typedef struct{
    float champ1;
    int champ2;
}nomStructure;

int main()
{
 -->nomStructure s1, s2;
    s1.champ1 = 0.5;
    s1.champ2 = 10;
    /* eq. à s2.champ1 = s1.champ1; et s2.champ2 = s1.champ2; */
    s2 = s1;
    return 0;
}
```



# Copie de structure

- L'opérateur d'affectation « = » entre deux structures réalise une copie champ à champ.

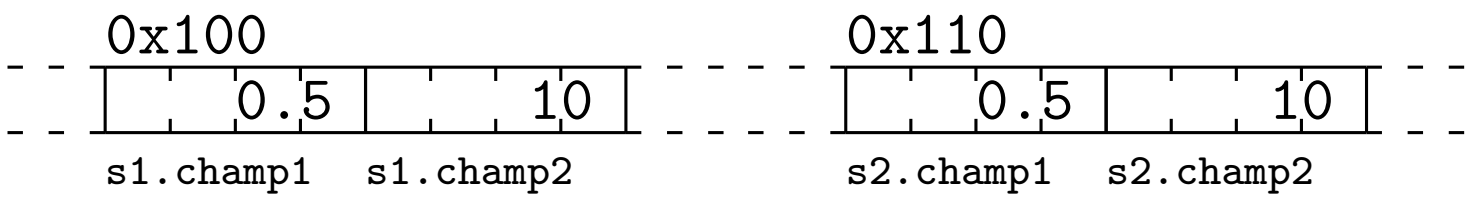
```
typedef struct{
    float champ1;
    int champ2;
}nomStructure;

int main() {
    nomStructure s1, s2;
    s1.champ1 = 0.5;
 -->s1.champ2 = 10;
    /* eq. à s2.champ1 = s1.champ1; et s2.champ2 = s1.champ2; */
    s2 = s1;
    return 0;
}
```



# Copie de structure

- L'opérateur d'affectation « = » entre deux structures réalise une copie champ à champ.

```
typedef struct{
    float champ1;
    int champ2;
}nomStructure;

int main() {
    nomStructure s1, s2;
    s1.champ1 = 0.5;
    s1.champ2 = 10;
    /* eq. à s2.champ1 = s1.champ1; et s2.champ2 = s1.champ2; */
 -->s2 = s1;
    return 0;
}
```



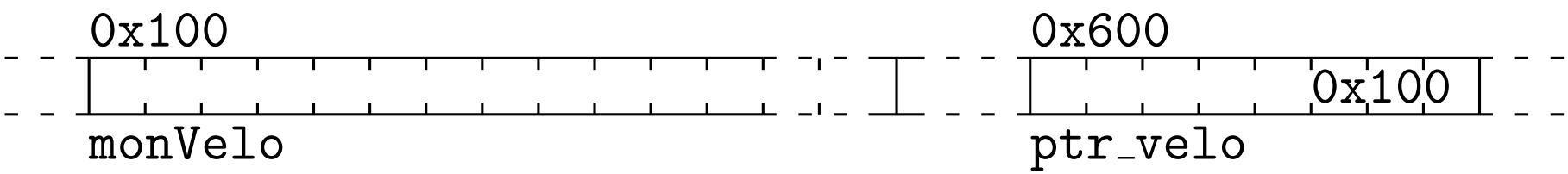
# Structure et fonction

```c
#include <stdio.h>
/* défintion du type velo */

velo saisir_velo(){ /* valeur de retour de type velo */
    velo v;
    printf("Saisir la hauteur, le nombre de vitesses et la marque du vélo");
    scanf("%f%i%s", &v.hauteur, &v.nbVitesses, v.marque);
    return v;
}

void afficher_velo(velo v){ /* paramètre d'entrée de type velo */
    printf("Vélo de marque %s, de hauteur %f et possède %i vitesses",
    v.marque, v.hauteur, v.nbVitesses);
}

int main()
{
    velo monVelo = saisir_velo();
    afficher_velo(monVelo);
    return 0;
}
```

# Pointeur et structure

- Comme pour tout autre type on peut déclarer et manipuler un pointeur sur une structure :

```c
typedef struct velo {
    int nbVitesses;
    float hauteur;
    char marque[250];
} velo;

int main()
{
    velo monVelo;
    velo* ptr_velo = &monVelo;
    return 0;
}
```



# Pointeur et structure

- Utiliser notamment pour modifier une structure au sein d'un fonction en passant une structure en paramètre par adresse

```c
typedef struct {
    int nbVitesses;
    float hauteur;
    char marque[250];
} velo;

void modifier_nbVitesse(velo* v1, int nb) {
    /* *v1 permet d'écrire directement sur l'emplacement monVelo de
    la fonction main */
    (*v1).nbVitesses = nb;
}

int main() {
    velo monVelo;
    /* Modifie le nbVitesses de mon vélo, égal à 6 après l'appel */
    modifier_nbVitesse(&monVelo, 6);
    return 0;
}
```

# Pointeur et structure

- Lorsqu'on dispose d'un pointeur sur une variable structurée, l'opérateur "->" permet d'accéder directement à un champ à partir d'une structure.
- Beaucoup utilisé dans les fonctions où l'on passe la structure par adresse

```c
typedef struct {
    int nbVitesses;
    float hauteur;
    char marque[250];
} velo;

void modifier_nbVitesse(velo* v1, int nb) {
    /* Modifie le champs nbVitesse de la variable monVelo du main */
    v1-> nbVitesses = nb; /* equivalent à (*v1).nbVitesses = nb; */
}

int main() {
    velo monVelo;
    /* Modifie le nbVitesses de mon vélo, égal à 6 après la fonction */
    modifier_nbVitesse(&monVelo, 6);
    return 0;
}
```

# Passage par pointeur constant de structure

## Mot clé const

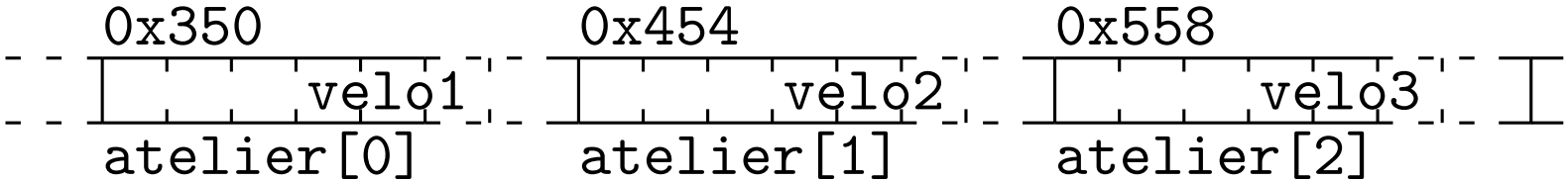
- Une fonction prend une structure en paramètre d'entre, il y a un recopie de la variable structurée
- Les variables structurées peuvent être lourde en mémoire
- Pour éviter des recopies trop lourdes, on peut faire un passage par pointeur (création uniquement d'un pointeur de 4 ou 8 octets).
- Utilisation du mot-clé `const` pour spécifier que le contenu pointé ne peut être modifier par la fonction

```
void fonction(const velo* v1, int nb) {
    v1-> nbVitesses = nb; /* erreur de compilation */
}
```

# Tableau de structure

- On peut déclarer un tableau de structures de la même façon qu'un tableau d'un type de base :
  - Tableau statique : `velo atelier[50];`
  - Tableau dynamique : `velo* atelier = malloc(50*sizeof(velo));`

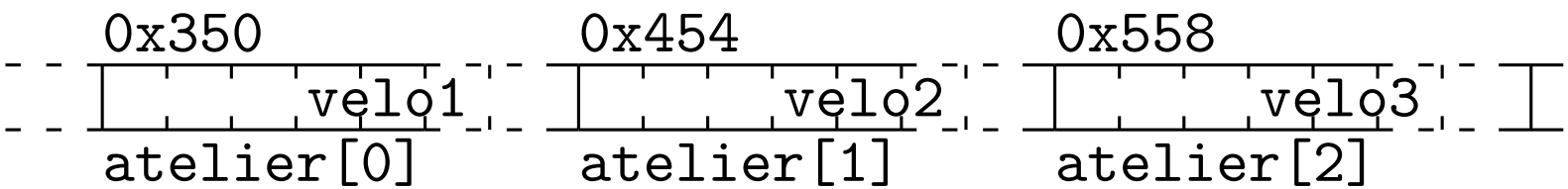
- `atelier` se comporte comme (dans le cas des tableaux dynamiques est) un pointeur sur `velo` (ici, contient l'adresse `0x350`)



# Tableau de structure

- On manipuler un tableau d'un type structure comme dans n'importe quel type de tableau
  - Arithmétique des pointeurs : pour un pointeur `p` sur `velo`, `p + 1` contient l'adresse de `p` décalée positivement de `sizeof(velo)` octets
  - `atelier + i` est un pointeur sur le `i+1`$^{\text{ème}}$ `velo` du tableau
  - `atelier[i]` est le `i+1`$^{\text{ème}}$ élément du tableau
- Par exemple, on peut effectuer le parcours suivant :

```
int i;
/* Parcours du tableau de vélo */
for(i = 0; i < 50; i++)
{
    printf("vélo %i : modèle : %s\n", i, atelier[i].modele);
}
```

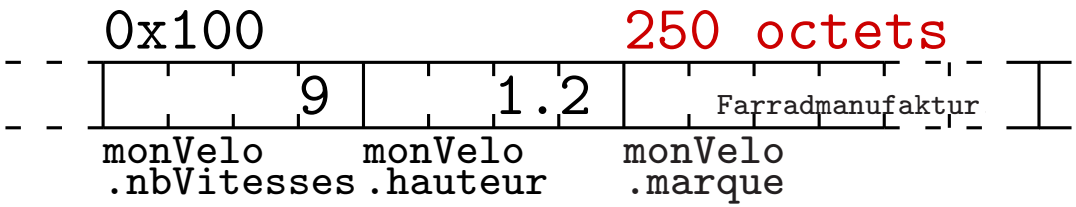


# Différence entre contenu statique et contenu dynamique

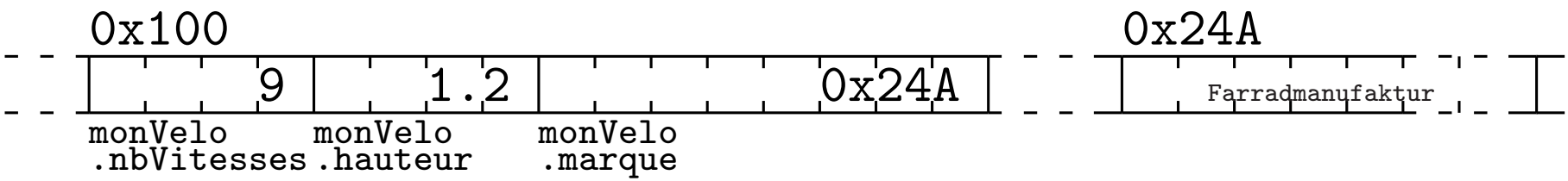
```c
typedef struct {
    int nbVitesses;
    float hauteur;
    char marque[250];
} VeloV1;
```

```c
typedef struct {
    int nbVitesses;
    float hauteur;
    char* marque;
} VeloV2;
```

- veloV1 : chaîne de caractères marque contenue dans un tableau statique
  - tous les octets du tableau statique sont contenus dans la structure
  - `sizeof(veloV1)` est égal à 260




- veloV2 : chaîne de caractères marque contenue dans un tableau dynamique
  - contient uniquement le pointeur vers la chaîne de caractères
    `sizeof(veloV2)` vaut 16



# Différence entre contenu statique et contenu dynamique

## Initialisation

- Fonction d'initialisation garantissant que les chaînes sont valides (et vides)

```
void initVeloV1(VeloV1 *pv){
    pv->marque[0] = '\0';
    pv->nbVitesses = 0;
    pv->hauteur = 0;
}
```

```
void initVeloV2(VeloV2 *pv) {
    pv->marque = malloc(sizeof(char)*250);
    pv->marque[0] = '\0';
    pv->nbVitesses = 0;
    pv->hauteur = 0;
}
```

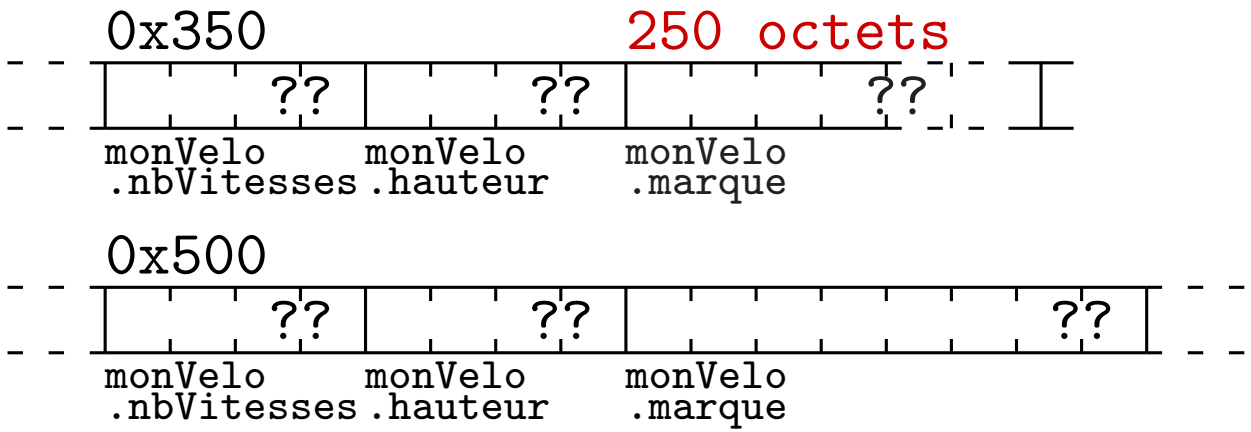
## Libération

- Pour la version 2, il y a une allocation dynamique. On doit définir une fonction de libération devant être appelée sur chaque vélo lorsque celui-ci n'est plus utilisé.

```
void libereVeloV2(VeloV2 *pv){
    free(pv->marque);
    pv->marque = NULL;
}
```

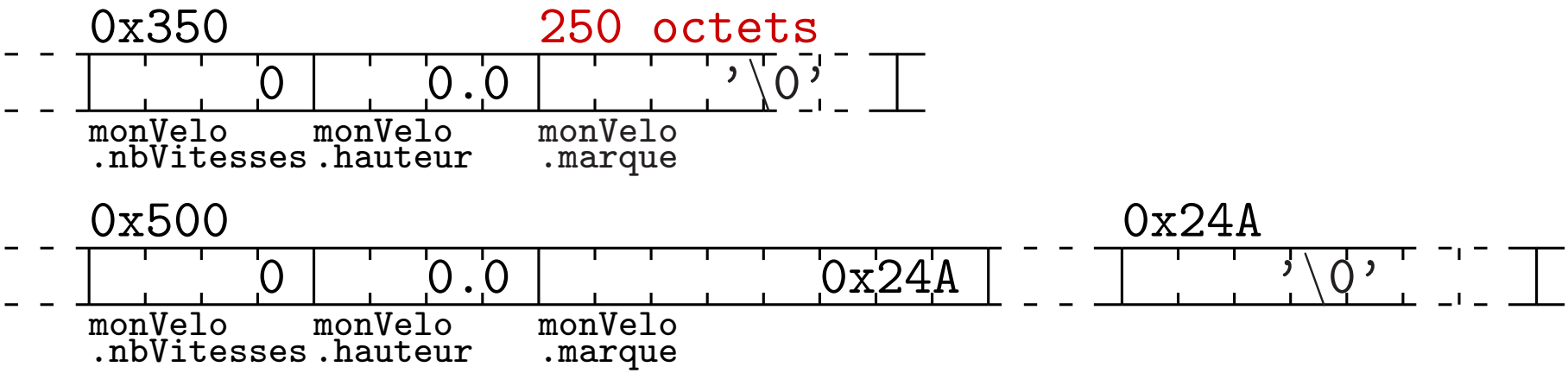
# Différence entre contenu statique et contenu dynamique

```
#include <stdlib.h>
/* Définition des structures et des fonctions d'initialisation */
int main()
{
    VeloV1 monVelo;
 -->VeloV2 autreVelo;
    initVeloV1(&monVelo);
    initVeloV2(&autreVelo);
    /* ... */
    libereVeloV2(&autreVelo);
    return 0;
}
```



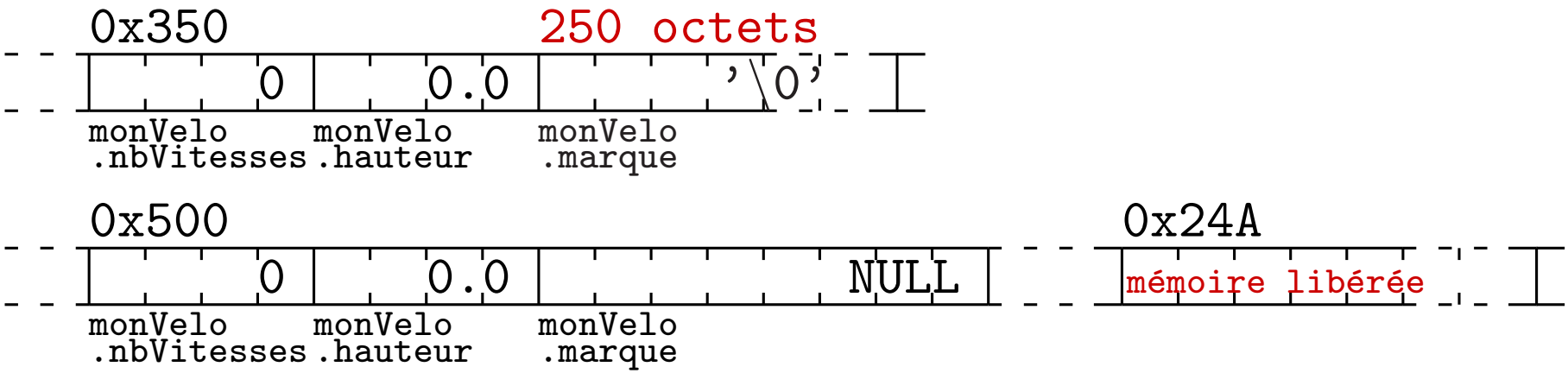
# Différence entre contenu statique et contenu dynamique

```
#include <stdlib.h>
/* Définition des structures et des fonctions d'initialisation */
int main()
{
    VeloV1 monVelo;
    VeloV2 autreVelo;
    initVeloV1(&monVelo);
 -->initVeloV2(&autreVelo);
    /* ... */
    libereVeloV2(&autreVelo);
    return 0;
}
```



# Différence entre contenu statique et contenu dynamique

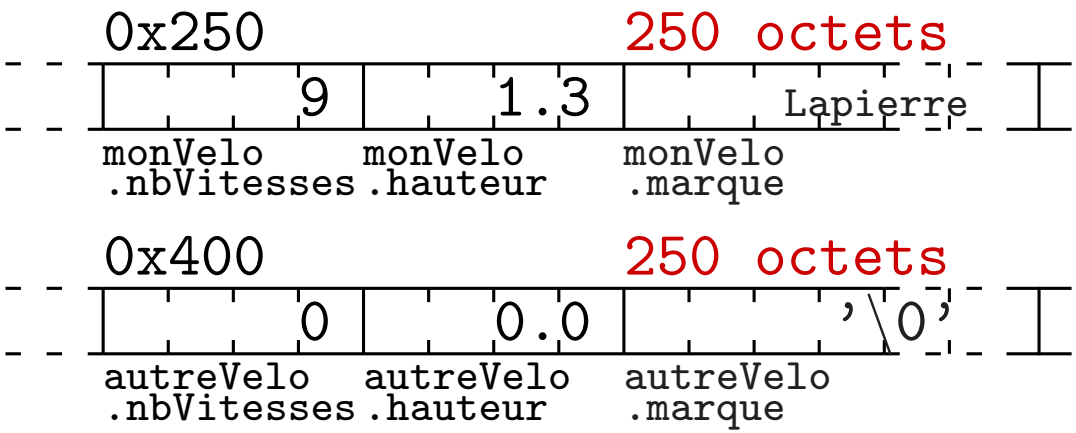
```c
#include <stdlib.h>
/* Définition des structures et des fonctions d'initialisation */
int main()
{
    VeloV1 monVelo;
    VeloV2 autreVelo;
    initVeloV1(&monVelo);
    initVeloV2(&autreVelo);
    /* ... */
 -->libereVeloV2(&autreVelo);
    return 0;
}
```



# Copie de structure contenant un tableau statique

- L'opérateur d'affectation « = » réalise une copie champ à champ y compris du contenu du tableau statique

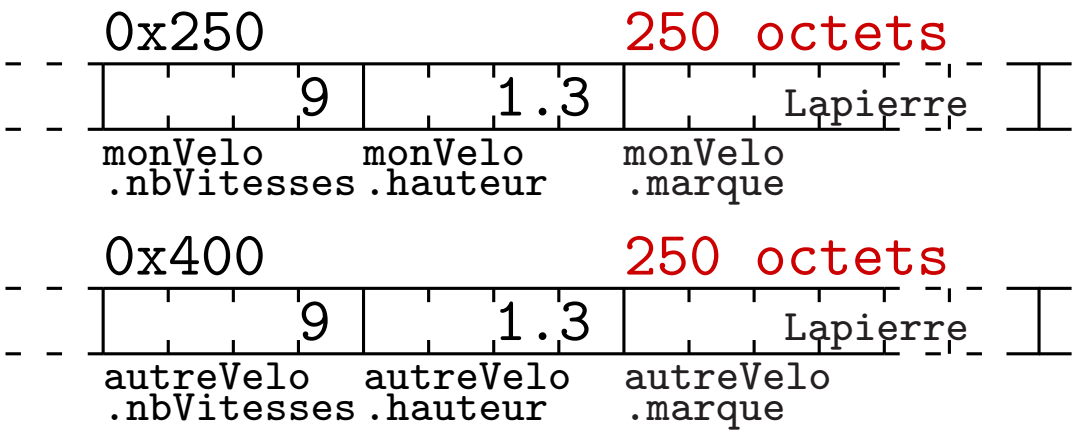
```
VeloV1 monVelo, autreVelo;
initVeloV1(&monVelo);
initVeloV1(&autreVelo);
monVelo.nbVitesses = 9;
monVelo.hauteur = 1.3;
-->strcpy(monVelo.marque, "Lapierre");
autreVelo = monVelo; /* copie valide */
```



# Copie de structure contenant un tableau statique

- L'opérateur d'affectation « = » réalise une copie champ à champ y compris du contenu du tableau statique

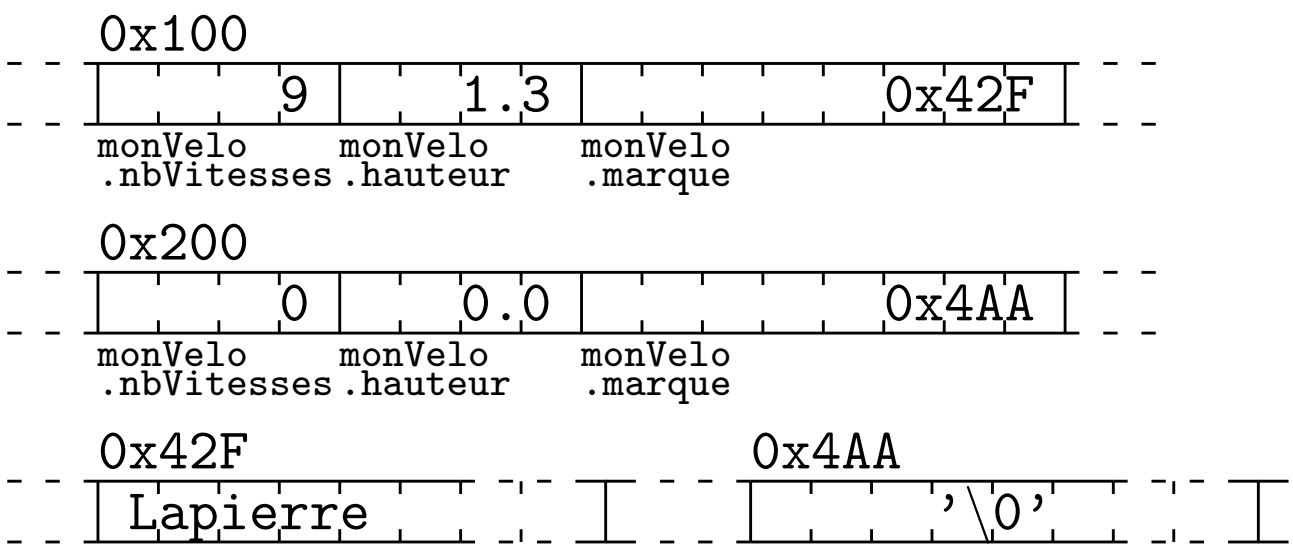
```
VeloV1 monVelo, autreVelo;
initVeloV1(&monVelo);
initVeloV1(&autreVelo);
monVelo.nbVitesses = 9;
monVelo.hauteur = 1.3;
strcpy(monVelo.marque, "Lapierre");
-->autreVelo = monVelo; /* copie valide */
```



# Copie de structure contenant un tableau dynamique

- La copie via l'opérateur d'affectation d'une structure contenant un tableau génère une situation malsaine en mémoire :
  - plus aucune variable locale ne contient l'adresse 0x42F
  - monVelo et autreVelo pointent vers la même zone mémoire

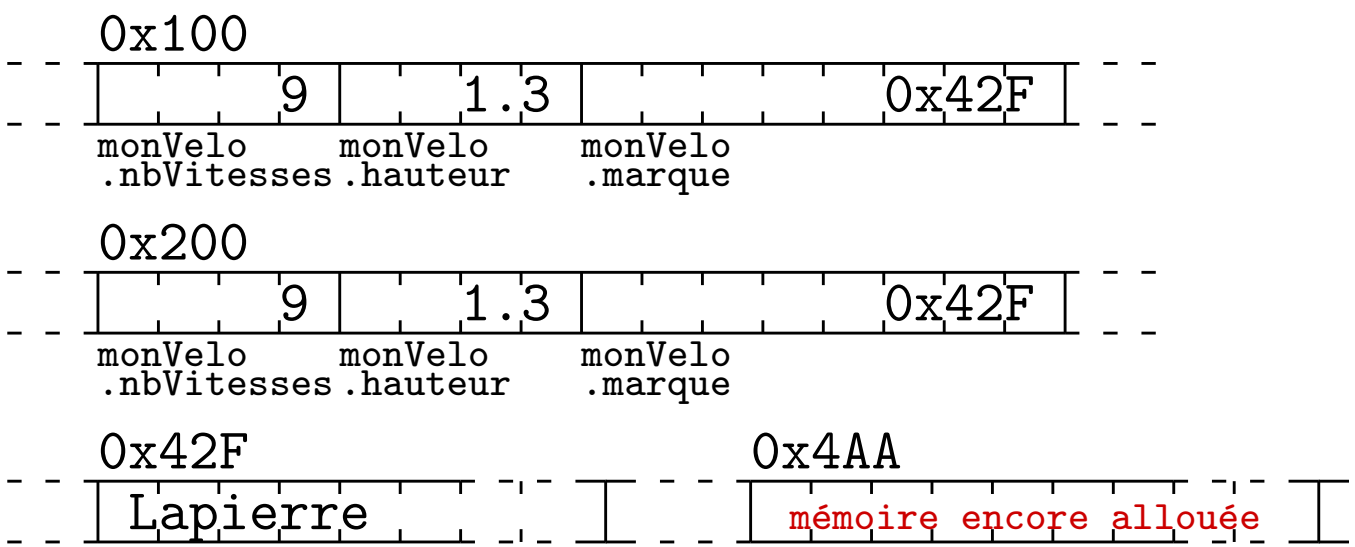
```
VeloV2 monVelo, autreVelo;
initVeloV2(&monVelo);
initVeloV2(&autreVelo);
monVelo.nbVitesses = 9;
monVelo.hauteur = 1.3;
-->strcpy(monVelo.marque, "Lapierre");
autreVelo = monVelo;
```



# Copie de structure contenant un tableau dynamique

- La copie via l'opérateur d'affectation d'une structure contenant un tableau génère une situation malsaine en mémoire :
  - plus aucune variable locale ne contient l'adresse `0x4AA`
  - monVelo et autreVelo pointent vers la même zone mémoire

```
VeloV2 monVelo, autreVelo;
initVeloV2(&monVelo);
initVeloV2(&autreVelo);
monVelo.nbVitesses = 9;
monVelo.hauteur = 1.3;
strcpy(monVelo.marque, "Lapierre");
-->autreVelo = monVelo;
```



# Copie de structure contenant un tableau dynamique

- Il est nécessaire d'écrire une fonction de recopie qui copie le contenu et non l'adresse de la chaîne de caractères.

```
void copieVeloV2(VeloV2* pDest, const VeloV2* pSrc)
{
    strcpy(pDest->marque, pSrc->marque);
    pDest->nbVitesses = pSrc->nbVitesses;
    pDest->hauteur = pSrc->hauteur;
}

/* Exemple d'utilisation */
VeloV2 monVelo, autreVelo;
initVeloV2(&monVelo);
initVeloV2(&autreVelo);
...
copieVeloV2(&monVelo, &autreVelo);
```

# Exemple

On souhaite écrire en C une application de gestion d'un atelier vélo. Plusieurs options sont possibles.

## Définition et initialisation d'un tableau statique de velov1

```
int main() {
    Velo atelier[10];
    int i;
    for (i=0; i<10; i++) {initVeloV1(&atelier[i]);}
    /* ... */
    return 0;
}
```

# Exemple

## Définition et initialisation d'un tableau dynamique de veloV1

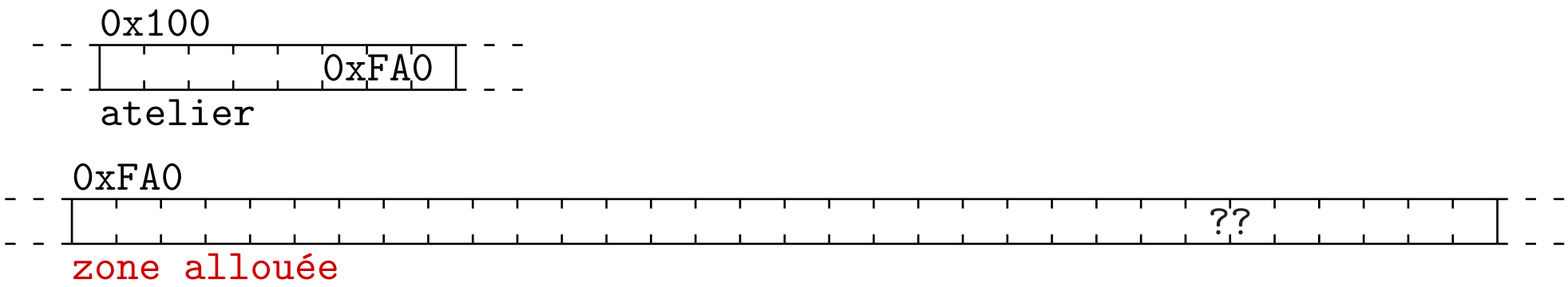
```
int main() {
    VeloV1 *atelier;
    int i, n=10;
    atelier = malloc(n*sizeof(VeloV1));
    for (i=0; i<n; i++) {initVeloV1(&atelier[i]);}
    /* ... */
    free(atelier);
    return 0;
}
```

# Exemple

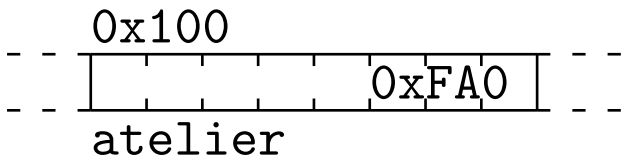
## Définition et initialisation d'un tableau statique de velov2

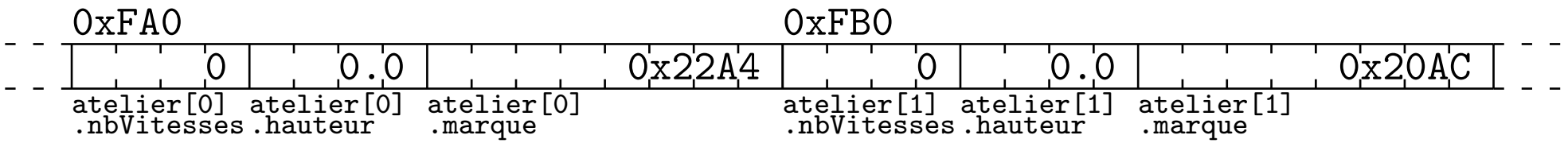
```
int main()
{
    VeloV2 atelier[10];
    int i;
    for (i=0; i<10; i++) {initVeloV2(&atelier[i]);}
    ...
    for (i=0; i<10; i++) {libereVeloV2(&atelier[i]);}
    return 0;
}
```

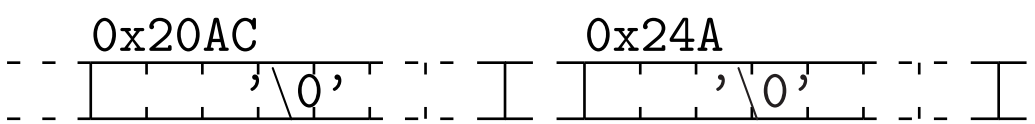
```
int main()
{
    VeloV2* atelier;
    int i, n=10;
 -->atelier = malloc(n*sizeof(VeloV2));
    for (i=0; i<n; i++) {initVeloV2(&atelier[i]);}
    ...
    for (i=0; i<n; i++) {libereVeloV2(&atelier[i]);}
    free(atelier);
    return 0;
}
```



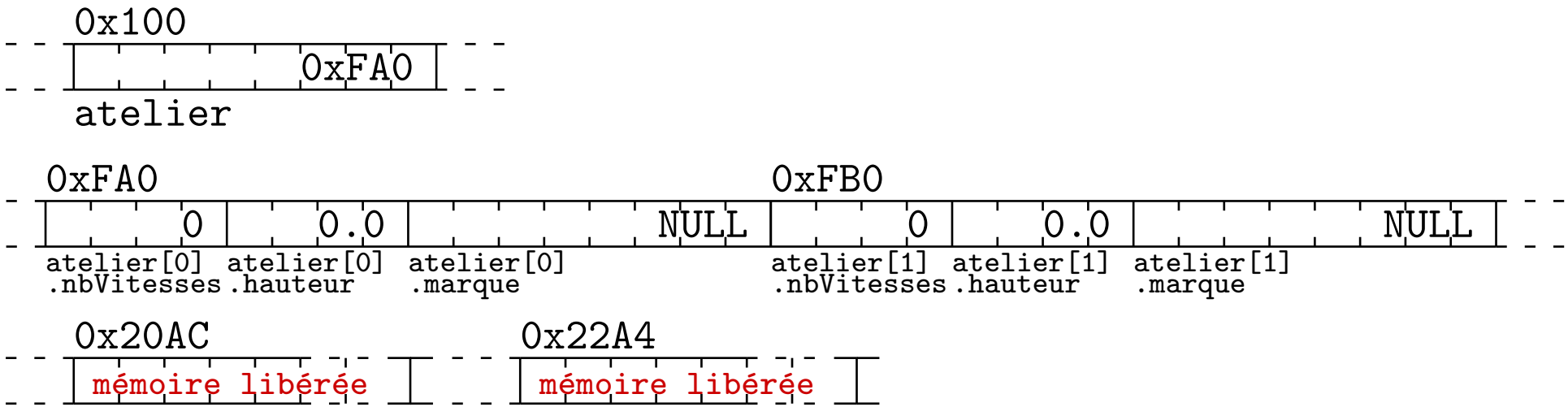
```
int main()
{
    VeloV2* atelier;
    int i, n=10;
    atelier = malloc(n*sizeof(VeloV2));
 -->for (i=0; i<n; i++) {initVeloV2(&atelier[i]);}
    ...
    for (i=0; i<n; i++) {libereVeloV2(&atelier[i]);}
    free(atelier);
    return 0;
}
```



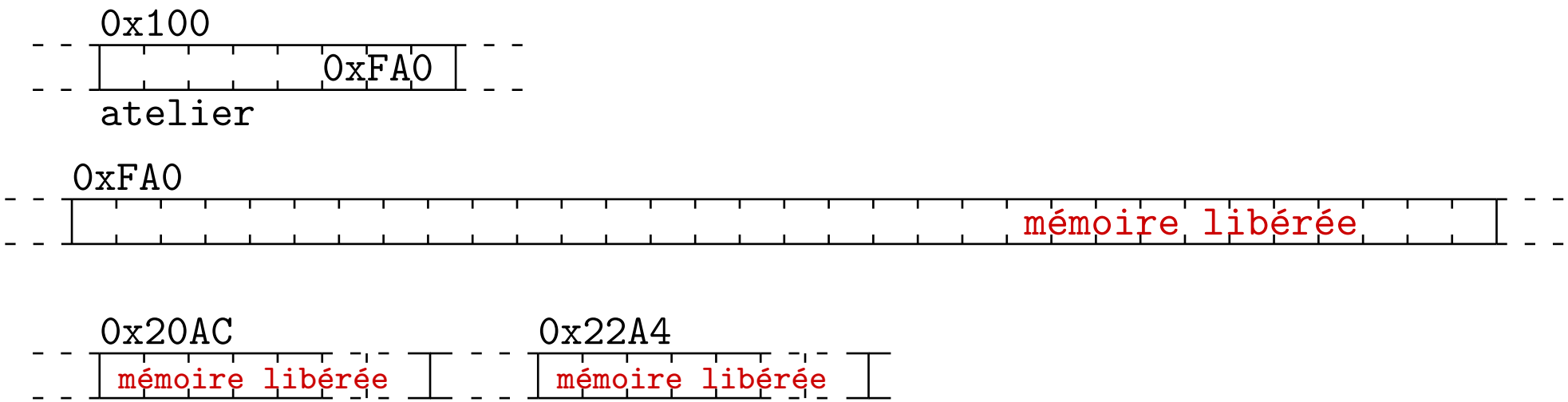



```c
int main()
{
    VeloV2* atelier;
    int i, n=10;
    atelier = malloc(n*sizeof(VeloV2));
    for (i=0; i<n; i++) {initVeloV2(&atelier[i]);}
    ...
 -->for (i=0; i<n; i++) {libereVeloV2(&atelier[i]);}
    free(atelier);
    return 0;
}
```



```c
int main()
{
    VeloV2* atelier;
    int i, n=10;
    atelier = malloc(n*sizeof(VeloV2));
    for (i=0; i<n; i++) {initVeloV2(&atelier[i]);}
    ...
    for (i=0; i<n; i++) {libereVeloV2(&atelier[i]);}
 -->free(atelier);
    return 0;
}
```




Attention à bien libérer chaque vélo avant de libérer l'atelier