

### Introduction

- De façon schématique, la mémoire d'un ordinateur s'apparente à un immense tableau dont chaque case a une adresse.

# C3 Pointeurs, types structurés

## 1. Mémoire en C

### Introduction

- De façon schématique, la mémoire d'un ordinateur s'apparente à un immense tableau dont chaque case a une adresse.
- Un **pointeur** est une variable contenant une de ces adresses.

# C3 Pointeurs, types structurés

## 1. Mémoire en C

### Introduction

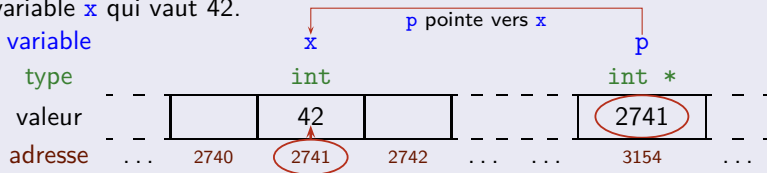
- De façon schématique, la mémoire d'un ordinateur s'apparente à un immense tableau dont chaque case a une adresse.
- Un **pointeur** est une variable contenant une de ces adresses.
- Le schéma ci-dessous représente un pointeur **p** contenant l'adresse d'une variable **x** qui vaut 42.

# C3 Pointeurs, types structurés

## 1. Mémoire en C

### Introduction

- De façon schématique, la mémoire d'un ordinateur s'apparente à un immense tableau dont chaque case a une adresse.
- Un **pointeur** est une variable contenant une de ces adresses.
- Le schéma ci-dessous représente un pointeur **p** contenant l'adresse d'une variable **x** qui vaut 42.

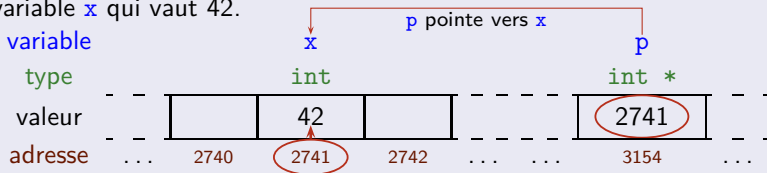


# C3 Pointeurs, types structurés

## 1. Mémoire en C

### Introduction

- De façon schématique, la mémoire d'un ordinateur s'apparente à un immense tableau dont chaque case a une adresse.
- Un **pointeur** est une variable contenant une de ces adresses.
- Le schéma ci-dessous représente un pointeur **p** contenant l'adresse d'une variable **x** qui vaut 42.

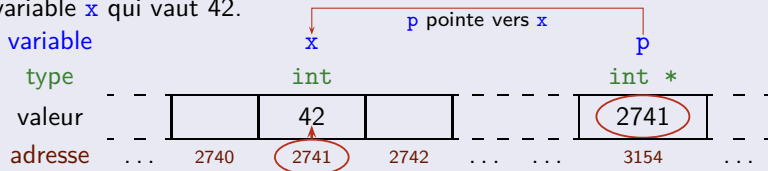


# C3 Pointeurs, types structurés

## 1. Mémoire en C

### Introduction

- De façon schématique, la mémoire d'un ordinateur s'apparente à un immense tableau dont chaque case a une adresse.
- Un **pointeur** est une variable contenant une de ces adresses.
- Le schéma ci-dessous représente un pointeur **p** contenant l'adresse d'une variable **x** qui vaut 42.



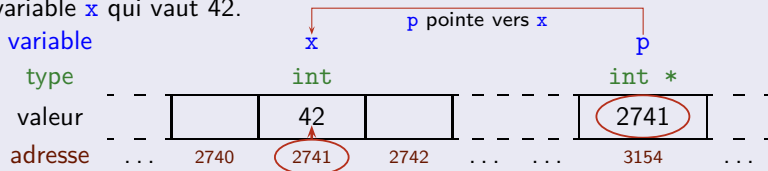
- En C, on peut directement manipuler des adresses mémoires (des pointeurs) :

# C3 Pointeurs, types structurés

## 1. Mémoire en C

### Introduction

- De façon schématique, la mémoire d'un ordinateur s'apparente à un immense tableau dont chaque case a une adresse.
- Un **pointeur** est une variable contenant une de ces adresses.
- Le schéma ci-dessous représente un pointeur **p** contenant l'adresse d'une variable **x** qui vaut 42.



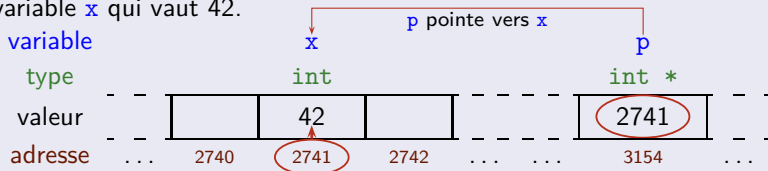
- En C, on peut directement manipuler des adresses mémoires (des pointeurs) :
  - &x** est l'adresse de la variable **x** (dans l'exemple précédent **&x = p (=2741)**), le type de **p** est **int \*** c'est à dire *adresse d'un entier*.

# C3 Pointeurs, types structurés

## 1. Mémoire en C

### Introduction

- De façon schématique, la mémoire d'un ordinateur s'apparente à un immense tableau dont chaque case a une adresse.
- Un **pointeur** est une variable contenant une de ces adresses.
- Le schéma ci-dessous représente un pointeur **p** contenant l'adresse d'une variable **x** qui vaut 42.



- En C, on peut directement manipuler des adresses mémoires (des pointeurs) :
  - &x** est l'adresse de la variable **x** (dans l'exemple précédent **&x = p (=2741)**), le type de **p** est **int \*** c'est à dire *adresse d'un entier*.
  - \*p** est le contenu de l'adresse **p** (dans l'exemple précédent **\*p = x (=42)**), le type de **\*p** est **int**.



# C3 Pointeurs, types structurés

## 1. Mémoire en C

### Schéma de l'organisation de la mémoire en C

# C3 Pointeurs, types structurés

## 1. Mémoire en C

### Schéma de l'organisation de la mémoire en C

↑  
-----  
adresses croissantes  
-----  
↑

# C3 Pointeurs, types structurés

## 1. Mémoire en C

### Schéma de l'organisation de la mémoire en C

↑  
-----  
adresses croissantes  
-----  
↑

Code compilé

En lecture seule

# C3 Pointeurs, types structurés

## 1. Mémoire en C

### Schéma de l'organisation de la mémoire en C

↑  
adresses croissantes

Données statiques

Code compilé

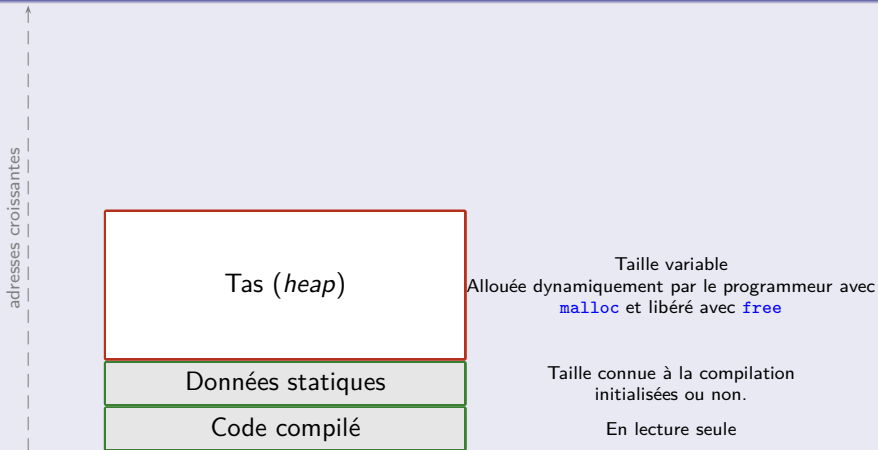
Taille connue à la compilation  
initialisées ou non.

En lecture seule

# C3 Pointeurs, types structurés

## 1. Mémoire en C

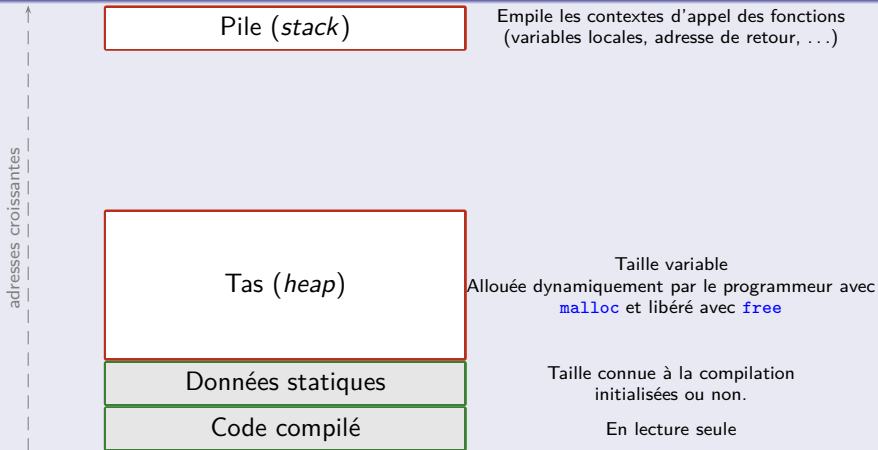
### Schéma de l'organisation de la mémoire en C



# C3 Pointeurs, types structurés

## 1. Mémoire en C

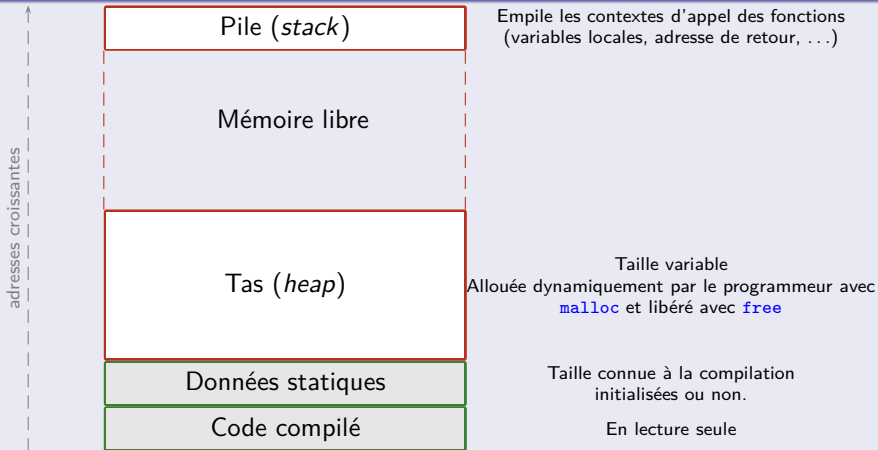
### Schéma de l'organisation de la mémoire en C



# C3 Pointeurs, types structurés

## 1. Mémoire en C

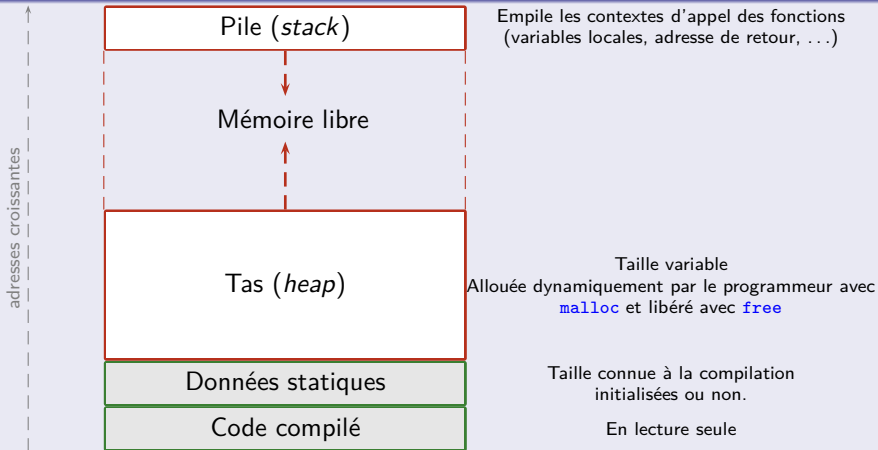
### Schéma de l'organisation de la mémoire en C



# C3 Pointeurs, types structurés

## 1. Mémoire en C

### Schéma de l'organisation de la mémoire en C

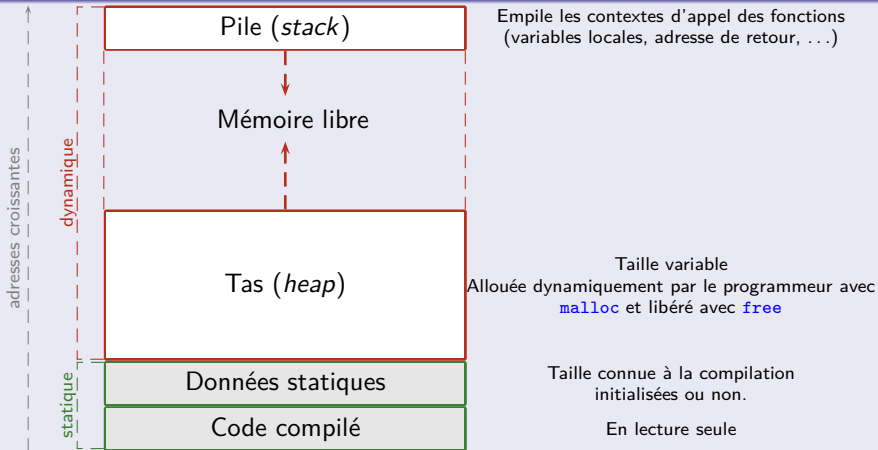




# C3 Pointeurs, types structurés

## 1. Mémoire en C

### Schéma de l'organisation de la mémoire en C



### Conséquences

Cette organisation de la mémoire a des conséquences importantes

- La taille de la pile est limitée (bien plus que celle du tas), donc une variable locale de taille importante risque de provoquer un débordement de pile (*stackoverflow*). Il est nettement préférable de l'allouer dans le tas.

### Conséquences

Cette organisation de la mémoire a des conséquences importantes

- La taille de la pile est limitée (bien plus que celle du tas), donc une variable locale de taille importante risque de provoquer un débordement de pile (*stackoverflow*). Il est nettement préférable de l'allouer dans le tas.
- Lors de l'appel à une fonction, les variables locales (et autres informations) sont stockées dans la pile. A la fin de l'exécution, ces informations sont supprimées de la pile. On ne doit donc pas conserver des adresses vers des variables locales.

### Conséquences

Cette organisation de la mémoire a des conséquences importantes

- La taille de la pile est limitée (bien plus que celle du tas), donc une variable locale de taille importante risque de provoquer un débordement de pile (*stackoverflow*). Il est nettement préférable de l'allouer dans le tas.
- Lors de l'appel à une fonction, les variables locales (et autres informations) sont stockées dans la pile. A la fin de l'exécution, ces informations sont supprimées de la pile. On ne doit donc pas conserver des adresses vers des variables locales.
- De la mémoire alloué par le programmeur dans le tas et non libérée est considérée comme non disponible, créant des fuites mémoires (*memory leak*).

# C3 Pointeurs, types structurés

## 1. Mémoire en C

### Un exemple

```
1  int var1; //Dans le segment de données (non initialisé)
2  int var2 = 42; //Dans le segment de données (initialisé)
3
4  int carre(int a)
5  {
6      int r = a*a; // Dans la pile lors d'un appel à carre
7      return r;
8  }
9
10 int main()
11 {
12     int var3; //Dans la pile (variable locale)
13     var3 = carre(var2); //Récupère la valeur renvoyée
14     return 0;
15 }
```

### Un second exemple

```
1  int main()  
2  {  
3      double big_array[1500000];  
4      return 0;  
5  }
```

- 1 Rappel la taille d'un `double`, en déduire la taille du tableau `big_array`

### Un second exemple

```
1  int main()  
2  {  
3      double big_array[1500000];  
4      return 0;  
5  }
```

- 1 Rappel la taille d'un `double`, en déduire la taille du tableau `big_array`
- 2 Comment expliquer que programme a provoqué une erreur de segmentation, alors qu'il a été exécuté sur une machine possédant 8 Go de mémoire vive ?

### Un second exemple

```
1  int main()  
2  {  
3      double big_array[1500000];  
4      return 0;  
5  }
```

- ❶ Rappeler la taille d'un `double`, en déduire la taille du tableau `big_array`
- ❷ Comment expliquer que programme a provoqué une erreur de segmentation, alors qu'il a été exécuté sur une machine possédant 8 Go de mémoire vive ?
- ❸ En déduire une information sur la taille de la pile sur cet ordinateur.



### Un second exemple

```
1  int main()  
2  {  
3      double big_array[1500000];  
4      return 0;  
5  }
```

- 1 Rappel la taille d'un `double`, en déduire la taille du tableau `big_array`
- 2 Comment expliquer que programme a provoqué une erreur de segmentation, alors qu'il a été exécuté sur une machine possédant 8 Go de mémoire vive ?
- 3 En déduire une information sur la taille de la pile sur cet ordinateur.
- 4 Comment résoudre ce problème ?

# C3 Pointeurs, types structurés

## 2. Pointeurs

### Opérateurs & et \*

- Si `t` est un type du C, par exemple (`int`, `char`, ...), alors `t *` est du type pointeur vers une valeur de type `t`. Par exemple :

# C3 Pointeurs, types structurés

## 2. Pointeurs

### Opérateurs & et \*

- Si `t` est un type du C, par exemple (`int`, `char`, ...), alors `t *` est du type pointeur vers une valeur de type `t`. Par exemple :

```
int *p; //déclare un pointeur p vers un entier
```

# C3 Pointeurs, types structurés

## 2. Pointeurs

### Opérateurs & et \*

- Si `t` est un type du C, par exemple (`int`, `char`, ...), alors `t *` est du type pointeur vers une valeur de type `t`. Par exemple :  
`int *p; //déclare un pointeur p vers un entier`
- Une valeur spéciale `NULL`, indique un pointeur qui ne pointe vers rien. Par exemple,

# C3 Pointeurs, types structurés

## 2. Pointeurs

### Opérateurs & et \*

- Si `t` est un type du C, par exemple (`int`, `char`, ...), alors `t *` est du type pointeur vers une valeur de type `t`. Par exemple :

```
int *p; //déclare un pointeur p vers un entier
```

- Une valeur spéciale `NULL`, indique un pointeur qui ne pointe vers rien. Par exemple,

```
char *c = NULL; // pointeur vers caractère initialisé à NULL
```

# C3 Pointeurs, types structurés

## 2. Pointeurs

### Opérateurs & et \*

- Si `t` est un type du C, par exemple (`int`, `char`, ...), alors `t *` est du type pointeur vers une valeur de type `t`. Par exemple :  
`int *p; //déclare un pointeur p vers un entier`
- Une valeur spéciale `NULL`, indique un pointeur qui ne pointe vers rien. Par exemple,  
`char *c = NULL; // pointeur vers caractère initialisé à NULL`
- L'opérateur unaire `&`, appelé opérateur d'adresse, permet en C de récupérer l'adresse mémoire d'une variable.

# C3 Pointeurs, types structurés

## 2. Pointeurs


### Opérateurs & et \*

- Si `t` est un type du C, par exemple (`int`, `char`, ...), alors `t *` est du type pointeur vers une valeur de type `t`. Par exemple :  
`int *p; //déclare un pointeur p vers un entier`
- Une valeur spéciale `NULL`, indique un pointeur qui ne pointe vers rien. Par exemple,  
`char *c = NULL; // pointeur vers caractère initialisé à NULL`
- L'opérateur unaire `&`, appelé opérateur d'adresse, permet en C de récupérer l'adresse mémoire d'une variable.
- L'opérateur unaire `*`, appelé opérateur de déréférencement, permet en C de récupérer la valeur stockée dans une adresse mémoire.
  - ❗ Déréférencer un pointeur `NULL` est un comportement indéfini.

# C3 Pointeurs, types structurés

## 2. Pointeurs

### Opérateurs & et \*

- Si `t` est un type du C, par exemple (`int`, `char`, ...), alors `t *` est du type pointeur vers une valeur de type `t`. Par exemple :  
`int *p; //déclare un pointeur p vers un entier`
- Une valeur spéciale `NULL`, indique un pointeur qui ne pointe vers rien. Par exemple,  
`char *c = NULL; // pointeur vers caractère initialisé à NULL`
- L'opérateur unaire `&`, appelé opérateur d'adresse, permet en C de récupérer l'adresse mémoire d'une variable.
- L'opérateur unaire `*`, appelé opérateur de déréférencement, permet en C de récupérer la valeur stockée dans une adresse mémoire.
-  Déréférencer un pointeur `NULL` est un comportement indéfini.

- Pour résumer :

	S'applique à	Permet de
<code>&amp;</code>	à une variable	récupérer son adresse
<code>*</code>	à un pointeur	récupérer la valeur à l'emplacement mémoire désigné



### Remarque

Lorsqu'on déclare un pointeur vers un entier par exemple avec `int * p;`, on peut considérer que :

# C3 Pointeurs, types structurés

## 2. Pointeurs

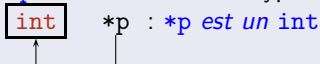
### Remarque

Lorsqu'on déclare un pointeur vers un entier par exemple avec `int * p;`, on peut considérer que :

- `p` est une variable de type `int*` qui contient l'adresse d'un entier :



- `*p` est une variable de type `int` qui contient la valeur de l'entier pointé par `p` :



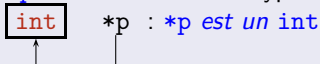
### Remarque

Lorsqu'on déclare un pointeur vers un entier par exemple avec `int * p;`, on peut considérer que :

- `p` est une variable de type `int*` qui contient l'adresse d'un entier :



- `*p` est une variable de type `int` qui contient la valeur de l'entier pointé par `p` :



On peut donc manipuler `p` :

- En lui attribuant l'adresse d'un entier, par exemple avec `p = &n` où `n` est un entier.
- En affectant un `int` à `*p`, à condition que l'espace mémoire pointé par `p` soit alloué. Par exemple avec `*p = n;` où `n` est un entier.

### Exemple 1

```
1  int main()
2  {
3      int n = 42; // Dans la pile (variable locale)
4      int *p = &n; // p est un pointeur vers n, stocké dans la pile
5      *p = 2024; // On modifie n (via le contenu de son adresse)
6      printf("Valeur de n = %d\n",n); // affiche 2024
7  }
```

### Exemple 2

# C3 Pointeurs, types structurés

## 2. Pointeurs

### Exemple 1

```
1  int main()
2  {
3      int n = 42; // Dans la pile (variable locale)
4      int *p = &n; // p est un pointeur vers n, stocké dans la pile
5      *p = 2024; // On modifie n (via le contenu de son adresse)
6      printf("Valeur de n = %d\n",n); // affiche 2024
7  }
```

Dans l'exemple ci-dessus, `p` est un pointeur qui contient l'adresse de `n`.

### Exemple 2

# C3 Pointeurs, types structurés

## 2. Pointeurs

### Exemple 1

```
1  int main()
2  {
3      int n = 42; // Dans la pile (variable locale)
4      int *p = &n; // p est un pointeur vers n, stocké dans la pile
5      *p = 2024; // On modifie n (via le contenu de son adresse)
6      printf("Valeur de n = %d\n",n); // affiche 2024
7  }
```

Dans l'exemple ci-dessus, `p` est un pointeur qui contient l'adresse de `n`. On modifie `n`, en modifiant le contenu de son adresse.

### Exemple 2

# C3 Pointeurs, types structurés

## 2. Pointeurs

### Exemple 1

```
1  int main()  
2  {  
3      int n = 42; // Dans la pile (variable locale)  
4      int *p = &n; // p est un pointeur vers n, stocké dans la pile  
5      *p = 2024; // On modifie n (via le contenu de son adresse)  
6      printf("Valeur de n = %d\n",n); // affiche 2024  
7  }
```

Dans l'exemple ci-dessus, `p` est un pointeur qui contient l'adresse de `n`. On modifie `n`, en modifiant le contenu de son adresse.

### Exemple 2

- 1 Ecrire une fonction `echange` qui prend en argument deux adresses vers des entiers et échange les valeurs de ces deux entiers.

# C3 Pointeurs, types structurés

## 2. Pointeurs

### Exemple 1

```
1  int main()  
2  {  
3      int n = 42; // Dans la pile (variable locale)  
4      int *p = &n; // p est un pointeur vers n, stocké dans la pile  
5      *p = 2024; // On modifie n (via le contenu de son adresse)  
6      printf("Valeur de n = %d\n",n); // affiche 2024  
7  }
```

Dans l'exemple ci-dessus, `p` est un pointeur qui contient l'adresse de `n`. On modifie `n`, en modifiant le contenu de son adresse.

### Exemple 2

- 1 Ecrire une fonction `echange` qui prend en argument deux adresses vers des entiers et échange les valeurs de ces deux entiers.
- 2 Comment utiliser cette fonction pour échanger les valeurs de deux entiers `a` et `b` ?



# C3 Pointeurs, types structurés

## 3. Fonctions `malloc` et `free`

### `malloc`

- La fonction `malloc` (disponible, comme `free`, après `#include <stdlib.h>`) permet d'allouer sur le tas, un bloc mémoire dont on donne la taille.

# C3 Pointeurs, types structurés

## 3. Fonctions `malloc` et `free`

### `malloc`

- La fonction `malloc` (disponible, comme `free`, après `#include <stdlib.h>`) permet d'allouer sur le tas, un bloc mémoire dont on donne la taille.
- Elle s'utilise donc souvent conjointement à `sizeof` qui donne la taille d'un objet en C.

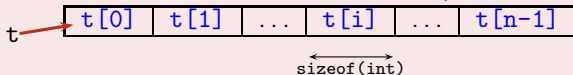
# C3 Pointeurs, types structurés

## 3. Fonctions malloc et free

### malloc

- La fonction `malloc` (disponible, comme `free`, après `#include <stdlib.h>`) permet d'allouer sur le tas, un bloc mémoire dont on donne la taille.
- Elle s'utilise donc souvent conjointement à `sizeof` qui donne la taille d'un objet en C.
- On peut allouer un bloc mémoire faisant  $n$  fois la taille d'un entier, et dans ce cas on accède aux différentes cases de ce bloc *de la même façon que pour un tableau*.

```
int *t = malloc(sizeof(int)*n); //n entier positif :
```



! Une syntaxe équivalente pour accéder à `t[i]` est `*(t+i)`.

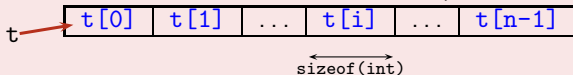
# C3 Pointeurs, types structurés

## 3. Fonctions malloc et free

### malloc

- La fonction `malloc` (disponible, comme `free`, après `#include <stdlib.h>`) permet d'allouer sur le tas, un bloc mémoire dont on donne la taille.
- Elle s'utilise donc souvent conjointement à `sizeof` qui donne la taille d'un objet en C.
- On peut allouer un bloc mémoire faisant  $n$  fois la taille d'un entier, et dans ce cas on accède aux différentes cases de ce bloc *de la même façon que pour un tableau*.

```
int *t = malloc(sizeof(int)*n); //n entier positif :
```



! Une syntaxe équivalente pour accéder à `t[i]` est `*(t+i)`.

- Comme pour les tableaux, accéder en dehors des limites du bloc alloué est un comportement indéfini.

# C3 Pointeurs, types structurés

## 3. Fonctions malloc et free

### Quelques exemples

- Pointeur non initialisé

```
1 int *ptr1; // Pointeur non initialisé
```

Le pointeur `ptr1`, est *non initialisé*, le déréfencer, par exemple pour lui attribuer une valeur (avec `*ptr1=42;`) est un comportement indéfini.

# C3 Pointeurs, types structurés

## 3. Fonctions malloc et free

### Quelques exemples

- Pointeur non initialisé

```
1 int *ptr1; // Pointeur non initialisé
```

Le pointeur `ptr1`, est *non initialisé*, le déréfencer, par exemple pour lui attribuer une valeur (avec `*ptr1=42;`) est un comportement indéfini.

- Pointeur initialisé à NULL

```
1 int *ptr2 = NULL; // Pointeur initialisé à NULL
```

Le pointeur `ptr2`, est initialisé à NULL, son déréférencement est un comportement indéfini.

# C3 Pointeurs, types structurés

## 3. Fonctions malloc et free

### Quelques exemples

- Pointeur non initialisé

```
1 int *ptr1; // Pointeur non initialisé
```

Le pointeur `ptr1`, est *non initialisé*, le déréféncer, par exemple pour lui attribuer une valeur (avec `*ptr1=42;`) est un comportement indéfini.

- Pointeur initialisé à NULL

```
1 int *ptr2 = NULL; // Pointeur initialisé à NULL
```

Le pointeur `ptr2`, est initialisé à NULL, son déréférencement est un comportement indéfini.

- Pointeur initialisé avec l'adresse d'une variable locale.

```
1 int n = 42;  
2 int* ptr3 = &n; // Pointeur vers une variable locale
```

Le pointeur `ptr3`, est initialisé avec l'adresse de `n` (situé dans la pile). Le déréférencement de `ptr3` donne la valeur de `n` (ici 42).

# C3 Pointeurs, types structurés

## 3. Fonctions malloc et free

### Quelques exemples

- Pointeur initialisé avec l'adresse d'un bloc mémoire alloué dynamiquement.

```
1 // Pointeur vers un bloc mémoire alloué dynamiquement
2 int* ptr4 = malloc(sizeof(int));
3 // Affectation de la valeur 14 au bloc mémoire pointé par
  ↪ ptr4
4 *ptr4 = 14;
```

Le pointeur `ptr4`, est initialisé avec l'adresse d'un bloc mémoire alloué dynamiquement. Ce bloc mémoire est situé dans le `tas`, cet espace mémoire est disponible pour y stocker un entier.



# C3 Pointeurs, types structurés

## 3. Fonctions `malloc` et `free`

### `free`

- La fonction `free` permet de libérer un bloc mémoire alloué grâce à `malloc`

# C3 Pointeurs, types structurés

## 3. Fonctions malloc et free

### free

- La fonction `free` permet de libérer un bloc mémoire alloué grâce à `malloc`
- On appelle donc `free` sur un pointeur `p` créé par `malloc`.  
⚠ Il est donc *impératif* de *toujours* disposer d'une référence vers un bloc mémoire qu'on a alloué.

```
1  int memleak(int a){
2      int* b = malloc(sizeof(int)); //b pointe vers le tas
3      *b = a*a;
4      return *b; //on renvoie la VALEUR de b
5  } //L'adresse de b est dépilée donc perdue
6
7  int main(){
8      int x = 42;
9      int y = memleak(x); //y récupère la valeur de b
10     printf("Valeur de y = %d\n", y);
11 } // On n'a pas de référence vers b, on ne peut pas free
```

# C3 Pointeurs, types structurés

## 3. Fonctions malloc et free

### free

- La fonction `free` permet de libérer un bloc mémoire alloué grâce à `malloc`
- On appelle donc `free` sur un pointeur `p` créé par `malloc`.  
⚠ Il est donc *impératif* de *toujours* disposer d'une référence vers un bloc mémoire qu'on a alloué.

```
1  int memleak(int a){
2      int* b = malloc(sizeof(int)); //b pointe vers le tas
3      *b = a*a;
4      return *b; //on renvoie la VALEUR de b
5  } //L'adresse de b est dépilée donc perdue
6
7  int main(){
8      int x = 42;
9      int y = memleak(x); //y récupère la valeur de b
10     printf("Valeur de y = %d\n", y);
11 } // On n'a pas de référence vers b, on ne peut pas free
```

# C3 Pointeurs, types structurés

## 3. Fonctions malloc et free

### free

- La fonction `free` permet de libérer un bloc mémoire alloué grâce à `malloc`
- On appelle donc `free` sur un pointeur `p` créé par `malloc`.  
⚠ Il est donc *impératif* de *toujours* disposer d'une référence vers un bloc mémoire qu'on a alloué.

```
1  int memleak(int a){
2      int* b = malloc(sizeof(int)); //b pointe vers le tas
3      *b = a*a;
4      return *b; //on renvoie la VALEUR de b
5  } //L'adresse de b est dépilée donc perdue
6
7  int main(){
8      int x = 42;
9      int y = memleak(x); //y récupère la valeur de b
10     printf("Valeur de y = %d\n", y);
11 } // On n'a pas de référence vers b, on ne peut pas free
```

- On utilisera toujours l'option `fsanitize = address` du compilateur pour détecter ces fuites mémoires.

# C3 Pointeurs, types structurés

## 3. Fonctions malloc et free

### Conséquences

En C, on n'écrira pas à *proprement parlé* de fonctions renvoyant un tableau, mais plutôt des fonctions renvoyant un pointeur vers un bloc mémoire alloué dynamiquement que l'on traitera *comme un tableau*.

## C3 Pointeurs, types structurés

### 3. Fonctions malloc et free

#### Conséquences

En C, on n'écrira pas à *proprement parlé* de fonctions renvoyant un tableau, mais plutôt des fonctions renvoyant un pointeur vers un bloc mémoire alloué dynamiquement que l'on traitera *comme un tableau*.

Il faut cependant respecter certaines règles :

### Conséquences

En C, on n'écrit pas à *proprement parlé* de fonctions renvoyant un tableau, mais plutôt des fonctions renvoyant un pointeur vers un bloc mémoire alloué dynamiquement que l'on traitera *comme un tableau*.

Il faut cependant respecter certaines règles :

- ⚠ L'allocation **doit** se faire avec `malloc`, sinon le bloc mémoire est alloué sur la pile, dans la zone de contexte d'appel de la fonction, cette zone est dépilée dès qu'on quitte la fonction et donc la zone mémoire du tableau n'est plus accessible.

### Conséquences

En C, on n'écrira pas à *proprement parlé* de fonctions renvoyant un tableau, mais plutôt des fonctions renvoyant un pointeur vers un bloc mémoire alloué dynamiquement que l'on traitera *comme un tableau*.

Il faut cependant respecter certaines règles :

- ⚠ L'allocation **doit** se faire avec **malloc**, sinon le bloc mémoire est alloué sur la pile, dans la zone de contexte d'appel de la fonction, cette zone est dépilé dès qu'on quitte la fonction et donc la zone mémoire du tableau n'est plus accessible.
- ⚠ La fonction appelante doit libérer la mémoire allouée avec **free** (sous peine de fuites mémoires.)



### Conséquences

En C, on n'écrira pas à *proprement parlé* de fonctions renvoyant un tableau, mais plutôt des fonctions renvoyant un pointeur vers un bloc mémoire alloué dynamiquement que l'on traitera *comme un tableau*.

Il faut cependant respecter certaines règles :

- ⚠ L'allocation **doit** se faire avec `malloc`, sinon le bloc mémoire est alloué sur la pile, dans la zone de contexte d'appel de la fonction, cette zone est dépillé dès qu'on quitte la fonction et donc la zone mémoire du tableau n'est plus accessible.
- ⚠ La fonction appelante doit libérer la mémoire allouée avec `free` (sous peine de fuites mémoires.)

D'autre part, on s'autorisera dans les exercices et à l'oral l'abus de langage « *renvoyer un tableau* ». Il faudra garder à l'esprit qu'on renvoie en fait un pointeur !

# C3 Pointeurs, types structurés

## 3. Fonctions malloc et free

### Exemple

Ecrire une fonction `int* make_tab(int size, int init)` qui renvoie un pointeur vers un bloc mémoire de `size` entiers tous initialisés à la valeur `init`.

# C3 Pointeurs, types structurés

## 3. Fonctions malloc et free

### Exemple

Ecrire une fonction `int* make_tab(int size, int init)` qui renvoie un pointeur vers un bloc mémoire de `size` entiers tous initialisés à la valeur `init`.

```
1  int *make_tab(int size, int init)
2  {
3      int *tab = malloc(sizeof(int) * size);
4      for (int i = 0; i < size; i++)
5      {
6          tab[i] = init;
7      }
8      return tab;
9  }
10
11 int main()
12 {
13     int *test_tab = make_tab(100, 42);
14     test_tab[0] = 34;
15     free(test_tab);
16 }
```

## C3 Pointeurs, types structurés

### 3. Fonctions malloc et free

#### Tableaux et pointeurs

⚠ En C, la confusion entre tableaux et pointeurs est fréquente et on a tendance à considérer que des écritures comme `int t[10];` et `int *t = malloc(sizeof(int)*10);` sont équivalentes. Bien qu'on puisse utiliser dans les deux cas la syntaxe `t[i]` (avec  $0 \leq i < 10$ ), c'est **faux** !  
Notamment :

## C3 Pointeurs, types structurés

### 3. Fonctions malloc et free

#### Tableaux et pointeurs

⚠ En C, la confusion entre tableaux et pointeurs est fréquente et on a tendance à considérer que des écritures comme `int t[10];` et `int *t = malloc(sizeof(int)*10);` sont équivalentes. Bien qu'on puisse utiliser dans les deux cas la syntaxe `t[i]` (avec  $0 \leq i < 10$ ), c'est **faux** !  
Notamment :

- Dans un cas `t` est alloué sur la pile, dans l'autre sur le tas.

### Tableaux et pointeurs

⚠ En C, la confusion entre tableaux et pointeurs est fréquente et on a tendance à considérer que des écritures comme `int t[10];` et `int *t = malloc(sizeof(int)*10);` sont équivalentes. Bien qu'on puisse utiliser dans les deux cas la syntaxe `t[i]` (avec  $0 \leq i < 10$ ), c'est **faux** !  
Notamment :

- Dans un cas `t` est alloué sur la pile, dans l'autre sur le tas.
- La fonction `sizeof` ne renvoie pas la même valeur dans les deux cas. En particulier il est possible d'obtenir la taille du tableau, pas celle du bloc mémoire.

### Tableaux et pointeurs

⚠ En C, la confusion entre tableaux et pointeurs est fréquente et on a tendance à considérer que des écritures comme `int t[10];` et `int *t = malloc(sizeof(int)*10);` sont équivalentes. Bien qu'on puisse utiliser dans les deux cas la syntaxe `t[i]` (avec  $0 \leq i < 10$ ), c'est **faux** !  
Notamment :

- Dans un cas `t` est alloué sur la pile, dans l'autre sur le tas.
- La fonction `sizeof` ne renvoie pas la même valeur dans les deux cas. En particulier il est possible d'obtenir la taille du tableau, pas celle du bloc mémoire.
- On ne peut pas avec un tableau utiliser directement l'opérateur d'assignation `=`, alors que cela est possible avec un pointeur.

### Tableaux et pointeurs

⚠ En C, la confusion entre tableaux et pointeurs est fréquente et on a tendance à considérer que des écritures comme `int t[10];` et `int *t = malloc(sizeof(int)*10);` sont équivalentes. Bien qu'on puisse utiliser dans les deux cas la syntaxe `t[i]` (avec  $0 \leq i < 10$ ), c'est **faux** !  
Notamment :

- Dans un cas `t` est alloué sur la pile, dans l'autre sur le tas.
- La fonction `sizeof` ne renvoie pas la même valeur dans les deux cas. En particulier il est possible d'obtenir la taille du tableau, pas celle du bloc mémoire.
- On ne peut pas avec un tableau utiliser directement l'opérateur d'assignation `=`, alors que cela est possible avec un pointeur.

Cependant, un tableau passé en argument à une fonction est traité comme un pointeur (par un mécanisme appelé *pointer decay*).



## C3 Pointeurs, types structurés

### 4. Argument en ligne de commande

#### Arguments de main

- La fonction `main` d'un programme C peut prendre en arguments un entier habituellement noté `argc` (*argument count*) et un tableau habituellement noté `argv[]` (*argument vector*) de chaînes de caractères.

# C3 Pointeurs, types structurés

## 4. Argument en ligne de commande

### Arguments de main

- La fonction `main` d'un programme C peut prendre en arguments un entier habituellement noté `argc` (*argument count*) et un tableau habituellement noté `argv[]` (*argument vector*) de chaînes de caractères.
- Ces arguments doivent alors être fournis à l'exécutable produit lors de la compilation.

### Arguments de main

- La fonction `main` d'un programme C peut prendre en arguments un entier habituellement noté `argc` (*argument count*) et un tableau habituellement noté `argv[]` (*argument vector*) de chaînes de caractères.
- Ces arguments doivent alors être fournis à l'exécutable produit lors de la compilation.
- Ces arguments sont traités comme des chaînes de caractères et doivent donc être convertis dans le type adéquat si besoin grâce aux fonctions suivantes disponibles dans `stdlib`.

## C3 Pointeurs, types structurés

### 4. Argument en ligne de commande

#### Arguments de main

- La fonction `main` d'un programme C peut prendre en arguments un entier habituellement noté `argc` (*argument count*) et un tableau habituellement noté `argv[]` (*argument vector*) de chaînes de caractères.
- Ces arguments doivent alors être fournis à l'exécutable produit lors de la compilation.
- Ces arguments sont traités comme des chaînes de caractères et doivent donc être convertis dans le type adéquat si besoin grâce aux fonctions suivantes disponibles dans `stdlib`.
  - La fonction `atoi` (*ASCII to integer*) permet de convertir une chaîne de caractères en un `int`

# C3 Pointeurs, types structurés

## 4. Argument en ligne de commande

### Arguments de main

- La fonction `main` d'un programme C peut prendre en arguments un entier habituellement noté `argc` (*argument count*) et un tableau habituellement noté `argv[]` (*argument vector*) de chaînes de caractères.
- Ces arguments doivent alors être fournis à l'exécutable produit lors de la compilation.
- Ces arguments sont traités comme des chaînes de caractères et doivent donc être convertis dans le type adéquat si besoin grâce aux fonctions suivantes disponibles dans `stdlib`.
  - La fonction `atoi` (*ASCII to integer*) permet de convertir une chaîne de caractères en un `int`
  - La fonction `atof` (*ASCII to float*) permet de convertir une chaîne de caractères en un `double`

## C3 Pointeurs, types structurés

### 4. Argument en ligne de commande

#### Exemple

Ecrire un exécutable `moyenne.exe` en C qui prend en argument sur la ligne de commande des flottants et écrit dans le terminal la moyenne de ces nombres. Par exemple `./moyenne.exe 12 10.5 16.5` doit écrire `13.0`.

# C3 Pointeurs, types structurés

## 4. Argument en ligne de commande

### Exemple

Ecrire un exécutable `moyenne.exe` en C qui prend en argument sur la ligne de commande des flottants et écrit dans le terminal la moyenne de ces nombres. Par exemple `./moyenne.exe 12 10.5 16.5` doit écrire `13.0`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[])
5  {
6      double somme = 0.0;
7      for (int i = 1; i < argc; i++)
8      {
9          somme = somme + atof(argv[i]);
10     }
11     printf("%f\n", somme / (argc - 1));
12 }
```

### Définitions

- On peut définir en C, des types structurés, appelé `struct` composés de plusieurs champs.



### Définitions

- On peut définir en C, des types structurés, appelé **struct** composés de plusieurs champs.
- La syntaxe générale de définition d'un type structuré est :

```
1  struct nom_type_struct {  
2      type1 elt1;  
3      type2 elt2;  
4      ...};
```

### Définitions

- On peut définir en C, des types structurés, appelé **struct** composés de plusieurs champs.
- La syntaxe générale de définition d'un type structuré est :

```
1  struct nom_type_struct {  
2      type1 elt1;  
3      type2 elt2;  
4      ...};
```

- Un nom de type (qui peut être celui du struct) peut être associé à un type structuré de façon à y faire référence plus rapidement.

**typedef struct nom\_type\_struct** nom\_type

Le programme de MP2I recommande d'utiliser `nom_type_s` pour le nom du struct et `nom_type` pour le type.

### Exemple

Pour créer le type structuré `personne_s` contenant les trois champs nom (chaîne de caractères), poids et taille (float) :

```
1 struct personne_s {  
2     char nom[50]; //Le nom est limité à 50 caractères max  
3     float poids;  
4     float taille;  
5 };
```

On peut donner un nom à ce type :

```
typedef struct personne_s personne;
```

### Déclaration, lecture et écriture d'un champ

- La déclaration d'une variable de type `personne` peut se faire maintenant avec :  
`personne bruce_banner;`

### Déclaration, lecture et écriture d'un champ

- La déclaration d'une variable de type personne peut se faire maintenant avec :  
`personne bruce_banner;`
- Eventuellement avec initialisation immédiate avec la notation `{` et `}` déjà rencontré sur les tableaux :  
`personne hulk = {.nom="Hulk", .poids = 635, .taille=2.43};`

### Déclaration, lecture et écriture d'un champ

- La déclaration d'une variable de type personne peut se faire maintenant avec :  
`personne bruce_banner;`
- Eventuellement avec initialisation immédiate avec la notation `{` et `}` déjà rencontré sur les tableaux :  
`personne hulk = {.nom="Hulk", .poids = 635, .taille=2.43};`
- On accède aux champs avec la notion `.`, pour les lire comme par exemple :  
`imc_hulk = hulk.poids / (hulk.taille*hulk.taille);`

### Déclaration, lecture et écriture d'un champ

- La déclaration d'une variable de type personne peut se faire maintenant avec :  
`personne bruce_banner;`
- Eventuellement avec initialisation immédiate avec la notation `{` et `}` déjà rencontré sur les tableaux :  
`personne hulk = {.nom="Hulk", .poids = 635, .taille=2.43};`
- On accède aux champs avec la notion `.`, pour les lire comme par exemple :  
`imc_hulk = hulk.poids / (hulk.taille*hulk.taille);`
- Ou pour les modifier, comme par exemple :  
`bruce_banner.taille = 1.75;`

# C3 Pointeurs, types structurés

## 5. Types structurés

### Exemple : Hulk fait un régime

Quel sera l'affichage produit par le programme ci-dessous, pourquoi ?

```
1  struct personne_s
2  {
3      char nom[50];
4      float taille;
5      float poids;
6  };
7  typedef struct personne_s personne;
8
9  void change_poids(personne p, float modification)
10 {
11     p.poids = p.poids + modification;
12 }
13
14 int main()
15 {
16     personne hulk = {.nom = "Hulk", .poids = 650, .taille = 2.50};
17     change_poids(hulk, -100.0);
18     printf("Le poids de %s est : %f\n", hulk.nom, hulk.poids);
19     return 0;
20 }
```



### Pointeur sur un struct

- L'utilisation de pointeurs sur des `struct` est courante en C.

Lorsqu'on passe un pointeur vers un `struct` on passe une unique valeur (l'adresse), alors que si on passe le struct, on doit passer une copie de chacun de ses champs. Donc on peut passer par référence, même si on ne modifie pas le struct afin d'économiser l'espace mémoire et le temps de recopie des champs.

### Exemple

### Pointeur sur un struct

- L'utilisation de pointeurs sur des `struct` est courante en C.  
Lorsqu'on passe un pointeur vers un `struct` on passe une unique valeur (l'adresse), alors que si on passe le struct, on doit passer une copie de chacun de ses champs. Donc on peut passer par référence, même si on ne modifie pas le struct afin d'économiser l'espace mémoire et le temps de recopie des champs.
- Si `p` est un pointeur sur un struct (c'est-à-dire `*p` est un struct) alors on accède aux champs avec la notation : `(*p).nom_champ`

### Exemple

### Pointeur sur un struct

- L'utilisation de pointeurs sur des `struct` est courante en C.  
Lorsqu'on passe un pointeur vers un `struct` on passe une unique valeur (l'adresse), alors que si on passe le struct, on doit passer une copie de chacun de ses champs. Donc on peut passer par référence, même si on ne modifie pas le struct afin d'économiser l'espace mémoire et le temps de recopie des champs.
- Si `p` est un pointeur sur un struct (c'est-à-dire `*p` est un struct) alors on accède aux champs avec la notation : `(*p).nom_champ`
- La notation précédente est raccourcie en `p->nom_champ`.

### Exemple

# C3 Pointeurs, types structurés

## 5. Types structurés

### Pointeur sur un struct

- L'utilisation de pointeurs sur des `struct` est courante en C.  
Lorsqu'on passe un pointeur vers un `struct` on passe une unique valeur (l'adresse), alors que si on passe le struct, on doit passer une copie de chacun de ses champs. Donc on peut passer par référence, même si on ne modifie pas le struct afin d'économiser l'espace mémoire et le temps de recopie des champs.
- Si `p` est un pointeur sur un struct (c'est-à-dire `*p` est un struct) alors on accède aux champs avec la notation : `(*p).nom_champ`
- La notation précédente est raccourcie en `p->nom_champ`.

### Exemple

Proposer une version correcte de la fonction modifiant le champ poids d'une variable de type struct `personne`.

### Correction

```
1  #include <stdio.h>
2
3  struct personne_struct{
4      char nom[50];
5      float taille;
6      float poids;};
7  typedef struct personne_struct personne;
8
9  void change_poids(personne *p, float modification){
10     p->poids = p->poids + modification;}
11
12  int main(){
13     personne hulk = {.nom = "Hulk", .poids=650, .taille=2.50};
14     change_poids(&hulk, -100.0);
15     printf("Le poids de %s est : %f\n", hulk.nom, hulk.poids);
16     return 0;}
```

### Manipulation des fichiers en C

- L'ouverture d'un fichier se fait à l'aide de `fopen` qui prend comme arguments le nom du fichier et le mode d'ouverture :

### Manipulation des fichiers en C

- L'ouverture d'un fichier se fait à l'aide de `fopen` qui prend comme arguments le nom du fichier et le mode d'ouverture :
  - `"r"` pour un accès en lecture

### Manipulation des fichiers en C

- L'ouverture d'un fichier se fait à l'aide de `fopen` qui prend comme arguments le nom du fichier et le mode d'ouverture :
  - `"r"` pour un accès en lecture
  - `"w"` pour un accès en écriture (le fichier est détruit s'il existait)



### Manipulation des fichiers en C

- L'ouverture d'un fichier se fait à l'aide de `fopen` qui prend comme arguments le nom du fichier et le mode d'ouverture :
  - `"r"` pour un accès en lecture
  - `"w"` pour un accès en écriture (le fichier est détruit s'il existait)
- Cette fonction renvoie un pointeur vers un objet de type `FILE` qui correspond à un flux de données sur lequel on peut lire ou écrire.

### Manipulation des fichiers en C

- L'ouverture d'un fichier se fait à l'aide de `fopen` qui prend comme arguments le nom du fichier et le mode d'ouverture :
  - `"r"` pour un accès en lecture
  - `"w"` pour un accès en écriture (le fichier est détruit s'il existait)
- Cette fonction renvoie un pointeur vers un objet de type `FILE` qui correspond à un flux de données sur lequel on peut lire ou écrire.
- Les fonctions `fscanf` et `fprint` permet respectivement de lire et d'écrire sur le flux de données.

### Manipulation des fichiers en C

- L'ouverture d'un fichier se fait à l'aide de `fopen` qui prend comme arguments le nom du fichier et le mode d'ouverture :
  - `"r"` pour un accès en lecture
  - `"w"` pour un accès en écriture (le fichier est détruit s'il existait)
- Cette fonction renvoie un pointeur vers un objet de type `FILE` qui correspond à un flux de données sur lequel on peut lire ou écrire.
- Les fonctions `fscanf` et `fprint` permet respectivement de lire et d'écrire sur le flux de données.
- Dans les deux cas, il faut fournir en argument le flux de données ainsi que les spécificateurs de format des données à lire et un pointeur vers les variables lues/écrites.

### Manipulation des fichiers en C

- L'ouverture d'un fichier se fait à l'aide de `fopen` qui prend comme arguments le nom du fichier et le mode d'ouverture :
  - `"r"` pour un accès en lecture
  - `"w"` pour un accès en écriture (le fichier est détruit s'il existait)
- Cette fonction renvoie un pointeur vers un objet de type `FILE` qui correspond à un flux de données sur lequel on peut lire ou écrire.
- Les fonctions `fscanf` et `fprint` permet respectivement de lire et d'écrire sur le flux de données.
- Dans les deux cas, il faut fournir en argument le flux de données ainsi que les spécificateurs de format des données à lire et un pointeur vers les variables lues/écrites.
- La valeur spéciale `EOF` est renvoyée par `fscanf` lorsque la fin du fichier est atteinte.

### Manipulation des fichiers en C

- L'ouverture d'un fichier se fait à l'aide de `fopen` qui prend comme arguments le nom du fichier et le mode d'ouverture :
  - `"r"` pour un accès en lecture
  - `"w"` pour un accès en écriture (le fichier est détruit s'il existait)
- Cette fonction renvoie un pointeur vers un objet de type `FILE` qui correspond à un flux de données sur lequel on peut lire ou écrire.
- Les fonctions `fscanf` et `fprint` permet respectivement de lire et d'écrire sur le flux de données.
- Dans les deux cas, il faut fournir en argument le flux de données ainsi que les spécificateurs de format des données à lire et un pointeur vers les variables lues/écrites.
- La valeur spéciale `EOF` est renvoyée par `fscanf` lorsque la fin du fichier est atteinte.
- Pour fermer un fichier, on utilise `fclose`.

### Somme des entiers

On suppose qu'un fichier `entiers.txt` contient des entiers séparés par des espaces.

- Ecrire l'instruction permettant d'ouvrir ce fichier en mode lecture

### Somme des entiers

On suppose qu'un fichier `entiers.txt` contient des entiers séparés par des espaces.

- Ecrire l'instruction permettant d'ouvrir ce fichier en mode lecture

### Somme des entiers

On suppose qu'un fichier `entiers.txt` contient des entiers séparés par des espaces.

- Ecrire l'instruction permettant d'ouvrir ce fichier en mode lecture



### Somme des entiers

On suppose qu'un fichier `entiers.txt` contient des entiers séparés par des espaces.

- Ecrire l'instruction permettant d'ouvrir ce fichier en mode lecture

### Somme des entiers

On suppose qu'un fichier `entiers.txt` contient des entiers séparés par des espaces.

- Ecrire l'instruction permettant d'ouvrir ce fichier en mode lecture

```
1 FILE *fichier = fopen("entiers.txt", "r");
```

- Déclarer deux entiers `n` (qui va contenir les entiers lus) et `somme` initialisé à 0.

### Somme des entiers

On suppose qu'un fichier `entiers.txt` contient des entiers séparés par des espaces.

- Ecrire l'instruction permettant d'ouvrir ce fichier en mode lecture

```
1 FILE *fichier = fopen("entiers.txt", "r");
```

- Déclarer deux entiers `n` (qui va contenir les entiers lus) et `somme` initialisé à 0.

```
1 int n, somme = 0;
```

- Ecrire une boucle `while` permettant de lire chacun des entiers jusqu'à la fin du fichier et d'en faire la somme dans la variable `somme`

### Somme des entiers

On suppose qu'un fichier `entiers.txt` contient des entiers séparés par des espaces.

- Ecrire l'instruction permettant d'ouvrir ce fichier en mode lecture

```
1 FILE *fichier = fopen("entiers.txt", "r");
```

- Déclarer deux entiers `n` (qui va contenir les entiers lus) et `somme` initialisé à 0.

```
1 int n, somme = 0;
```

- Ecrire une boucle `while` permettant de lire chacun des entiers jusqu'à la fin du fichier et d'en faire la somme dans la variable `somme`

```
1 while (fscanf(fichier, "%d", &n) != EOF) {  
2     somme = somme + n;}
```

- Ecrire l'instruction permettant de fermer le fichier

### Somme des entiers

On suppose qu'un fichier `entiers.txt` contient des entiers séparés par des espaces.

- Ecrire l'instruction permettant d'ouvrir ce fichier en mode lecture

```
1 FILE *fichier = fopen("entiers.txt", "r");
```

- Déclarer deux entiers `n` (qui va contenir les entiers lus) et `somme` initialisé à 0.

```
1 int n, somme = 0;
```

- Ecrire une boucle `while` permettant de lire chacun des entiers jusqu'à la fin du fichier et d'en faire la somme dans la variable `somme`

```
1 while (fscanf(fichier, "%d", &n) != EOF) {  
2     somme = somme + n;}
```

- Ecrire l'instruction permettant de fermer le fichier

```
1 fclose(fichier);
```

### Programme complet

```
1  #include <stdio.h>
2
3  int main(){
4      FILE *fichier = fopen("entiers.txt","r");
5      int n, somme = 0;
6      while (fscanf(fichier,"%d",&n)!=EOF) {
7          somme = somme + n;}
8      fclose(fichier);
9      printf("Somme = %d\n",somme);
10     return 0;}
```

### Principe

- De la même façon que certaines fonctions du langage C sont écrites dans des modules séparés et inclus à l'aide de la directive `#include` en début de programme, on peut écrire ses propres modules et y écrire des fonctions destinées à être réutilisées dans différents programmes.

### Principe

- De la même façon que certaines fonctions du langage C sont écrites dans des modules séparés et inclus à l'aide de la directive `#include` en début de programme, on peut écrire ses propres modules et y écrire des fonctions destinées à être réutilisées dans différents programmes.
- Ces modules peuvent être **compilés séparément**, ce qui induit de nombreux avantages (réduction de la taille du programme principal, structuration de l'application, maintenance facilitée du programme, ...)



### Méthode

### Méthode

- On commence par simplement écrire les signatures des fonctions dans un fichier `<module.h>`. C'est le fichier d'en-tête (le **h** vient de l'anglais *header*).

### Méthode

- On commence par simplement écrire les signatures des fonctions dans un fichier `<module.h>`. C'est le fichier d'en-tête (le `h` vient de l'anglais *header*).
- Les corps des fonctions sont écrites dans `<module.c>` et ce fichier est compilé de façon à obtenir un fichier objet `module.o` grâce à l'option `-c` de gcc.

### Méthode

- On commence par simplement écrire les signatures des fonctions dans un fichier `<module.h>`. C'est le fichier d'en-tête (le `h` vient de l'anglais *header*).
- Les corps des fonctions sont écrites dans `<module.c>` et ce fichier est compilé de façon à obtenir un fichier objet `module.o` grâce à l'option `-c` de `gcc`.
- Dans le programme principal utilisant le module, on écrit au début `#include "module.h"` ce qui permet de faire référence à ces fonctions sans déclencher de *warning* à la compilation.


### Méthode


- On commence par simplement écrire les signatures des fonctions dans un fichier `<module.h>`. C'est le fichier d'en-tête (le `h` vient de l'anglais *header*).
- Les corps des fonctions sont écrites dans `<module.c>` et ce fichier est compilé de façon à obtenir un fichier objet `module.o` grâce à l'option `-c` de gcc.
- Dans le programme principal utilisant le module, on écrit au début `#include "module.h"` ce qui permet de faire référence à ces fonctions sans déclencher de *warning* à la compilation.  
⚠ Ce sont bien des guillemets et pas `<` et `>`.
- Le programme principal est lui aussi compilé avec l'option `-c` de gcc afin de produire un fichier `main.o`.

### Méthode


- On commence par simplement écrire les signatures des fonctions dans un fichier `<module.h>`. C'est le fichier d'en-tête (le `h` vient de l'anglais *header*).
- Les corps des fonctions sont écrites dans `<module.c>` et ce fichier est compilé de façon à obtenir un fichier objet `module.o` grâce à l'option `-c` de gcc.
- Dans le programme principal utilisant le module, on écrit au début `#include "module.h"` ce qui permet de faire référence à ces fonctions sans déclencher de *warning* à la compilation.  
⚠ Ce sont bien des guillemets et pas `<` et `>`.
- Le programme principal est lui aussi compilé avec l'option `-c` de gcc afin de produire un fichier `main.o`.
- Enfin, on lie ensemble les deux fichiers objets pour produire l'exécutable grâce à la ligne de compilation :  
`gcc module.o main.o -o main.exe`


### Schéma des étapes d'une compilation séparée


 module.h

 main.c

 module.c

 main.o

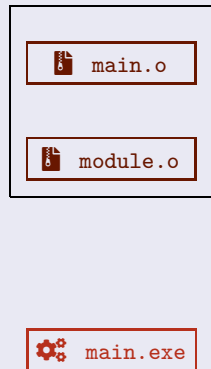
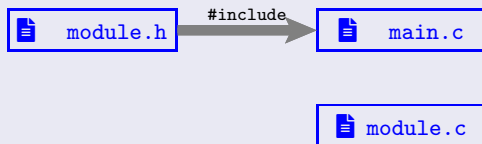
 module.o

 main.exe

# C3 Pointeurs, types structurés

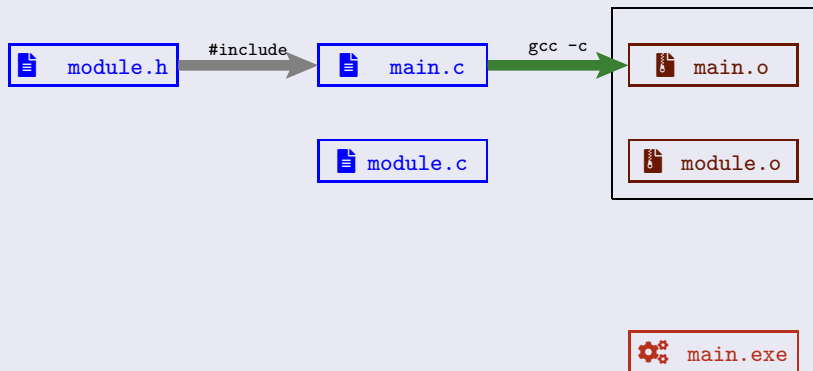
## 7. Compilation séparée

### Schéma des étapes d'une compilation séparée

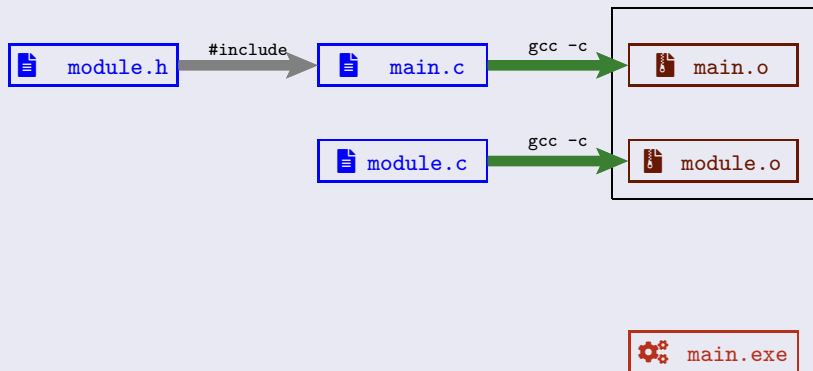




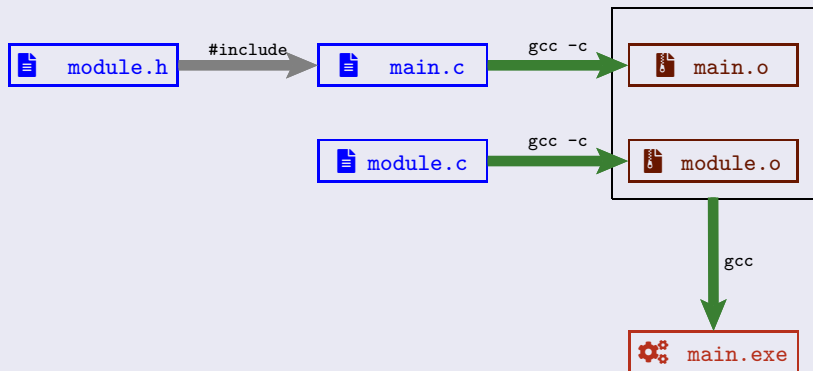
### Schéma des étapes d'une compilation séparée



### Schéma des étapes d'une compilation séparée



### Schéma des étapes d'une compilation séparée



# C3 Pointeurs, types structurés

## 7. Compilation séparée

### Remarques

- Des types composés `struct` peuvent être définis dans le fichier d'en-tête sans donner leur structure. Ce point sera détaillé plus loin dans le cours lorsqu'on abordera les structures de données.

### Remarques

- Des types composés `struct` peuvent être définis dans le fichier d'en-tête sans donner leur structure. Ce point sera détaillé plus loin dans le cours lorsqu'on abordera les structures de données.
- Afin d'éviter des redondances, on peut spécifier dans le fichier d'en-tête

```
1  #ifndef NONMODULE  
2  #define NONMODULE  
3  // ici définition des signatures des fonctions  
4  // et ici des types structurés éventuels  
5  #endif
```

Cela évite d'avoir des problèmes en cas de double inclusion malheureuse du même fichier.

### Remarques

- Des types composés `struct` peuvent être définis dans le fichier d'en-tête sans donner leur structure. Ce point sera détaillé plus loin dans le cours lorsqu'on abordera les structures de données.
- Afin d'éviter des redondances, on peut spécifier dans le fichier d'en-tête

```
1  #ifndef NONMODULE
2  #define NONMODULE
3  // ici définition des signatures des fonctions
4  // et ici des types structurés éventuels
5  #endif
```

Cela évite d'avoir des problèmes en cas de double inclusion malheureuse du même fichier.

- Afin d'éviter des conflits entre des fonctions portant le même nom et définies dans des modules ou dans le programme principale, on peut préfixer tous les noms de fonctions ou de type d'un module par le nom de ce module.

### Un exemple

On décide de créer un module `entiers` afin d'y définir des fonctions usuelles sur les entiers non présentes en C. Par exemple le calcul de puissances ou encore le calcul du PGCD :

### Un exemple

On décide de créer un module `entiers` afin d'y définir des fonctions usuelles sur les entiers non présentes en C. Par exemple le calcul de puissances ou encore le calcul du PGCD :

- 1 On commence par créer `entiers.h` et y écrire les signatures de nos fonctions



### Un exemple

On décide de créer un module `entiers` afin d'y définir des fonctions usuelles sur les entiers non présentes en C. Par exemple le calcul de puissances ou encore le calcul du PGCD :

- 1 On commence par créer `entiers.h` et y écrire les signatures de nos fonctions  
On y trouvera par exemple  

```
int entiers_puissance(int a, int n);
```
- 2 On crée le fichier `entiers.c` qui contient ces fonctions et on le compile avec `gcc -c entiers.c`, cela produit un fichier `entiers.o`

### Un exemple

On décide de créer un module `entiers` afin d'y définir des fonctions usuelles sur les entiers non présentes en C. Par exemple le calcul de puissances ou encore le calcul du PGCD :

- 1 On commence par créer `entiers.h` et y écrire les signatures de nos fonctions  
On y trouvera par exemple  

```
int entiers_puissance(int a, int n);
```
- 2 On crée le fichier `entiers.c` qui contient ces fonctions et on le compile avec `gcc -c entiers.c`, cela produit un fichier `entiers.o`
- 3 Dans le programme principal on inclut le fichier `entiers.h` avec :  

```
#include "entiers.h"
```

### Un exemple

On décide de créer un module `entiers` afin d'y définir des fonctions usuelles sur les entiers non présentes en C. Par exemple le calcul de puissances ou encore le calcul du PGCD :

- 1 On commence par créer `entiers.h` et y écrire les signatures de nos fonctions  
On y trouvera par exemple  

```
int entiers_puissance(int a, int n);
```
- 2 On crée le fichier `entiers.c` qui contient ces fonctions et on le compile avec `gcc -c entiers.c`, cela produit un fichier `entiers.o`
- 3 Dans le programme principal on inclut le fichier `entiers.h` avec :  

```
#include "entiers.h"
```
- 4 On compile le programme principal avec `gcc -c main.c`, cela produit un fichier `main.o`

### Un exemple

On décide de créer un module `entiers` afin d'y définir des fonctions usuelles sur les entiers non présentes en C. Par exemple le calcul de puissances ou encore le calcul du PGCD :

- 1 On commence par créer `entiers.h` et y écrire les signatures de nos fonctions  
On y trouvera par exemple  

```
int entiers_puissance(int a, int n);
```
- 2 On crée le fichier `entiers.c` qui contient ces fonctions et on le compile avec `gcc -c entiers.c`, cela produit un fichier `entiers.o`
- 3 Dans le programme principal on inclut le fichier `entiers.h` avec :  

```
#include "entiers.h"
```
- 4 On compile le programme principal avec `gcc -c main.c`, cela produit un fichier `main.o`
- 5 Enfin, on produit l'exécutable grâce à la compilation `gcc main.o entiers.o -o main.exe`

### Allocation de l'espace mémoire

- En C, on peut allouer des tableaux multidimensionnels sur la pile en précisant les tailles attendues pour chaque dimension. Par exemple :

### Allocation de l'espace mémoire

- En C, on peut allouer des tableaux multidimensionnels sur la pile en précisant les tailles attendues pour chaque dimension. Par exemple :

```
float matrice[4][4];
```

déclare un tableau de flottants de 4 lignes sur 4 colonnes. L'accès aux élément se fait alors avec la notation `matrice[i][j]`.

### Allocation de l'espace mémoire

- En C, on peut allouer des tableaux multidimensionnels sur la pile en précisant les tailles attendues pour chaque dimension. Par exemple :

```
float matrice[4][4];
```

déclare un tableau de flottants de 4 lignes sur 4 colonnes. L'accès aux éléments se fait alors avec la notation `matrice[i][j]`.

La taille de la pile étant limitée, cela est réservé aux tableaux de petites tailles.

### Allocation de l'espace mémoire

- En C, on peut allouer des tableaux multidimensionnels sur la pile en précisant les tailles attendues pour chaque dimension. Par exemple :

```
float matrice[4][4];
```

déclare un tableau de flottants de 4 lignes sur 4 colonnes. L'accès aux élément se fait alors avec la notation `matrice[i][j]`.

La taille de la pile étant limitée, cela est réservé aux tableaux de petites tailles.

- Pour une allocation sur le tas d'un tableau  $t$ , de  $n$  lignes et  $m$  colonnes, (qui sera nécessaire pour les tableaux de taille importante), trois options sont envisageables :



### Allocation de l'espace mémoire

- En C, on peut allouer des tableaux multidimensionnels sur la pile en précisant les tailles attendues pour chaque dimension. Par exemple :

```
float matrice[4][4];
```

déclare un tableau de flottants de 4 lignes sur 4 colonnes. L'accès aux élément se fait alors avec la notation `matrice[i][j]`.

La taille de la pile étant limitée, cela est réservé aux tableaux de petites tailles.

- Pour une allocation sur le tas d'un tableau  $t$ , de  $n$  lignes et  $m$  colonnes, (qui sera nécessaire pour les tableaux de taille importante), trois options sont envisageables :

$V_0$  Allouer un tableau à une dimension de taille  $n \times m$ .

### Allocation de l'espace mémoire

- En C, on peut allouer des tableaux multidimensionnels sur la pile en précisant les tailles attendues pour chaque dimension. Par exemple :

```
float matrice[4][4];
```

déclare un tableau de flottants de 4 lignes sur 4 colonnes. L'accès aux élément se fait alors avec la notation `matrice[i][j]`.

La taille de la pile étant limitée, cela est réservé aux tableaux de petites tailles.

- Pour une allocation sur le tas d'un tableau  $t$ , de  $n$  lignes et  $m$  colonnes, (qui sera nécessaire pour les tableaux de taille importante), trois options sont envisageables :

$V_0$  Allouer un tableau à une dimension de taille  $n \times m$ .

$V_1$  Allouer un tableau de tableau à une dimension.

### Allocation de l'espace mémoire

- En C, on peut allouer des tableaux multidimensionnels sur la pile en précisant les tailles attendues pour chaque dimension. Par exemple :

```
float matrice[4][4];
```

déclare un tableau de flottants de 4 lignes sur 4 colonnes. L'accès aux élément se fait alors avec la notation `matrice[i][j]`.

La taille de la pile étant limitée, cela est réservé aux tableaux de petites tailles.

- Pour une allocation sur le tas d'un tableau  $t$ , de  $n$  lignes et  $m$  colonnes, (qui sera nécessaire pour les tableaux de taille importante), trois options sont envisageables :

$V_0$  Allouer un tableau à une dimension de taille  $n \times m$ .

$V_1$  Allouer un tableau de tableau à une dimension.

$V_2$  Allouer un tableau de pointeurs, chacun des pointeurs pointe alors vers une ligne du tableau.

### $V_0$ : Allocation d'un tableau à une dimension

- On alloue un tableau d'entiers de tailles  $n \times m$  et on considère que les  $m$  premiers forment la première ligne, que les  $m$  suivant sont la deuxième, ...

# C3 Pointeurs, types structurés

## 8. Tableaux multidimensionnels

### $V_0$ : Allocation d'un tableau à une dimension

- On alloue un tableau d'entiers de tailles  $n \times m$  et on considère que les  $m$  premiers forment la première ligne, que les  $m$  suivant sont la deuxième, ....
- La structure de données correspondante et "à visualiser" est alors la suivante :



## $V_0$ : Allocation d'un tableau à une dimension

- On alloue un tableau d'entiers de tailles  $n \times m$  et on considère que les  $m$  premiers forment la première ligne, que les  $m$  suivant sont la deuxième, ...
- La structure de données correspondante et "à visualiser" est alors la suivante :

$$\underbrace{\boxed{t(0,0)} \quad \dots \quad \boxed{t(0,m-1)}}_{\text{ligne 0 (m cases)}} \quad \dots \quad \underbrace{\boxed{t(n-1,0)} \quad \dots \quad \boxed{t(n-1,m-1)}}_{\text{ligne n-1 (m cases)}}$$

- Pour accéder à l'élément situé en colonne  $j$  et ligne  $i$  on doit donc faire `tab[m*i+j]`, c'est-à-dire qu'on perd la notation (pourtant pratique) `tab[i][j]` des tableaux à deux dimensions, c'est le seul inconvénient de cette méthode.

# C3 Pointeurs, types structurés

## 8. Tableaux multidimensionnels

### $V_0$ : Allocation d'un tableau à une dimension

- On alloue un tableau d'entiers de tailles  $n \times m$  et on considère que les  $m$  premiers forment la première ligne, que les  $m$  suivants sont la deuxième, ....
- La structure de données correspondante et "à visualiser" est alors la suivante :



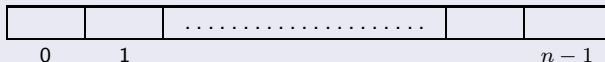
- Pour accéder à l'élément situé en colonne  $j$  et ligne  $i$  on doit donc faire `tab[m*i+j]`, c'est-à-dire qu'on perd la notation (pourtant pratique) `tab[i][j]` des tableaux à deux dimensions, c'est le seul inconvénient de cette méthode.
- Cette solution est la plus simple et sera celle à privilégier cette année. En effet, on ne fait qu'un `malloc` et un `free` et on manipule en fait un tableau à une dimension (ce qu'on doit normalement savoir faire !)

## C3 Pointeurs, types structurés

### 8. Tableaux multidimensionnels

#### $V_1$ : Allocation d'un tableau de tableaux

- Un tableau d'entier de type `int` et de taille  $n$  à la structure suivante en mémoire (où chaque case à la taille prévue pour accueillir un `int`)



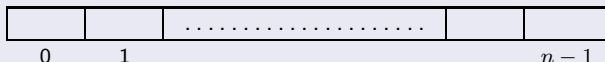


# C3 Pointeurs, types structurés

## 8. Tableaux multidimensionnels

### $V_1$ : Allocation d'un tableau de tableaux

- Un tableau d'entier de type `int` et de taille  $n$  à la structure suivante en mémoire (où chaque case à la taille prévue pour accueillir un `int`)



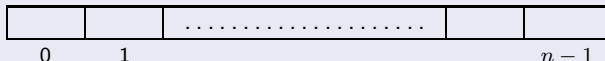
- Pour allouer une matrice de dimension  $n \times m$ , on peut "linéariser" le tableau et prévoir que chaque case à maintenant la capacité d'accueillir  $m$  entiers. *c'est-à-dire que c'est la solution précédente mais avec une gestion automatique des indices par le langage C.*

# C3 Pointeurs, types structurés

## 8. Tableaux multidimensionnels

### $V_1$ : Allocation d'un tableau de tableaux

- Un tableau d'entier de type `int` et de taille  $n$  à la structure suivante en mémoire (où chaque case à la taille prévue pour accueillir un `int`)



- Pour allouer une matrice de dimension  $n \times m$ , on peut "linéariser" le tableau et prévoir que chaque case à maintenant la capacité d'accueillir  $m$  entiers. *c'est-à-dire que c'est la solution précédente mais avec une gestion automatique des indices par le langage C.*
- La structure de données correspondante et "à visualiser" est alors la suivante :



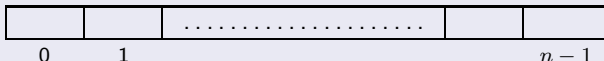
Chaque case du "grand" tableau contient un tableau de  $m$  entiers.

# C3 Pointeurs, types structurés

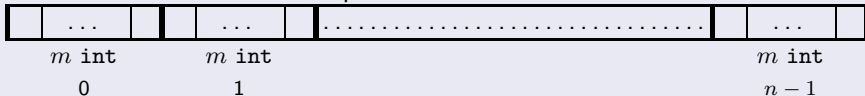
## 8. Tableaux multidimensionnels

### $V_1$ : Allocation d'un tableau de tableaux

- Un tableau d'entier de type `int` et de taille  $n$  à la structure suivante en mémoire (où chaque case à la taille prévue pour accueillir un `int`)



- Pour allouer une matrice de dimension  $n \times m$ , on peut "linéariser" le tableau et prévoir que chaque case à maintenant la capacité d'accueillir  $m$  entiers. *c'est-à-dire que c'est la solution précédente mais avec une gestion automatique des indices par le langage C.*
- La structure de données correspondante et "à visualiser" est alors la suivante :



Chaque case du "grand" tableau contient un tableau de  $m$  entiers.

- Cette allocation s'effectue avec :

```
int (*mat)[n] = malloc(sizeof(*mat)*m);
```

### $V_2$ : Allocation d'un tableau de pointeurs

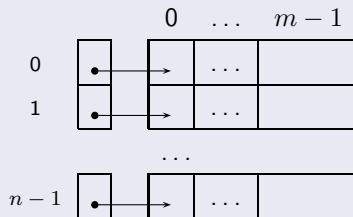
- On peut aussi, allouer un tableau de pointeurs, chaque pointeur pointe sur une ligne de la matrice. La structure de données correspondante est alors :

# C3 Pointeurs, types structurés

## 8. Tableaux multidimensionnels

### $V_2$ : Allocation d'un tableau de pointeurs

- On peut aussi, allouer un tableau de pointeurs, chaque pointeur pointe sur une ligne de la matrice. La structure de données correspondante est alors :

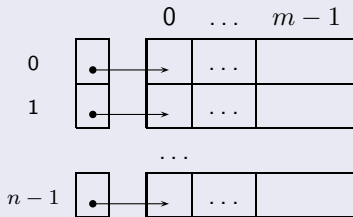


# C3 Pointeurs, types structurés

## 8. Tableaux multidimensionnels

### $V_2$ : Allocation d'un tableau de pointeurs

- On peut aussi, allouer un tableau de pointeurs, chaque pointeur pointe sur une ligne de la matrice. La structure de données correspondante est alors :



- Dans ce cas il faut allouer le tableau de pointeurs :

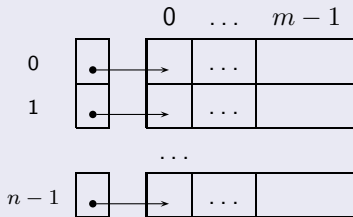
```
int **mat = malloc(sizeof(int*)*n);
```

## C3 Pointeurs, types structurés

### 8. Tableaux multidimensionnels

#### $V_2$ : Allocation d'un tableau de pointeurs

- On peut aussi, allouer un tableau de pointeurs, chaque pointeur pointe sur une ligne de la matrice. La structure de données correspondante est alors :



- Dans ce cas il faut allouer le tableau de pointeurs :  
`int **mat = malloc(sizeof(int*)*n);`
- Puis chacune des lignes de la matrice individuellement :  
`mat[i] = malloc(sizeof(int) * m);` (à faire pour  $0 \leq i \leq n$ )

### Le bilan ...

- Chacune des méthodes a ses avantages et ses inconvénients



### Le bilan ...

- Chacune des méthodes a ses avantages et ses inconvénients
- La seule qui doit être maîtrisée est la méthode  $V_0$  (puisque'on reste en fait dans le cas de la dimension 1 en manipulant soi-même les indices)

### Le bilan ...

- Chacune des méthodes a ses avantages et ses inconvénients
- La seule qui doit être maîtrisée est la méthode  $V_0$  (puisque'on reste en fait dans le cas de la dimension 1 en manipulant soi-même les indices)
- Pour aller un peu plus loin dans l'apprentissage du C, les élèves intéressés peuvent essayer de coder une version utilisant la  $V_1$  ou la  $V_2$ .