

# 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.

# 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

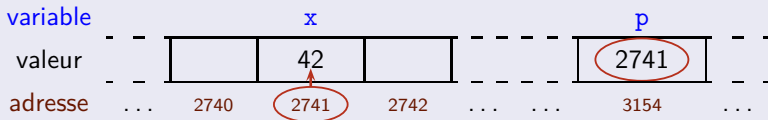
- De façon schématique, la mémoire d'un ordinateur s'apparente à un immense tableau dont chaque case a une adresse.
- Un **pointeur** est une variable contenant une de ces adresses.
- Le schéma ci-dessous représente un pointeur **p** contenant l'adresse d'une variable **x** qui vaut 42.

# 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.

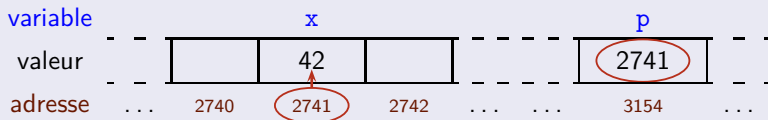


# C3 Pointeurs, types structurés

## 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.



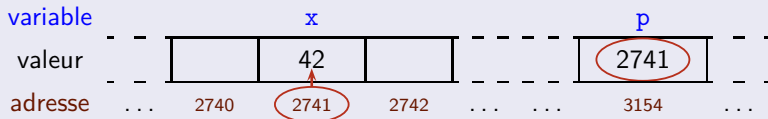
- La valeur particulière **NULL** indique qu'un pointeur ne pointe sur rien.

# 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.



- La valeur particulière **NULL** indique qu'un pointeur ne pointe sur rien.
- En C, la gestion de la mémoire n'est pas totalement automatique (comme en Python ou en OCaml). Certains aspects reviennent au programmeur, ce qui impose de comprendre le modèle mémoire du 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

↑  
-----  
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

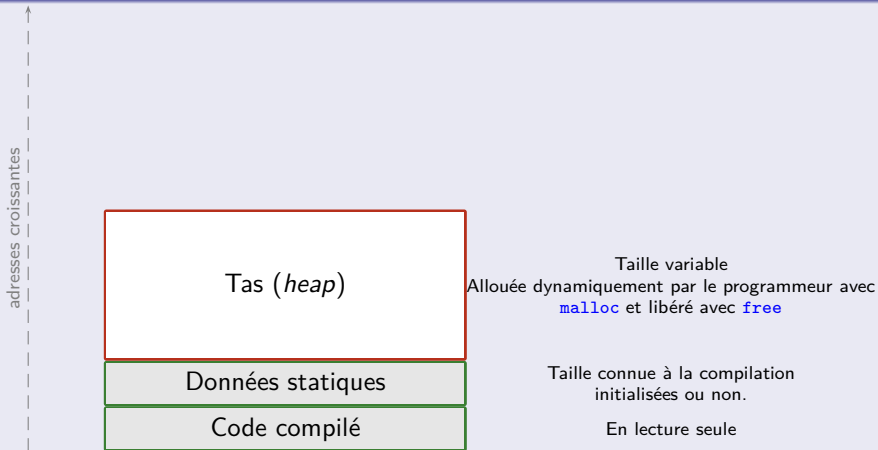
Taille connue à la compilation  
initialisées ou non.

En lecture seule

# C3 Pointeurs, types structurés

## 1. Mémoire en C

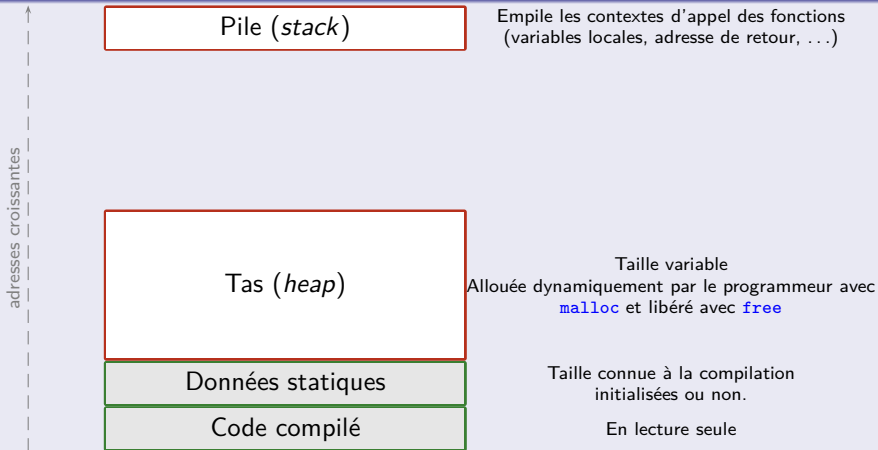
### Schéma de l'organisation de la mémoire en C



# C3 Pointeurs, types structurés

## 1. Mémoire en C

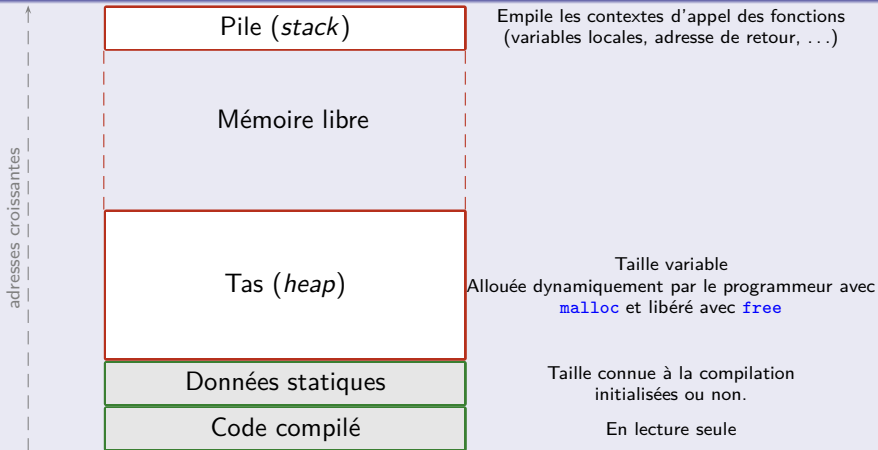
### Schéma de l'organisation de la mémoire en C



# C3 Pointeurs, types structurés

## 1. Mémoire en C

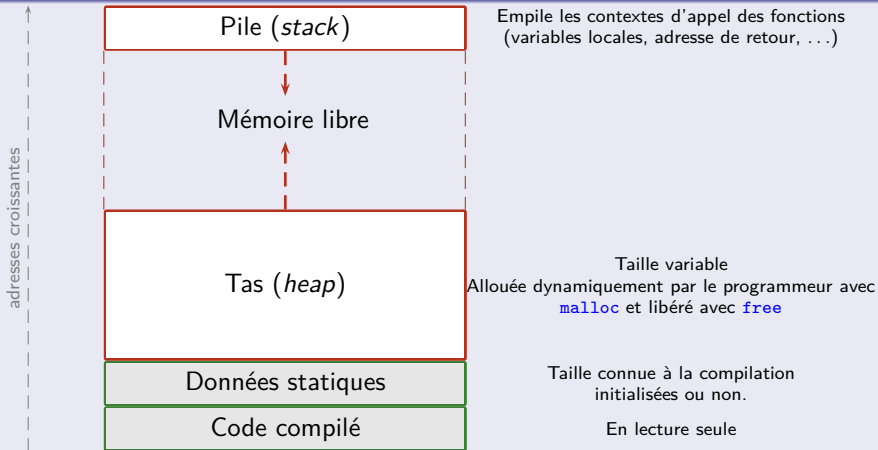
### Schéma de l'organisation de la mémoire en C



# C3 Pointeurs, types structurés

## 1. Mémoire en C

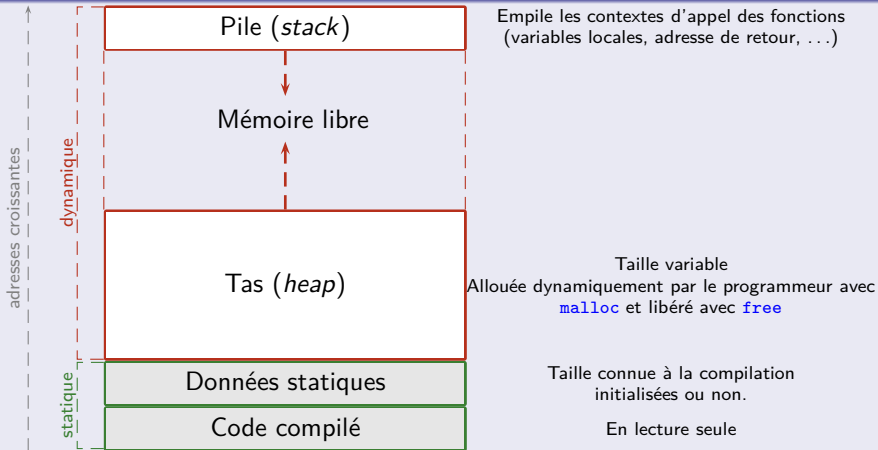
### Schéma de l'organisation de la mémoire en C



# 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 stockés dans la pile. A la fin de l'exécution, ces informations sont supprimés de la pile. Conserver des pointeurs vers des adresses de variables locales est donc problématique.

### 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 stockés dans la pile. A la fin de l'exécution, ces informations sont supprimés de la pile. Conserver des pointeurs vers des adresses de variables locales est donc problématique.
- 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 premier exemple

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

- ❶ Rappeler la taille d'un `double`, en déduire la taille du tableau `big_array`

# C3 Pointeurs, types structurés

## 1. Mémoire en C

### Un premier exemple

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

- 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 ?

# C3 Pointeurs, types structurés

## 1. Mémoire en C

### Un premier exemple

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

- 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 d'appel sur cet ordinateur.

### Un premier exemple

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

- 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 d'appel sur cet ordinateur.
- ❹ Comment résoudre ce problème ?

### Un premier exemple

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

- ❶ Rappeler la taille d'un `double`, en déduire la taille du tableau `big_array`  
Un `double` occupe 8 octets, donc ce tableau  $8 \times 1,5 = 12$  Mo.
- 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 d'appel sur cet ordinateur.
- ❹ Comment résoudre ce problème ?

# C3 Pointeurs, types structurés

## 1. Mémoire en C

### Un premier exemple

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

- ❶ Rappeler la taille d'un `double`, en déduire la taille du tableau `big_array`  
Un `double` occupe 8 octets, donc ce tableau  $8 \times 1,5 = 12$  Mo.
- 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 ?  
La taille du tableau dépasse celle de la pile sur laquelle il est alloué.
- ❸ En déduire une information sur la taille de la pile d'appel sur cet ordinateur.
- ❹ Comment résoudre ce problème ?



# C3 Pointeurs, types structurés

## 1. Mémoire en C

### Un premier exemple

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

- ❶ Rappeler la taille d'un `double`, en déduire la taille du tableau `big_array`  
Un `double` occupe 8 octets, donc ce tableau  $8 \times 1,5 = 12$  Mo.
- 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 ?  
La taille du tableau dépasse celle de la pile sur laquelle il est alloué.
- ❸ En déduire une information sur la taille de la pile d'appel sur cet ordinateur.  
La pile fait moins de 12Mo (sa taille est de l'ordre de 8Mo sur l'ordinateur utilisé)
- ❹ Comment résoudre ce problème ?

# C3 Pointeurs, types structurés

## 1. Mémoire en C

### Un premier exemple

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

- ❶ Rappeler la taille d'un `double`, en déduire la taille du tableau `big_array`  
Un `double` occupe 8 octets, donc ce tableau  $8 \times 1,5 = 12$  Mo.
- 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 ?  
La taille du tableau dépasse celle de la pile sur laquelle il est alloué.
- ❸ En déduire une information sur la taille de la pile d'appel sur cet ordinateur.  
La pile fait moins de 12Mo (sa taille est de l'ordre de 8Mo sur l'ordinateur utilisé)
- ❹ Comment résoudre ce problème ?  
La mémoire occupée par le tableau doit être alloué sur le tas.

### Opérateurs & et \*

### Opérateurs & et \*

- L'opérateur unaire `&`, appelé opérateur d'adresse, permet en C de récupérer l'adresse mémoire d'une variable.

### Opérateurs & et \*

- 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.

### Opérateurs & et \*

- 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
<b>&amp;</b>	à une variable	récupérer son adresse
<b>*</b>	à un pointeur	récupérer la valeur à l'emplacement mémoire désigné

### Opérateurs & et \*

- 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

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

### Opérateurs & et \*

- 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

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

```
int ma_fonction(int *n)
```



### Opérateurs & et \*

- 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

- Si `t` est un type du C, par exemple (`int`, `char`, ...), alors `t *` est du type pointeur vers `t`. Par exemple :  
`int` `ma_fonction`(`int` `*n`)  
est une fonction qui prend en paramètre l'adresse d'un `int`

### Exemple 1

```
1  int n = 42;  
2  int p = &n;  
3  printf("Valeur stockée à l'adresse de p = %d",*p);
```

### Exemple 2

### Exemple 1

```
1  int n = 42;  
2  int p = &n;  
3  printf("Valeur stockée à l'adresse de p = %d",*p);
```

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 n = 42;  
2  int p = &n;  
3  printf("Valeur stockée à l'adresse de p = %d",*p);
```

Dans l'exemple ci-dessus, `p` est un pointeur qui contient l'adresse de `n`. Le `printf` affiche le contenu de l'adresse pointée par `p` donc 42.

### Exemple 2

### Exemple 1

```
1  int n = 42;  
2  int p = &n;  
3  printf("Valeur stockée à l'adresse de p = %d",*p);
```

Dans l'exemple ci-dessus, `p` est un pointeur qui contient l'adresse de `n`. Le `printf` affiche le contenu de l'adresse pointée par `p` donc 42.

### 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 n = 42;  
2  int p = &n;  
3  printf("Valeur stockée à l'adresse de p = %d",*p);
```

Dans l'exemple ci-dessus, `p` est un pointeur qui contient l'adresse de `n`. Le `printf` affiche le contenu de l'adresse pointée par `p` donc 42.

### 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 Ajouter un appel à `echange` dans le programme ci-dessous de façon à échanger les valeurs des variables `a` et `b`.

```
1  int a = 42;  
2  int b = 14;
```

### Correction de l'exemple



### Correction de l'exemple

- ❏ Récupérer les valeurs stockées aux adresses `p1` et `p2`.



### Correction de l'exemple

- ❏ Récupérer les valeurs stockées aux adresses `p1` et `p2`.  
Utiliser une variable temporaire et les échanger

### Correction de l'exemple

- ❌ Récupérer les valeurs stockées aux adresses `p1` et `p2`.  
Utiliser une variable temporaire et les échanger  
Pour l'appel, récupérer les adresses de `a` et `b` afin de les passer en paramètres.

### Correction de l'exemple

❌ Récupérer les valeurs stockées aux adresses `p1` et `p2`.

Utiliser une variable temporaire et les échanger

Pour l'appel, récupérer les adresses de `a` et `b` afin de les passer en paramètres.

```
1 void echange(int *p1, int *p2)
2 {
3     int v1 = *p1;
4     int v2 = *p2;
5     int temp = v1;
6     *p1 = v2;
7     *p2 = temp;
8 }
```

### Correction de l'exemple

❌ Récupérer les valeurs stockées aux adresses `p1` et `p2`.

Utiliser une variable temporaire et les échanger

Pour l'appel, récupérer les adresses de `a` et `b` afin de les passer en paramètres.

```
1 void echange(int *p1, int *p2)
2 {
3     int v1 = *p1;
4     int v2 = *p2;
5     int temp = v1;
6     *p1 = v2;
7     *p2 = temp;
8 }
```

Pour l'appel : `echange(&a,&b);`

# C3 Pointeurs, types structurés

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

### `malloc`

- La fonction `malloc` permet d'allouer sur le tas, un bloc mémoire dont on donne la taille

### Exemple

# C3 Pointeurs, types structurés

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

### `malloc`

- La fonction `malloc` 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.

### Exemple

# C3 Pointeurs, types structurés

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

### `malloc`

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

### Exemple

# C3 Pointeurs, types structurés

## 3. Fonctions malloc et free

### malloc

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

### Exemple

```
1 double *t = malloc(sizeof(int)*100); //alloue le bloc mémoire
2 t[5] = 12; // affecte la valeur 12 au 6eme élément du bloc
3 t[113] = 27; // Comportement indéfini
```



# C3 Pointeurs, types structurés

## 3. Fonctions malloc et free

### free

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

### free

- La fonction `free` permet de libérer un bloc mémoire précédemment alloué grâce à `malloc`
- On appelle donc `free` sur un pointeur créé `p` par `malloc`. Cet appel doit impérativement se faire sur la portée de `p`. En dehors, le bloc mémoire n'est plus libérable

### free

- La fonction `free` permet de libérer un bloc mémoire précédemment alloué grâce à `malloc`
- On appelle donc `free` sur un pointeur créé `p` par `malloc`. Cet appel doit impérativement se faire sur la portée de `p`. En dehors, le bloc mémoire n'est plus libérable
- Des blocs mémoire non libérés (ou non libérables) créent des fuites de mémoire.

# C3 Pointeurs, types structurés

## 3. Fonctions malloc et free

### free

- La fonction `free` permet de libérer un bloc mémoire précédemment alloué grâce à `malloc`
- On appelle donc `free` sur un pointeur créé `p` par `malloc`. Cet appel doit impérativement se faire sur la portée de `p`. En dehors, le bloc mémoire n'est plus libérable
- Des blocs mémoire non libérés (ou non libérables) créent des fuites de mémoire.
- On utilisera toujours l'option `fsanitize = adress` du compilateur pour détecter ses fuites mémoires.

# C3 Pointeurs, types structurés

## 3. Fonctions malloc et free

### Exemple : fonction renvoyant un tableau

- Ecrire une fonction `make_tab` qui prend en argument deux entiers `size` et `init` et renvoie un pointeur vers un tableau de `size` cases initialisés à la valeur `init`.

# C3 Pointeurs, types structurés

## 3. Fonctions malloc et free

### Exemple : fonction renvoyant un tableau

- Ecrire une fonction `make_tab` qui prend en argument deux entiers `size` et `init` et renvoie un pointeur vers un tableau de `size` cases initialisés à la valeur `init`.

```
1 // taille size initialisé avec la valeur init
2 int * make_tab(int size, int init) {
3     int * tab = malloc(sizeof(int)*size);
4     for (int i=0;i<size;i++)
5         { tab[i] = init;}
6     return tab;}
7
```

# C3 Pointeurs, types structurés

## 3. Fonctions malloc et free

### Exemple : fonction renvoyant un tableau

- Ecrire une fonction `make_tab` qui prend en argument deux entiers `size` et `init` et renvoie un pointeur vers un tableau de `size` cases initialisés à la valeur `init`.

```
1 // taille size initialisé avec la valeur init
2 int * make_tab(int size, int init) {
3     int * tab = malloc(sizeof(int)*size);
4     for (int i=0;i<size;i++)
5         { tab[i] = init;}
6     return tab;}
7
```

- ⚠ L'allocation **doit** se faire avec `malloc`, sinon elle est faite sur la pile et donc disparaît lorsqu'on quitte la fonction.

# C3 Pointeurs, types structurés

## 3. Fonctions malloc et free

### Exemple : fonction renvoyant un tableau

- Ecrire une fonction `make_tab` qui prend en argument deux entiers `size` et `init` et renvoie un pointeur vers un tableau de `size` cases initialisés à la valeur `init`.

```
1 // taille size initialisé avec la valeur init
2 int * make_tab(int size, int init) {
3     int * tab = malloc(sizeof(int)*size);
4     for (int i=0;i<size;i++)
5         { tab[i] = init;}
6     return tab;}
7
```

- ⚠ L'allocation **doit** se faire avec `malloc`, sinon elle est faite sur la pile et donc disparaît lorsqu'on quitte la fonction.
- La fonction appelante doit libérer la mémoire allouée avec `free` (sous peine de fuites mémoires.)



## 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`

#### 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      double somme = 0.0;
6      for (int i=1; i<argc; i++) {
7          somme = somme + atof(argv[i]);
8      }
9      printf("%f\n", somme/(argc-1));
10 }
```

### 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
```

# C3 Pointeurs, types structurés

## 5. Types structurés

### Exemple

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

```
1 struct personne_struct {  
2     char nom[50];  
3     poids float;  
4     taille float;  
5 }
```

On peut donner un nom à ce type :

```
typedef struct personne_struct 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  #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;}
```



### Pointeur sur un struct

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

### Exemple

### Pointeur sur un struct

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

### Pointeur sur un struct

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

# C3 Pointeurs, types structurés

## 7. Compilation séparée


### Schéma des étapes d'une compilation séparée

 module.h

 main.c

 module.c

 main.o

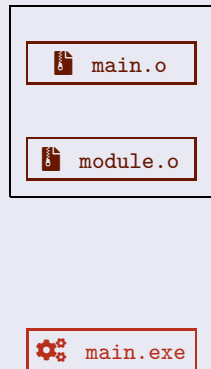
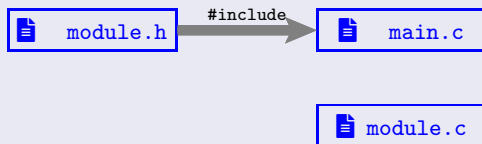
 module.o

 main.exe

# C3 Pointeurs, types structurés

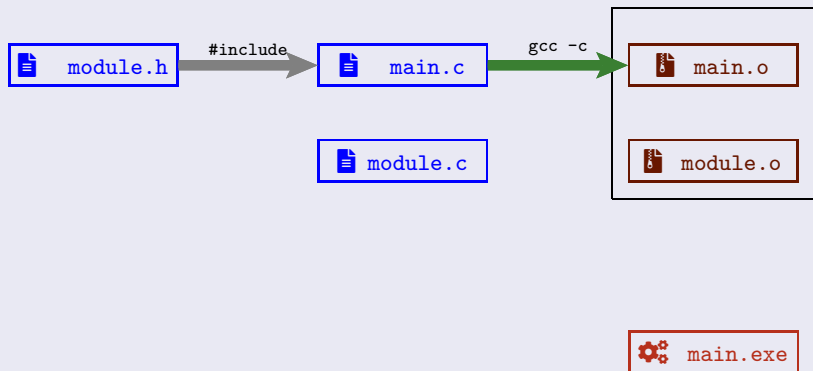
## 7. Compilation séparée

### Schéma des étapes d'une compilation séparée

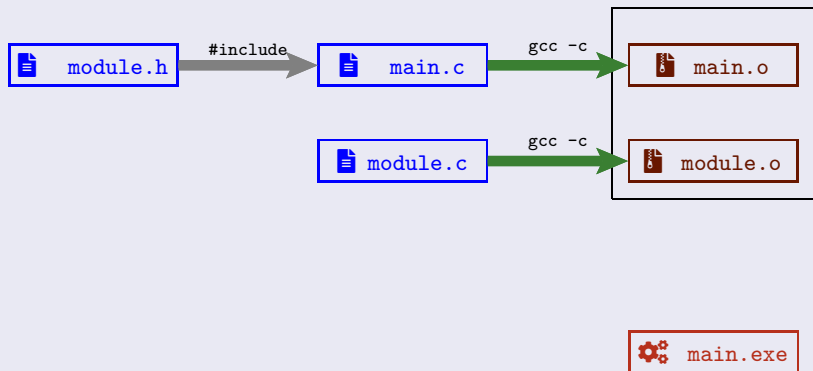




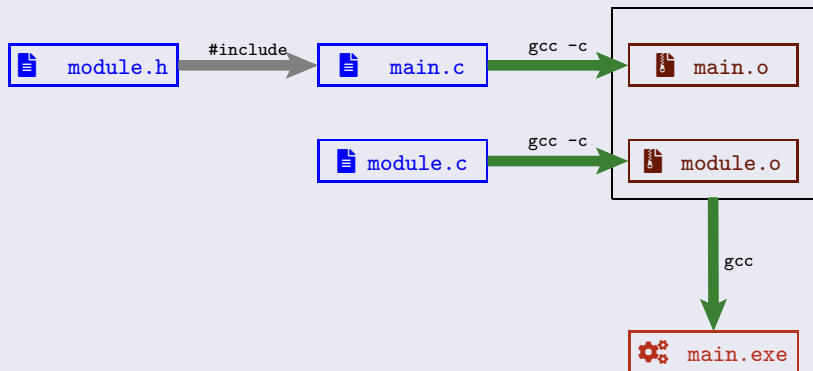
### Schéma des étapes d'une compilation séparée



### 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 définie dans des modules ou dans le programme principale, on peut prefixer 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 inclus 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`