

# Langage C avancé : Séance 5

Représentation mémoire : vecteurs et structures

Position en mémoire des données

Organisation de la mémoire utilisée par le code

---

Marc TAJCHMAN

@-mail : [marc.tajchman@cea.fr](mailto:marc.tajchman@cea.fr)

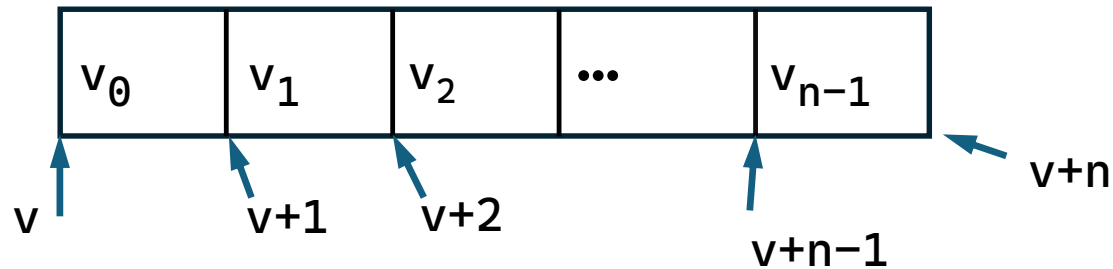
CEA - DES/ISAS/DM2S/STMF/LDEI

Représentation mémoire (suite)

## Représentation des vecteurs

Un vecteur est un ensemble de  $n$  données **du même type** (composantes), on accède à une composante par son indice (un entier de  $0$  à  $n-1$ ).

Les composantes d'un vecteur sont les unes à côté des autres.  
Une variable  $v$  de type vecteur est un pointeur contenant l'adresse du début de la première composante.



Pour accéder à une composante d'indice  $i$  ( $0 \leq i < n$ ), on utilise le pointeur  $v + i$  (qui contient l'adresse du début de la composante) :

$$*(v + i) = 3.5 \text{ ou } v[i] = 3.5$$

## Représentation des vecteurs

L'accès aux composantes du vecteur se fait en général très souvent et dans un ordre quelconque a priori, il faut donc que l'accès soit le plus rapide possible.

Ceci explique pourquoi, **les composantes d'un vecteur doivent être du même type**: l'adresse de la composante  $i$  du vecteur  $v$  est alors l'adresse du début du vecteur +  $i \times$  le nombre d'octets occupés par une valeur du type des composantes

Il est inutile (et déconseillé) de calculer soi-même l'adresse d'une composante, il suffit d'utiliser l'une des 2 expressions, par exemple pour accéder à la composante d'indice 3 du vecteur de doubles  $v$ :

```
v[3] = 1.5;  
*(v + 3) = 1.5;
```

# Structures

Une structure est constituée d'un ensemble **fixe** de composantes de types différents ou non.

Au contraire des types de base, une structure est un type défini par le programmeur.

Le mot clef `struct` doit être utilisé pour définir un type structure et pour déclarer des variables de ce type.

Exemple de structure :

```
struct S {  
    int n;  
    double X;  
};  
  
struct S v;
```

On peut aussi définir un type utilisateur (un alias) :

```
struct S {  
    int n;  
    double X;  
};  
  
typedef struct S Vecteur;  
Vecteur v;
```

# Structures

Quand on définit une structure, une zone mémoire suffisante est réservée pour contenir toutes les composantes.

La taille de cette zone est donnée par

- `sizeof(a)` si `a` est une variable de type structure
- `sizeof(struct S)` si `S` est un type structure
- `sizeof(T)` si `T` est un type structure (défini par `typedef`)

L'accès aux composantes se fait par la notation :

`nom_variable.nom_composante`

```
struct S {  
    int n;  
    double X;  
};
```

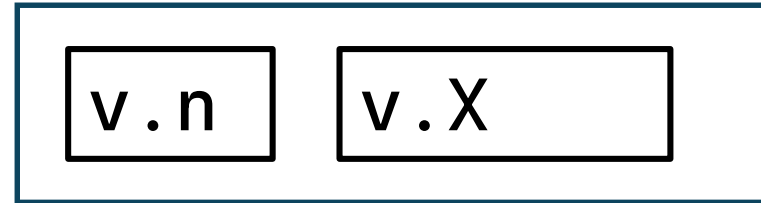
```
struct S v;  
v.n = 3;  
v.X = 1.5;
```

## Représentation mémoire d'une structure

Les composantes d'une structure sont placées dans une zone mémoire de taille suffisante (il peut y avoir des espaces vides entre les composantes: plus de précisions dans la suite)

```
struct S {  
    int n;  
    double X;  
};  
struct S v;
```

v



Contraintes sur les adresses mémoires

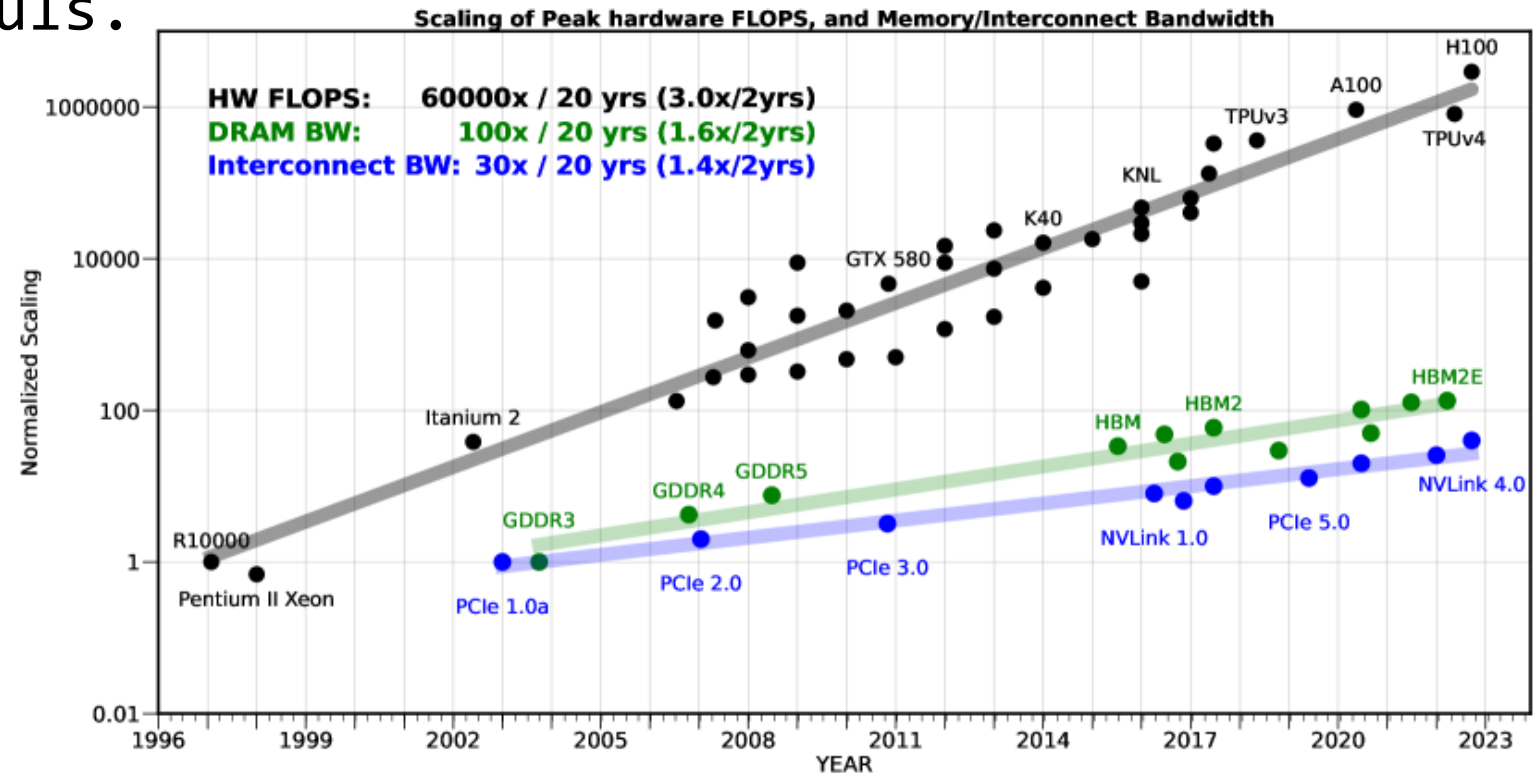


# Transfert de données entre la mémoire et le processeur

Un ordinateur comporte de la mémoire connectée à un (ou plusieurs) processeur(s).

Ces dernières années, la vitesse des processeurs à augmenté beaucoup plus vite que celle de la mémoire.

On dit parfois que les processeurs passent plus de temps à attendre la réception des données ou l'envoi des résultats, qu'à effectuer des calculs.



## Transfert de données entre la mémoire et le processeur

Pour essayer d'améliorer la situation, à part d'améliorer le matériel, on essaie de diminuer le nombre de transferts par rapport au nombre de calculs:

- On a plusieurs niveaux de mémoire intermédiaires entre la mémoire de travail et le processeur (de plus en plus rapide quand on est « s'approche » du processeur), **on parle de mémoire cache**

*Dans ce cas, le processeur ne communique pas directement avec la mémoire de travail, mais avec une mémoire plus petite mais plus rapide : une mémoire cache*

- On fait des transferts par **bloc d'octets** (appelés **lignes de cache**) plutôt que par octets individuels

*Souvent, on fait des suites d'instructions de calcul qui utilisent des données proches, par exemple*

```
for (i=0; i<n; i++) v[i] = ...
```

## Transfert de données entre la mémoire et le processeur

Si on exécute une instruction qui a besoin de données très proches de celles utilisées par l'instruction précédente, les données ont peut-être déjà été transférées, ce qui accélère le traitement de l'instruction

*Voir fichier MemoireCache.pdf*

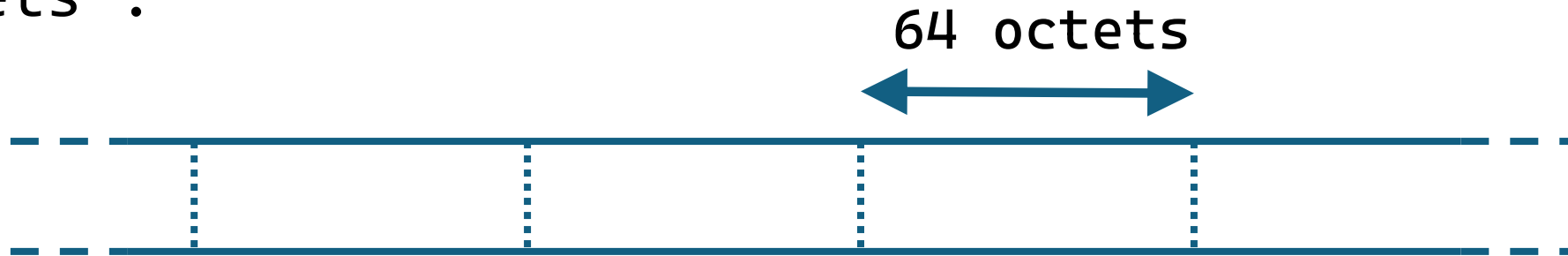
## Transferts par bloc

Le processeur utilise la mémoire cache, il faut donc faire des transferts mémoire de travail – mémoire cache, dans les 2 sens.

Les transferts mémoire  $\leftrightarrow$  mémoire cache se font par blocs de la taille d'une ligne de cache.

Supposons qu'une ligne de cache a une taille de 64 octets (c'est en général le cas).

Le système considère la mémoire comme l'union de blocs de 64 octets :



## Choix par le système des adresses mémoire

Quand on définit une variable, le système réserve une zone mémoire en essayant

- quand le processeur a besoin de la donnée, d'effectuer le moins possible de transferts de blocs de données
- si la variable est de type structure ou de type vecteur, que l'accès à chaque composante soit optimisé de la même façon

Chaque compilateur décide quel est le critère à appliquer pour choisir une adresse

## Exemple : vecteur

Considérons une chaîne de 100 caractères.  
Sans contraintes, il y a plusieurs choix pour la position de la chaîne, entre autres:

Placement 1



Placement 2



Placement 3



← 101 octets →

Pour que le processeur utilise la variable, il faudra :  
3 transferts de blocs de 64 octets (placement 1), 2 transferts (placement 2) ou 2 transferts (placement 3)

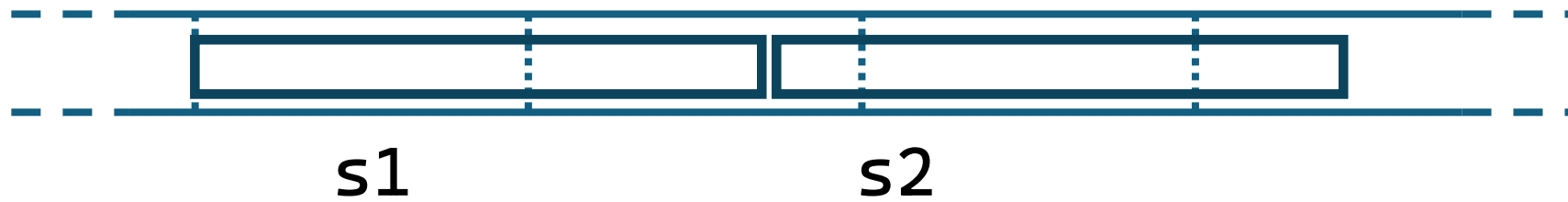
## Exemple: vecteur

Donc le compilateur ne choisira pas un placement de type 1.

En fonction des variables précédemment déclarées, il pourra choisir un placement 2 ou 3, ou tout autre placement qui n'a besoin que de 2 transferts.

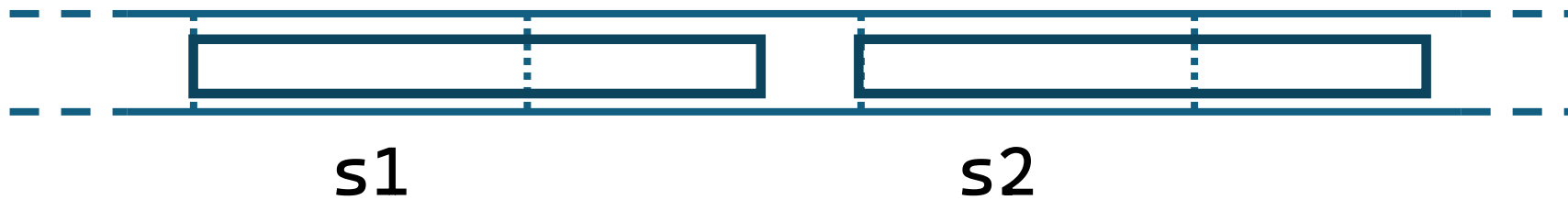
## Exemple: vecteurs

Le code utilise 2 chaînes s1 et s2 de 100 caractères, si les 2 chaînes sont mises côte l'une de l'autre:



Le transfert de s1 se fera en 2 blocs et le transfert de s2 en 3 blocs.

Donc les compilateurs écartent s1 et s2:



Pour que les transferts de s1 et s2 nécessitent chacun 2 blocs.



## Exemple : structure

Une structure peut avoir des composantes de différents type et taille.

De plus, certaines instructions travailleront avec une partie seulement des composantes.

```
struct S {  
    int n;  
    double X;  
};  
struct S v;
```

```
v.X = 1.4;
```

Il faut donc que chaque composante de la structure demande le moins de transferts possibles

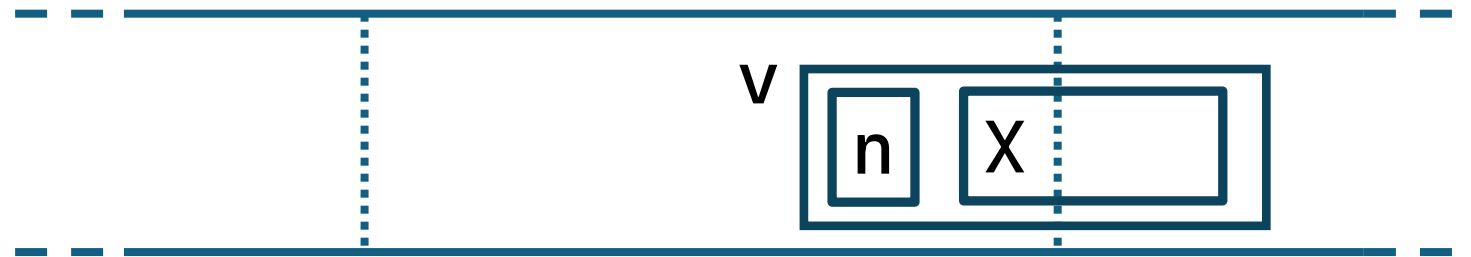
## Exemple : structure

```
struct S {  
    int n;  
    double X;  
};  
struct S v;
```

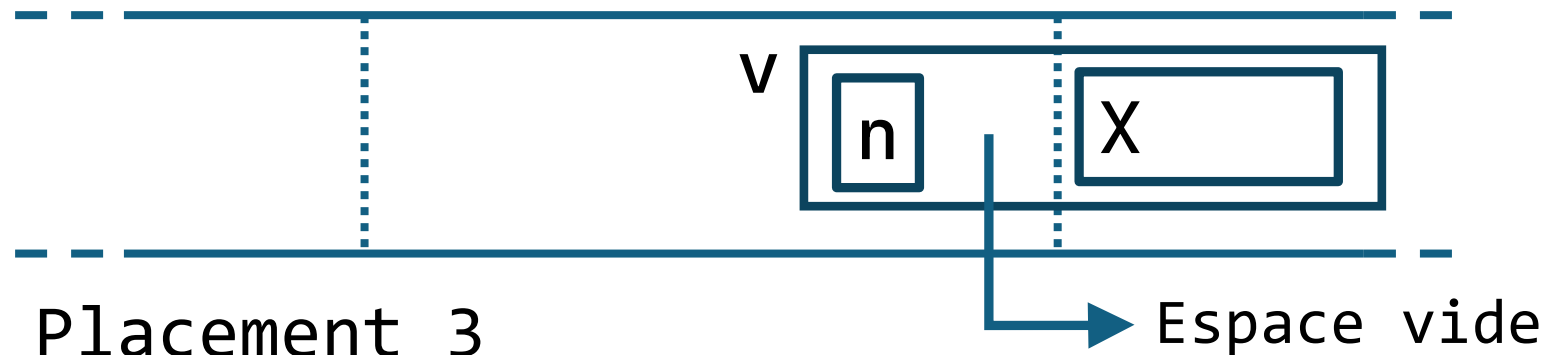
**`v.X = 1.4;`**

Parmi les possibilités:

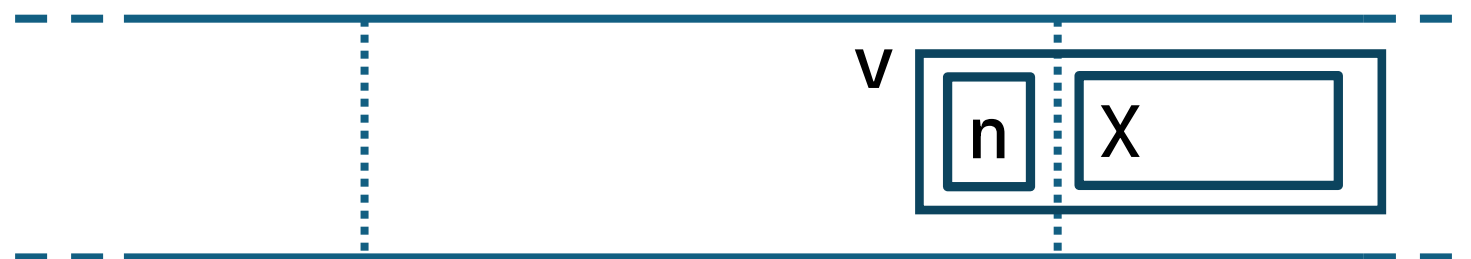
Placement 1



Placement 2

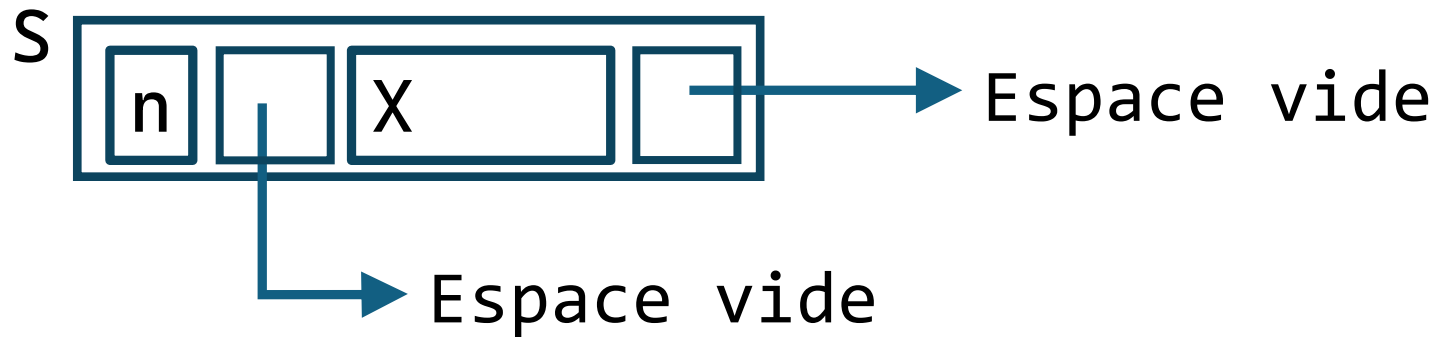


Placement 3



## Placement des structures

Pour pouvoir s'adapter à tous les cas (structures, structures de vecteurs, vecteurs de structures), la plupart des compilateurs introduisent, **si c'est utile**, des espaces mémoire non utilisés à l'intérieur et à la fin des structures.



La technique d'insérer des espaces mémoire inutilisés vus dans les pages précédentes est appelée **padding**

Elle a un inconvénient et un avantage:

- Avantage : la vitesse d'accès aux données dans la mémoire de travail est (nettement) meilleure
- Inconvénient : les espaces mémoire inutilisés occupent de la place qui n'est plus disponible pour les données

Tous les systèmes actuels utilisent cette technique (l'avantage est plus important)

## Padding

On peut essayer d'aider le compilateur à utiliser le moins possible du padding.

La règle souvent suivie est d'ordonner les déclarations de variable et les composantes de structure : on déclare les variables/composantes par taille décroissante.

Par exemple, au lieu de :

On écrira:

```
struct S {  
    int n;  
    char c;  
    double X;  
};
```

```
struct S {  
    double X;  
    int n;  
    char c;  
};
```

Organisation de la mémoire utilisée par un code

## Zones mémoire utilisées par un code

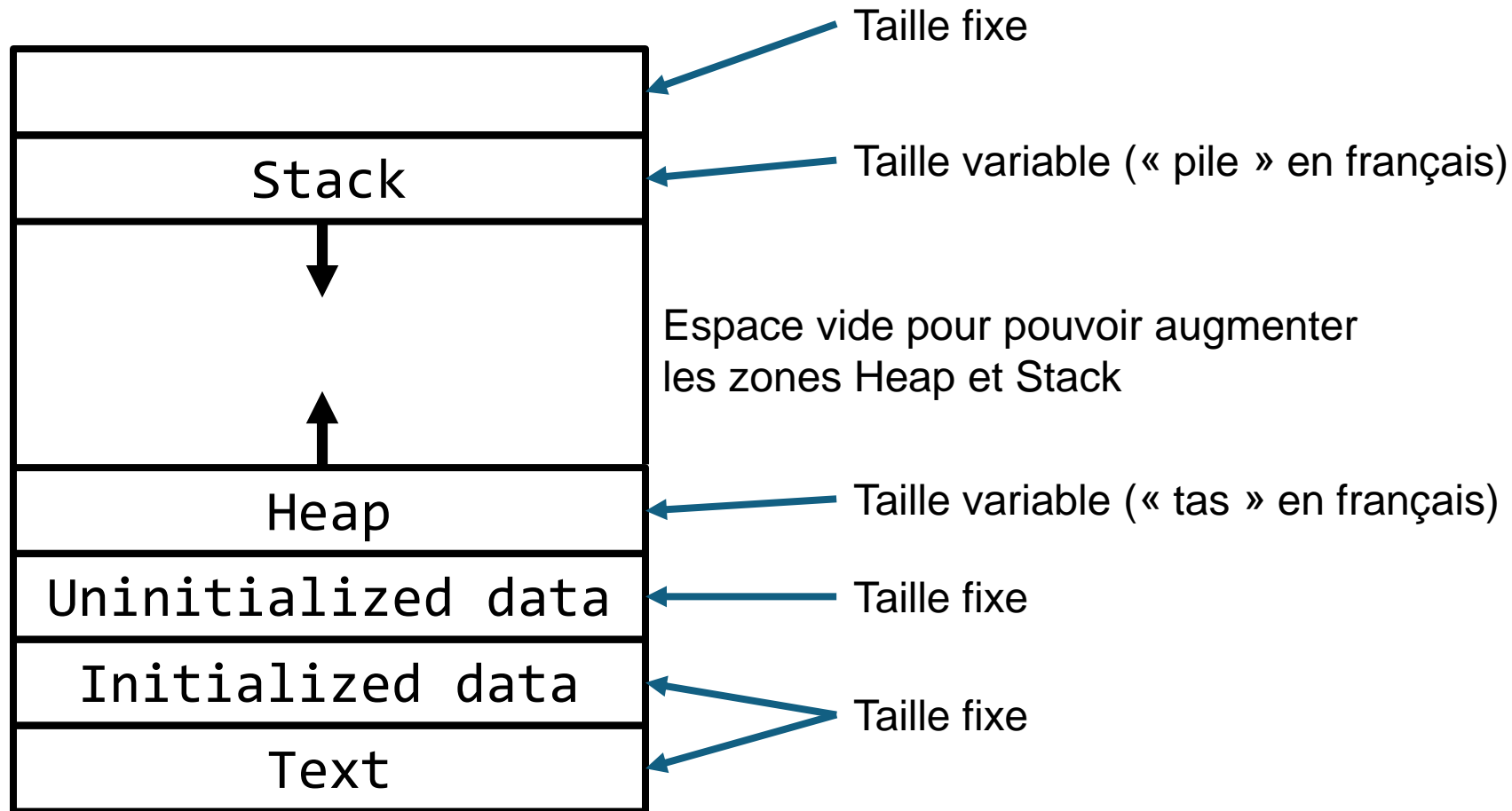
Cette partie n'est pas spécifique au langage C.

Les codes écrits en d'autres langages, en général compilés (C++, java, etc.) suivent la même organisation de la mémoire

La mémoire utilisée est répartie en plusieurs zones (on parle souvent de segments de mémoire)

## Zones mémoire utilisées par un code

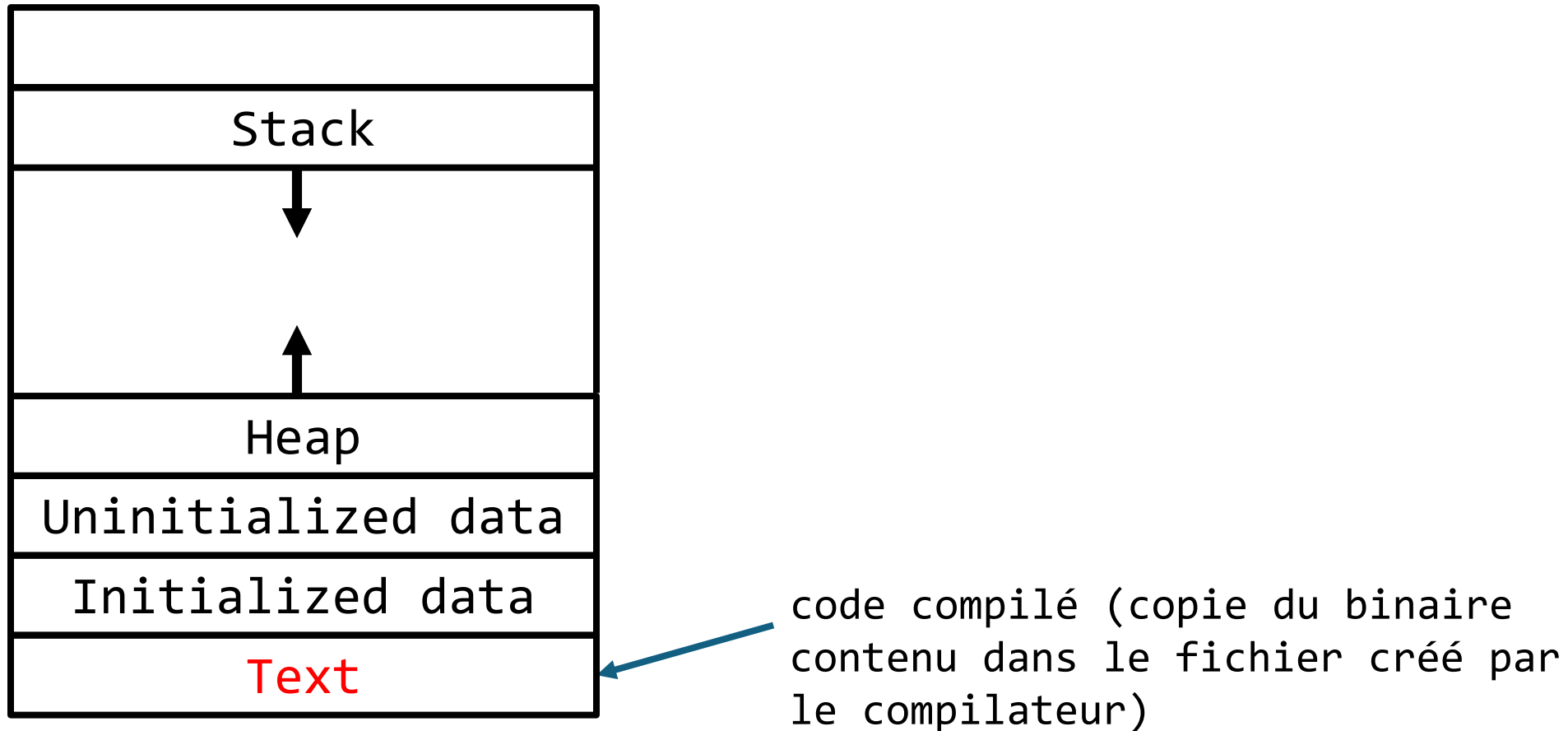
La mémoire utilisée par un code C, en général, est constituée de 6 parties :





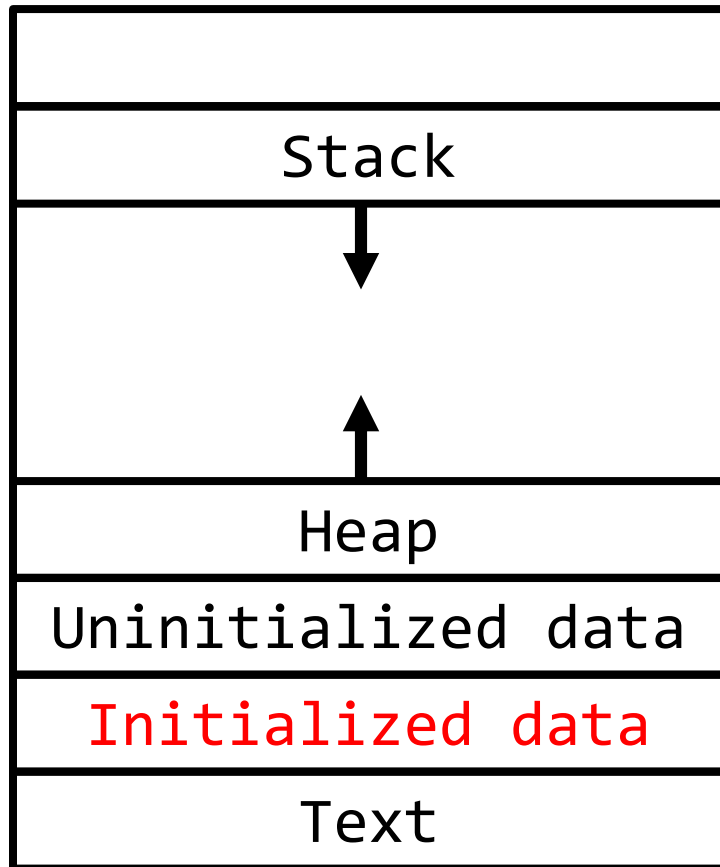
## Zones mémoire utilisées par un code

La mémoire utilisée par un code, en général, est constituée de 6 parties :



## Zones mémoire utilisées par un code

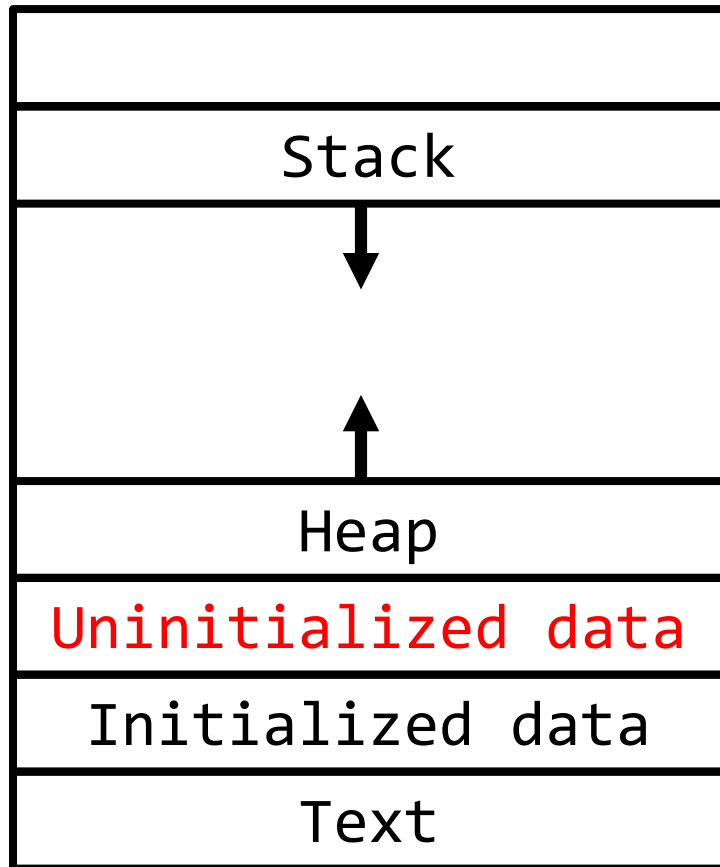
La mémoire utilisée par un code, en général, est constituée de 6 parties :



variables globales et locales statiques  
dans les fonctions (initialisées),  
contenues dans le fichier binaire

## Zones mémoire utilisées par un code

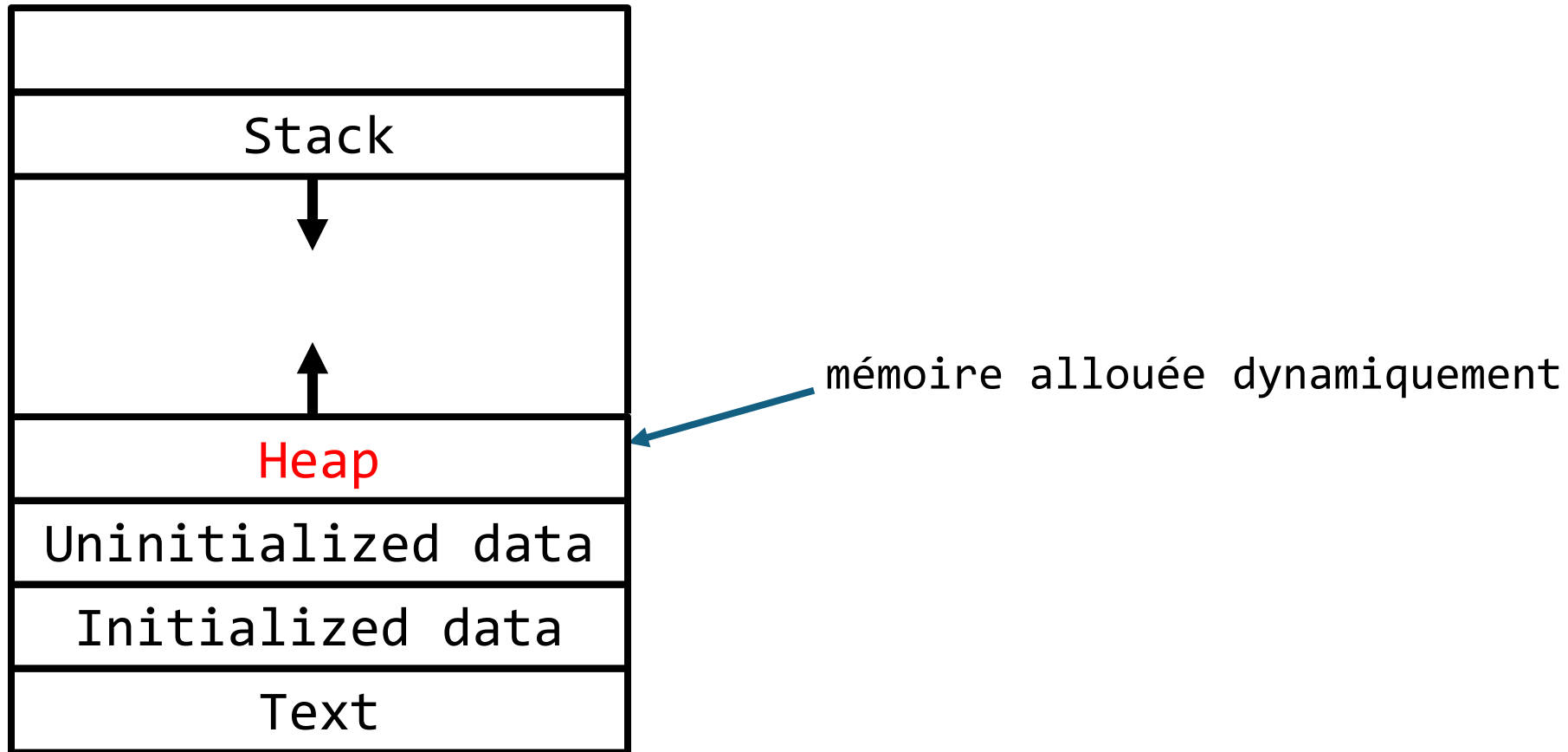
La mémoire utilisée par un code, en général, est constituée de 6 parties :



variables globales et locales statiques  
dans les fonctions (non initialisées),  
initialisation à 0 au démarrage de  
l'exécution

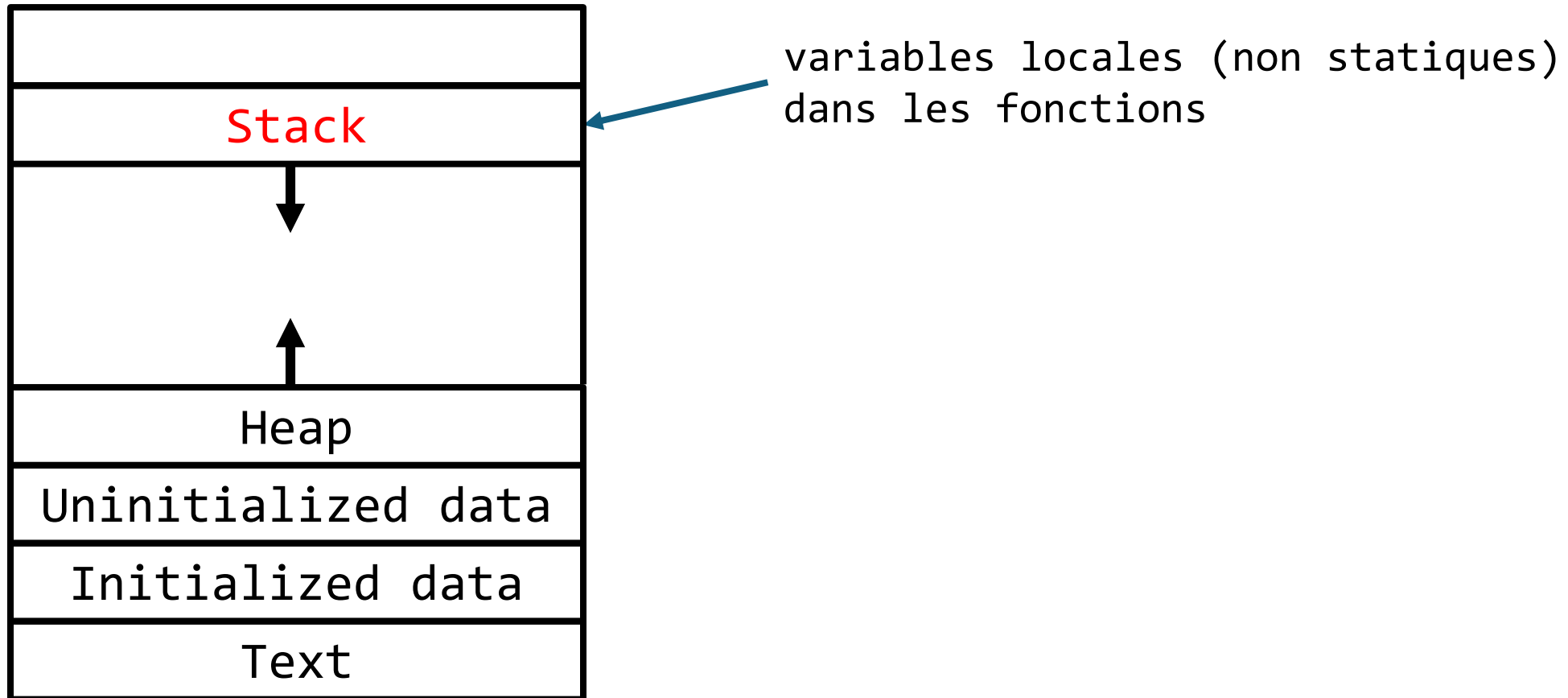
## Zones mémoire utilisées par un code

La mémoire utilisée par un code, en général, est constituée de 6 parties :



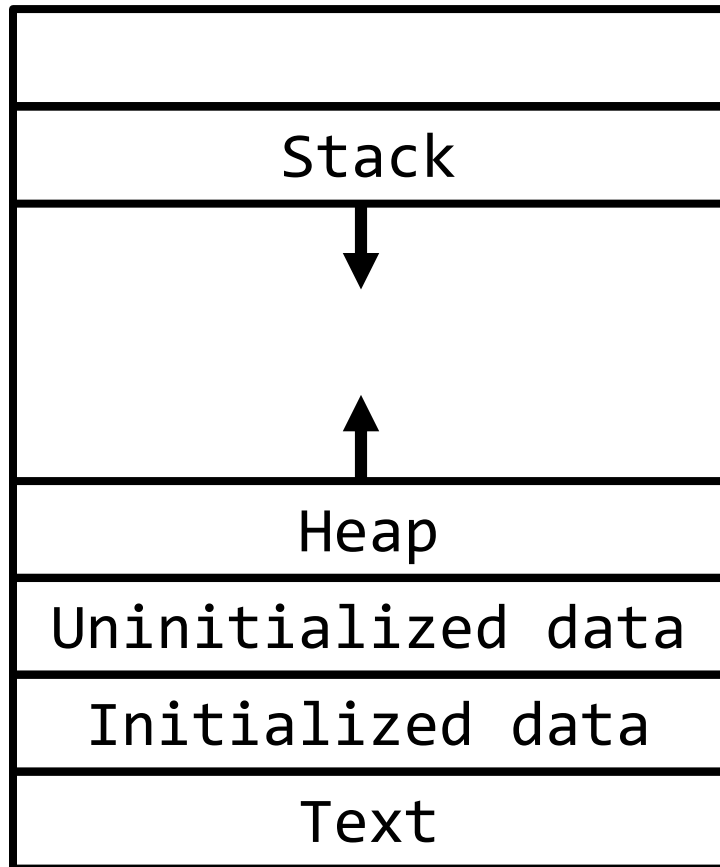
## Zones mémoire utilisées par un code

La mémoire utilisée par un code, en général, est constituée de 6 parties :



## Zones mémoire utilisées par un code

La mémoire utilisée par un code, en général, est constituée de 6 parties :



Arguments de la ligne de commande, valeur de retour du code (un entier) mise dans la variable d'environnement `$?` (linux) et autres variables d'environnement

## Zones mémoire « Text »

« Text » contient les instructions binaires produites par le compilateur.

Cette zone mémoire est copiée du fichier binaire dans la mémoire de travail de l'ordinateur.

Son contenu est fixe pendant l'exécution du code (autrement dit le code ne change pas pendant l'exécution)

## Zones mémoire « Initialized data »

Cette zone mémoire est copiée du fichier binaire dans la mémoire de travail de l'ordinateur.

Elle contient (une partie) des **variables globales** et (une partie) des **variables statiques locales**.

Leur valeur initiale est contenue dans le fichier binaire mais peut changer au cours de l'exécution.



## Zone « Initialized data » : Variable globale initialisée

`v` est une variable entière **globale** dont la valeur initiale (contenue dans le fichier) est 5 mais qui peut être modifiée pendant l'exécution:

```
int v = 5;

int main()
{
    v = v + 1;
    return 0;
}
```

## Zone « Initialized data » : Variable locale statique initialisée

`v` est une **variable locale statique** entière dont la valeur initiale est contenue dans le fichier binaire.

Elle peut être modifiée seulement dans la fonction **run** et garde sa valeur entre 2 appels de la fonction **run**.

```
int run()
{
    static int v = 1
    v = v + 1;
    return v;
}
```

```
int main() {
    int w;
    w = run();
    w = run();
    print("w = %d\n", w)
    return 0;
}
```

Question: quelle est la valeur affichée par le programme principal ?

## Zone mémoire « Uninitialized data »

Cette zone est de taille constante.

La différence avec la zone précédente est que le fichier binaire ne contient pas la valeur initiale des variables de cette zone.

Ces variables sont initialisées à 0 au début de l'exécution (avant de commencer le programme principal).


Pendant l'exécution, le code peut modifier leur valeur.

## Zone mémoire « Stack »

Cette zone est de taille variable et est utilisée pour les données locales des fonctions y compris celle de la fonction main (le programme principal).

## Registre d'instruction

Quand on exécute un code C, par exemple:



```
int f(int i, double x)
{
    double y = 2*x;
    int j = i+1;
    return y*j;
}

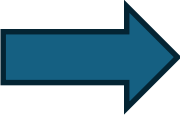
int main()
{
    double a, b;
    a = 1.5;
    b = f(2, a)
    return 0;
}
```

L'exécution du code démarre en créant un **registre d'instruction**, pointeur positionné à l'adresse du début du code de la fonction main dans la zone mémoire « text ».

Chaque fois qu'une instruction du code est exécutée, le système met l'adresse du binaire de l'instruction suivante dans le registre.

L'instruction pointée par le registre est appelée « **instruction courante** ».

## Evolution de la « stack » pendant l'exécution



```
int f(int i, double x)
{
    double y = 2*x;
    int j = i+1;
    return y*j;
}
int main()
{
    double a, b;
    a = 1.5;
    b = f(2, a);
    return 0;
}
```


### Stack

main:  
:



Résultat de la fonction  
main (variable locale  
entière sans nom et  
sans valeur)

## Evolution de la « stack » pendant l'exécution



```
int f(int i, double x)
{
    double y = 2*x;
    int j = i+1;
    return y*j;
}
int main()
{
    double a, b;
    a = 1.5;
    b = f(2, a)
    return 0;
}
```

Stack

main:


:

a:

b:



## Evolution de la « stack » pendant l'exécution



```
int f(int i, double x)
{
    double y = 2*x;
    int j = i+1;
    return y*j;
}
int main()
{
    double a, b;
    a = 1.5;
    b = f(2, a)
    return 0;
}
```

### Stack

main:

:

a:

b:

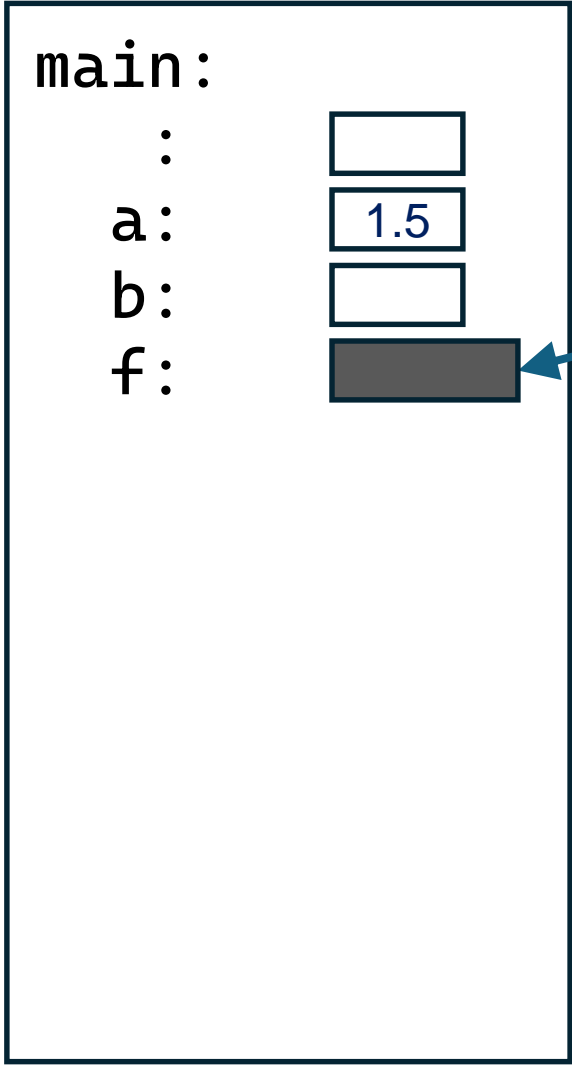
1.5



# Evolution de la « stack » pendant l'exécution


```
int f(int i, double x)
{
    double y = 2*x;
    int j = i+1;
    return y*j;
}
int main()
{
    double a, b;
    a = 1.5;
    b = f(2, a);
    return 0;
}
```

## Stack



Pointeur :  
adresse de l'instruction  
courante (qui appelle f)

## Evolution de la « stack » pendant l'exécution



```
int f(int i, double x)
{
    double y = 2*x;
    int j = i+1;
    return y*j;
}
int main()
{
    double a, b;
    a = 1.5;
    b = f(2, a)
    return 0;
}
```

### Stack

main:

:

a: 1.5

b:


f:

:

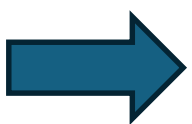
i: 2

x: 1.5

Résultat de la fonction f  
(variable locale entière  
sans nom et sans valeur)



## Evolution de la « stack » pendant l'exécution



```
int f(int i, double x)
{
    double y = 2*x;
    int j = i+1;
    return y*j;
}
int main()
{
    double a, b;
    a = 1.5;
    b = f(2, a)
    return 0;
}
```

### Stack

main:

:

a: 1.5

b:

f:


:

i: 2

x: 1.5

y: 3.0

## Evolution de la « stack » pendant l'exécution




```
int f(int i, double x)
{
    double y = 2*x;
    int j = i+1;
    return y*j;
}
int main()
{
    double a, b;
    a = 1.5;
    b = f(2, a);
    return 0;
}
```

### Stack

main:

:	<input type="text"/>
a:	<input type="text" value="1.5"/>
b:	<input type="text"/>
f:	<input type="text"/>
:	<input type="text"/>
i:	<input type="text" value="2"/>
x:	<input type="text" value="1.5"/>
y:	<input type="text" value="3.0"/>
j:	<input type="text" value="3"/>

## Evolution de la « stack » pendant l'exécution




```
int f(int i, double x)
{
    double y = 2*x;
    int j = i+1;
    return y*j;
}
int main()
{
    double a, b;
    a = 1.5;
    b = f(2, a);
    return 0;
}
```

### Stack

main:

:	<input type="text"/>
a:	<input type="text" value="1.5"/>
b:	<input type="text"/>
f:	<input type="text"/>
:	<input type="text" value="9.0"/>
i:	<input type="text" value="2"/>
x:	<input type="text" value="1.5"/>
y:	<input type="text" value="3.0"/>
j:	<input type="text" value="3"/>

## Evolution de la « stack » pendant l'exécution



```
int f(int i, double x)
{
    double y = 2*x;
    int j = i+1;
    return y*j;
}
int main()
{
    double a, b;
    a = 1.5;
    b = f(2, a);
    return 0;
}
```

### Stack

main:

:

a: 1.5


b:

f:

: 9.0

Les variables locales  
de f sont détruites ainsi  
que les copies locales  
des paramètres de f

## Evolution de la « stack » pendant l'exécution



```
int f(int i, double x)
{
    double y = 2*x;
    int j = i+1;
    return y*j;
}
int main()
{
    double a, b;
    a = 1.5;
    b = f(2, a);
    return 0;
}
```

### Stack

main:

:	
a:	1.5
b:	9.0


Le résultat de f est  
copié dans b

Le reste des données  
de f sont détruites:

- variable résultat
- adresse de retour

## Evolution de la « stack » pendant l'exécution

```
int f(int i, double x)
{
    double y = 2*x;
    int j = i+1;
    return y*j;
}
int main()
{
    double a, b;
    a = 1.5;
    b = f(2, a);
    return 0;
}
```



### Stack

main:

:	<div>0</div>
a:	<div>1.5</div>
b:	<div>9.0</div>

Le résultat de f est  
copié dans b


Le reste des données  
de f sont détruites:

- variable résultat
- adresse de retour



## Evolution de la « stack » pendant l'exécution

```
int f(int i, double x)
{
    double y = 2*x;
    int j = i+1;
    return y*j;
}
int main()
{
    double a, b;
    a = 1.5;
    b = f(2, a);
    return 0;
}
```



### Stack

main:

:

0

La variables locales de  
main sont détruites

## Zone mémoire « Heap »

Cette zone est de taille variable et contient les zones mémoire allouées par malloc.

## Evolution de « Heap » pendant l'exécution

On considère le code suivant qui utilise de la mémoire dynamique:


```
int main()
{
    double *v1, *v2, *v3;
    v1 = (double *) malloc(10*sizeof(double));
    v2 = (double *) malloc(10*sizeof(double));

    /* ... */

    free(v1);
    v3 = (double *) malloc(50*sizeof(double));
    free(v2);
    free(v3);
    return 0;
}
```

# Evolution de « Heap » pendant l'exécution

Text (code)



```
int main()
{
    double *v1, *v2, *v3;
    v1 = (double *) malloc(10*sizeof(double));
    v2 = (double *) malloc(10*sizeof(double));

    /* ... */

    free(v1);
    v3 = (double *) malloc(50*sizeof(double));
    free(v2);
    free(v3);
    return 0;
}
```

Stack


v1:   
v2:   
v3:

Heap



# Evolution de « Heap » pendant l'exécution

Text (code)






```
int main()
{
    double *v1, *v2, *v3;
    v1 = (double *) malloc(10*sizeof(double));
    v2 = (double *) malloc(10*sizeof(double));

    /* ... */

    free(v1);
    v3 = (double *) malloc(50*sizeof(double));
    free(v2);
    free(v3);
    return 0;
}
```

Stack


v1:   
v2:   
v3: 

Heap



# Evolution de « Heap » pendant l'exécution

Text (code)






```
int main()
{
    double *v1, *v2, *v3;
    v1 = (double *) malloc(10*sizeof(double));
    v2 = (double *) malloc(10*sizeof(double));

    /* ... */

    free(v1);
    v3 = (double *) malloc(50*sizeof(double));
    free(v2);
    free(v3);
    return 0;
}
```

Stack

v1:   
v2:   
v3: 

Heap



# Evolution de « Heap » pendant l'exécution




Text (code)

```
int main()
{
    double *v1, *v2, *v3;
    v1 = (double *) malloc(10*sizeof(double));
    v2 = (double *) malloc(10*sizeof(double));

    /* ... */

    free(v1);
    v3 = (double *) malloc(50*sizeof(double));
    free(v2);
    free(v3);
    return 0;
}
```

Stack

v1:   
v2:   
v3: 

Heap



# Evolution de « Heap » pendant l'exécution




Text (code)

```
int main()
{
    double *v1, *v2, *v3;
    v1 = (double *) malloc(10*sizeof(double));
    v2 = (double *) malloc(10*sizeof(double));

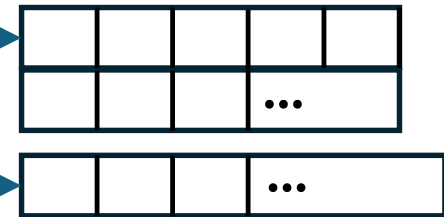
    /* ... */

    free(v1);
    v3 = (double *) malloc(50*sizeof(double));
    free(v2);
    free(v3);
    return 0;
}
```

Stack

v1:   
v2:   
v3: 

Heap



« Trou » dans la Heap



# Evolution de « Heap » pendant l'exécution




Text (code)

```
int main()
{
    double *v1, *v2, *v3;
    v1 = (double *) malloc(10*sizeof(double));
    v2 = (double *) malloc(10*sizeof(double));

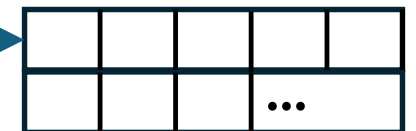
    /* ... */

    free(v1);
    v3 = (double *) malloc(50*sizeof(double));
    free(v2);
    free(v3);
    return 0;
}
```

Stack

v1:   
v2:   
v3: 

Heap



« Trou »  
dans la Heap

# Evolution de « Heap » pendant l'exécution

Text (code)

```
int main()
{
    double *v1, *v2, *v3;
    v1 = (double *) malloc(10*sizeof(double));
    v2 = (double *) malloc(10*sizeof(double));

    /* ... */

    free(v1);
    v3 = (double *) malloc(50*sizeof(double));
    free(v2);
    free(v3);
    return 0;
}
```

Stack

v1:   
v2:   
v3:

Heap



## Morcellement de la « Heap »

On voit que au cours de l'exécution, quand des zones mémoire dynamiques sont libérées, cela peut aboutir à avoir de la mémoire disponible en plusieurs parties disjointes.

On appelle ce phénomène la **fragmentation de la mémoire**.

Tant qu'il reste une zone disponible et en un seul morceau, assez grande pour les allocations mémoire, cela ne pose pas de problème.

Sinon cela empêche de traiter des problèmes de grande taille même si la mémoire totale est théoriquement suffisante.

**Pour éviter, si possible cette situation, il faut ordonner les allocations mémoire.**

## Morcellement de la « Heap »

Par exemple, le bloc d'instructions à gauche peut donner de la fragmentation de la mémoire, et celui de droite non

```
v1 = (int *) malloc(N*sizeof(int));  
v2 = (int *) malloc(N*sizeof(int));  
  
free(v1);  
  
/* Ici, la mémoire est fragmentée */  
  
free(v2);
```

```
v1 = (int *) malloc(N*sizeof(int));  
v2 = (int *) malloc(N*sizeof(int));  
  
free(v2);  
  
/* Ici, pas de fragmentation */  
  
free(v1);
```