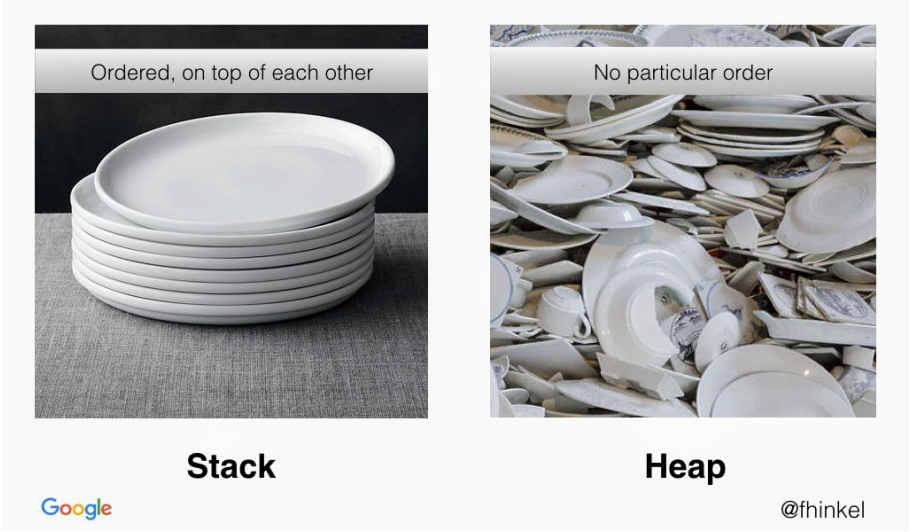
Programmation algorithmique

Leçon 2
La pile et le tas



Rébecca Lapointe

29 octobre 2018

Quelques questions...

Comment un exécutable fait-il pour savoir l'ordre d'exécution des instructions?

Comment un exécutable fait-il pour savoir quoi exécuter après un `return`?

Comment la mémoire est-elle allouée et libérée?

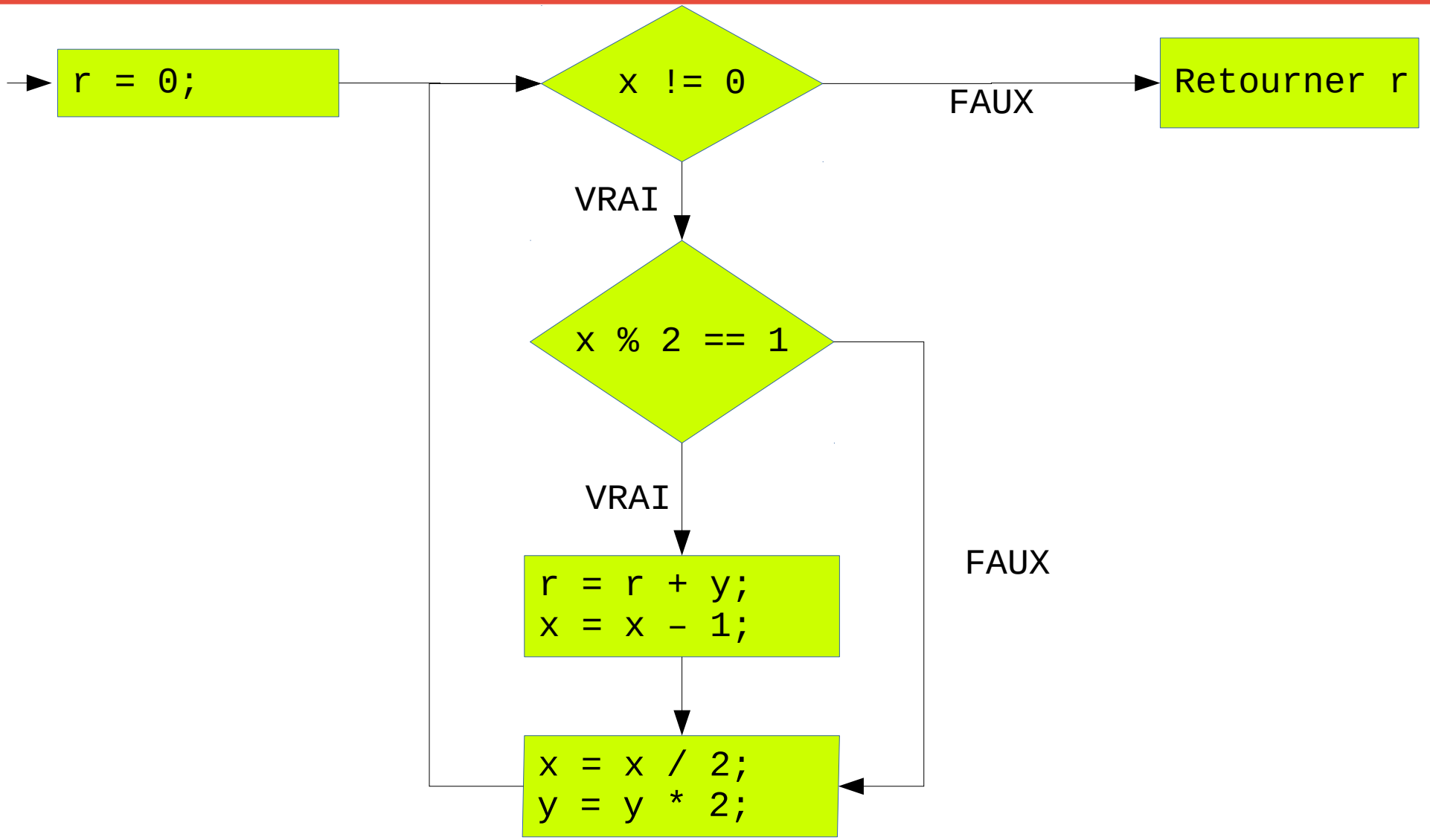
Comment les programmes sont-ils exécutés?

Les langages structurés suivent un « pointeur d'exécution »

Dans le langage courant, cela signifie « À quel numéro de ligne sommes-nous dans l'exécution du programme ».

On peut visualiser l'exécution d'un programme à l'aide d'un diagramme de flux et en suivant les flèches de haut en bas. Notre doigt, c'est le pointeur d'exécution.

Le diagramme de flux de la multiplication à la russe.



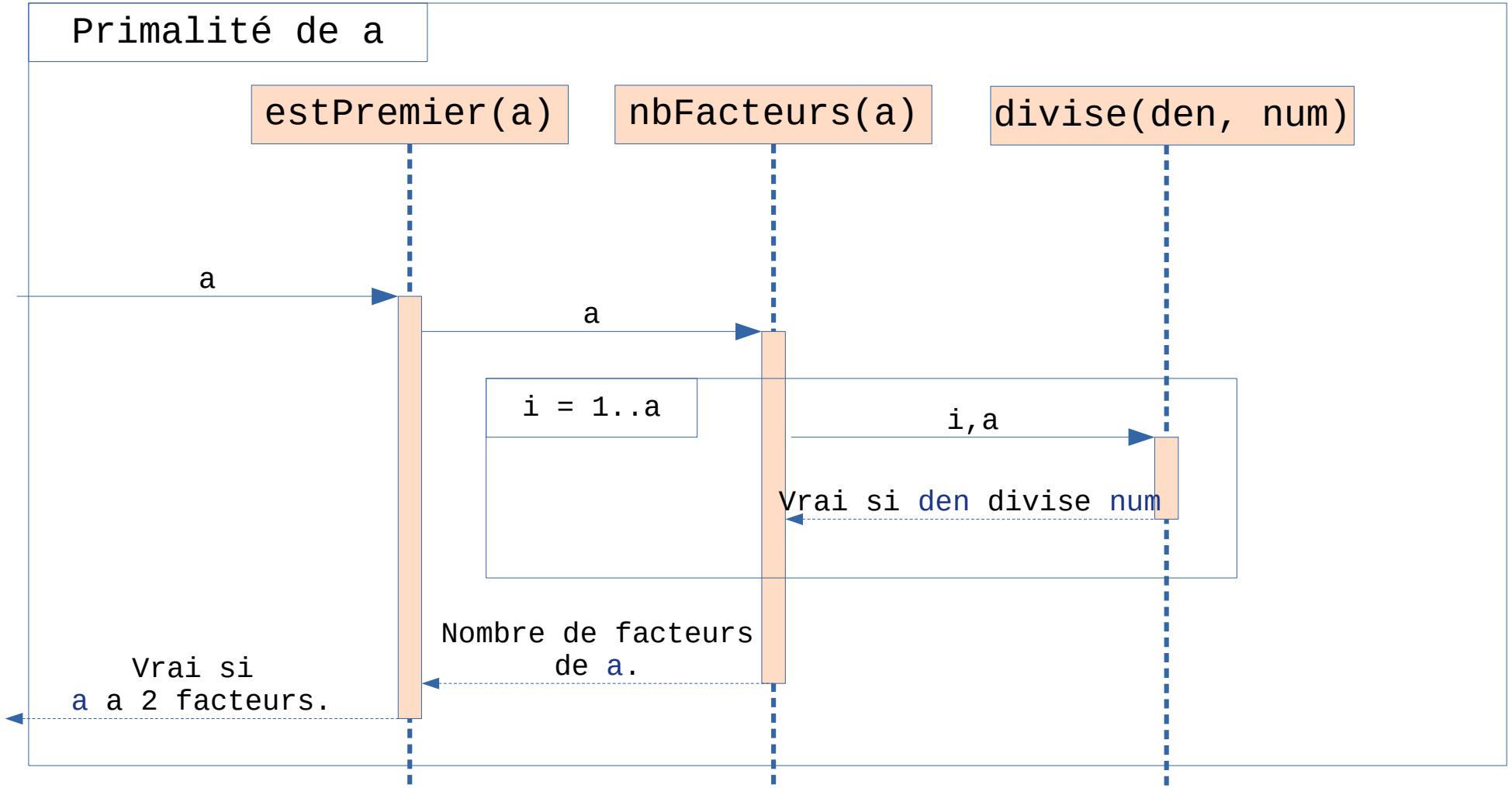
Le pointeur d’exécution

Sur les applications très simples, on peut facilement suivre le diagramme de flux.

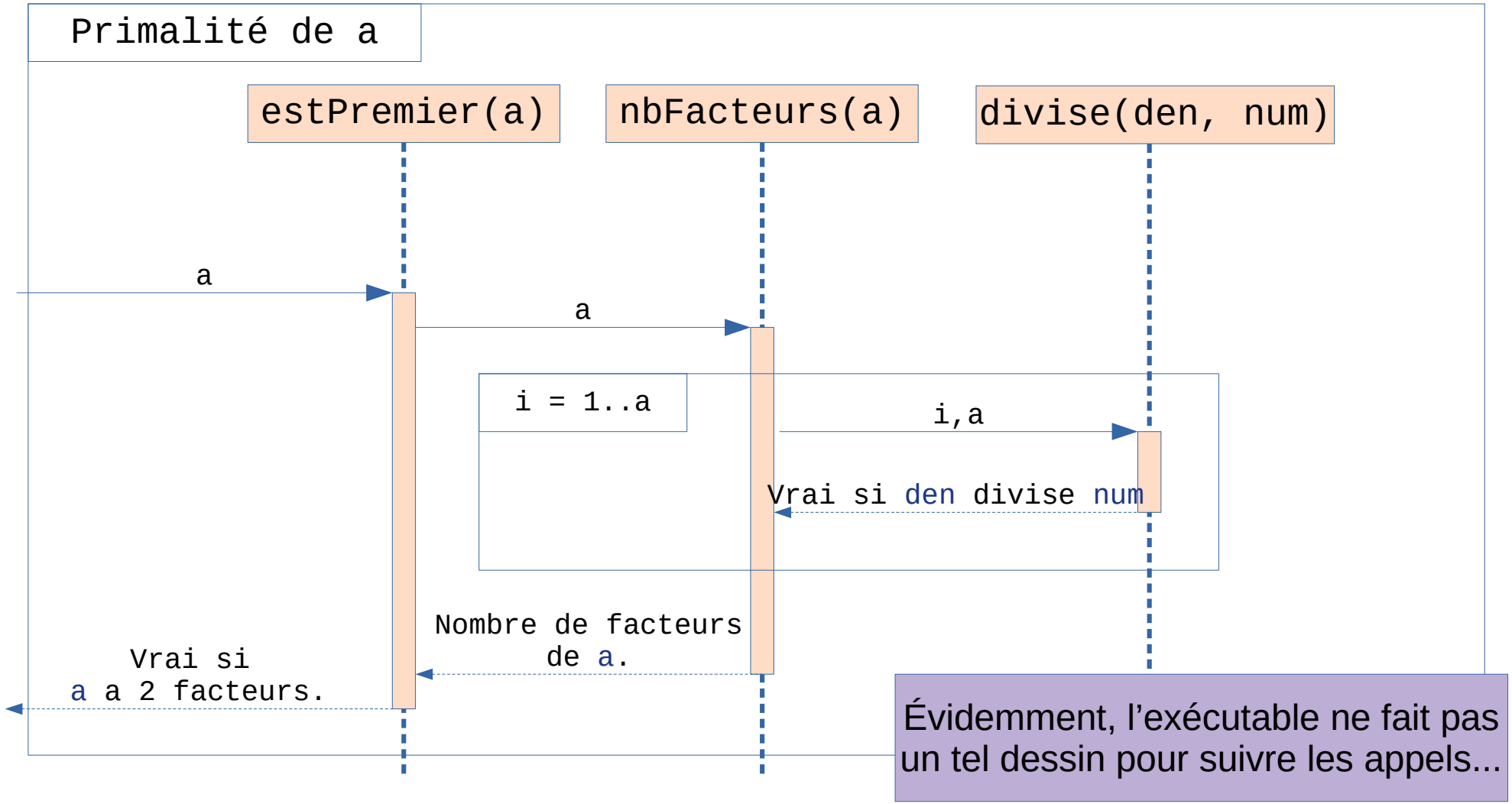
Mais comment l’exécution s’y retrouve si...

- On a beaucoup d’instructions?
- On a des appels de fonctions?
- … On fait du multithread? (dans un cours futur)

Le diagramme de séquence d'un petit programme.



Le diagramme de séquence d'un petit programme.



La solution : la pile d’exécution

L’exécution se fait à la fois en conservant un pointeur d’exécution, mais aussi une **pile d’exécution**.

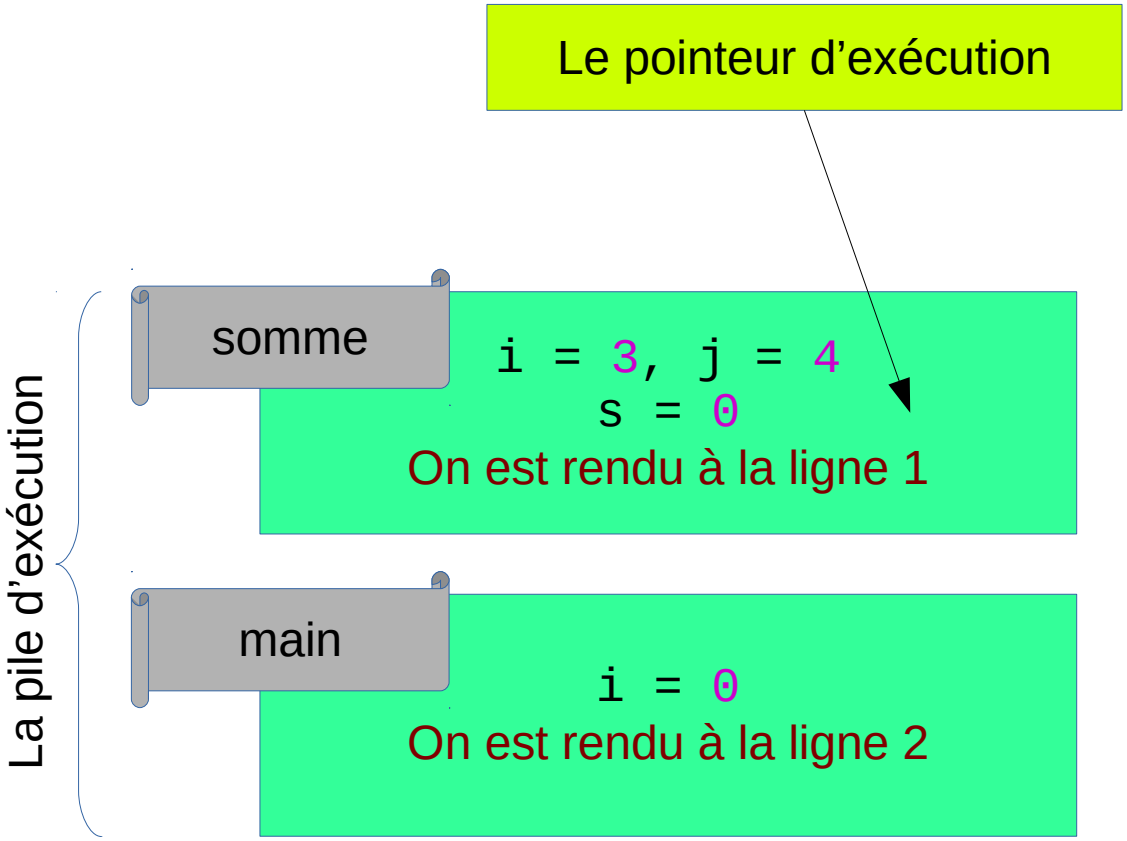
Une pile (*stack*), c’est une structure de données très simple : Dernier arrivé premier servi.

La pile d'exécution

Voyons un exemple avec des appels de fonction…

```
int somme(int i, int j)
{
   int s = 0;
   s = i + j;
   return s;
}

int main()
{
   int i = 0;
   i = somme(3,4);
}
```

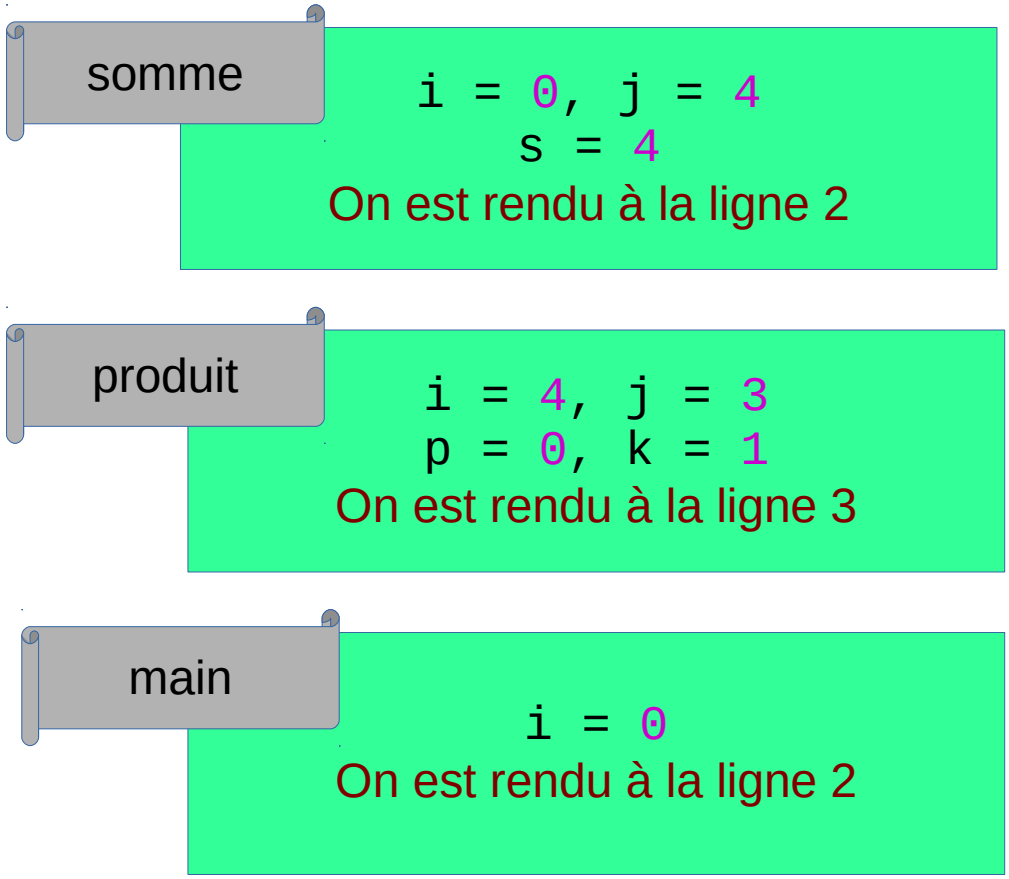


La pile d’exécution

```
int somme(int i, int j){
   int s = i + j;
   return s;
}
int produit (int i, int j){
   int p = 0;
   for (int k = 1; k <= j; k ++){
      p = somme(p, i);
   }
   return p;
}
int main(){
   int i = 0;
   i = produit(4,3);
}
```

Exercice : dessiner la pile pour l'appel à produit(4, 3);

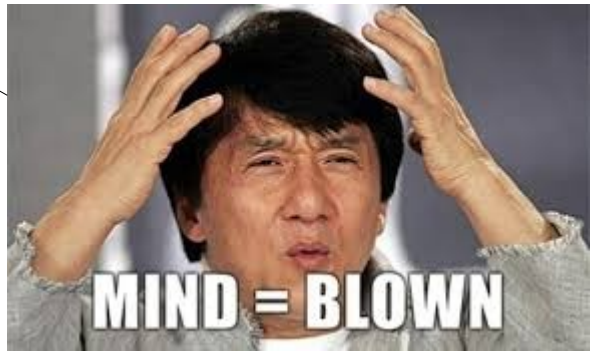
```
int somme(int i, int j){
   int s = i + j;
   return s;
}
int produit (int i, int j){
   int p = 0;
   for (int k = 1; k <= j; k ++){
      p = somme(p, i);
   }
   return p;
}
int main(){
   int i = 0;
   i = produit(4, 3);
}
```



Une fonction peut s'appeler elle même! C'est la *récurrence*.

```
int factorielle(int i)
{
   if (i <= 1)
      return i;
   return i * factorielle(i - 1);
}

int main()
{
   int i = 0;
   i = factorielle(4);
}
```



Une minute de silence pour apprécier la merveille qu'est la pile d'exécution

À tout instant, nous connaissons la profondeur des appels de fonctions. (`print(Thread.callStackSymbols)`)

On peut avoir n'importe quel ordre d'appel de fonctions sans que le pointeur d'exécution ne se perde.

On peut avoir des fonctions récursives.

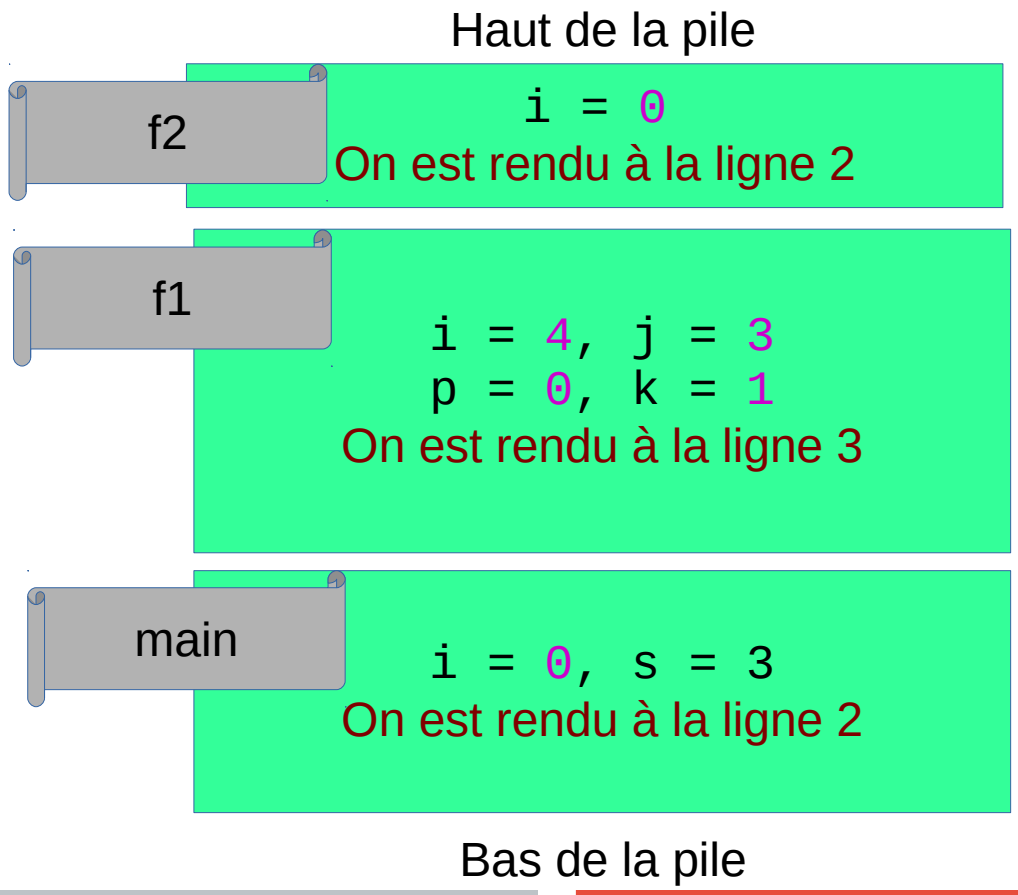
Les variables locales ont une portée précise. Elles « meurent » après le retour et n'encombrent plus la mémoire.

La pile d’exécution – À retenir

À la base de la pile est le `main`.

Chaque appel de fonction empile une case sur la pile. Cette case contient toutes les variables locales.

Au `return`, la case du dessus est effacée et le pointeur d’exécution se retrouve dans la case directement en dessous.

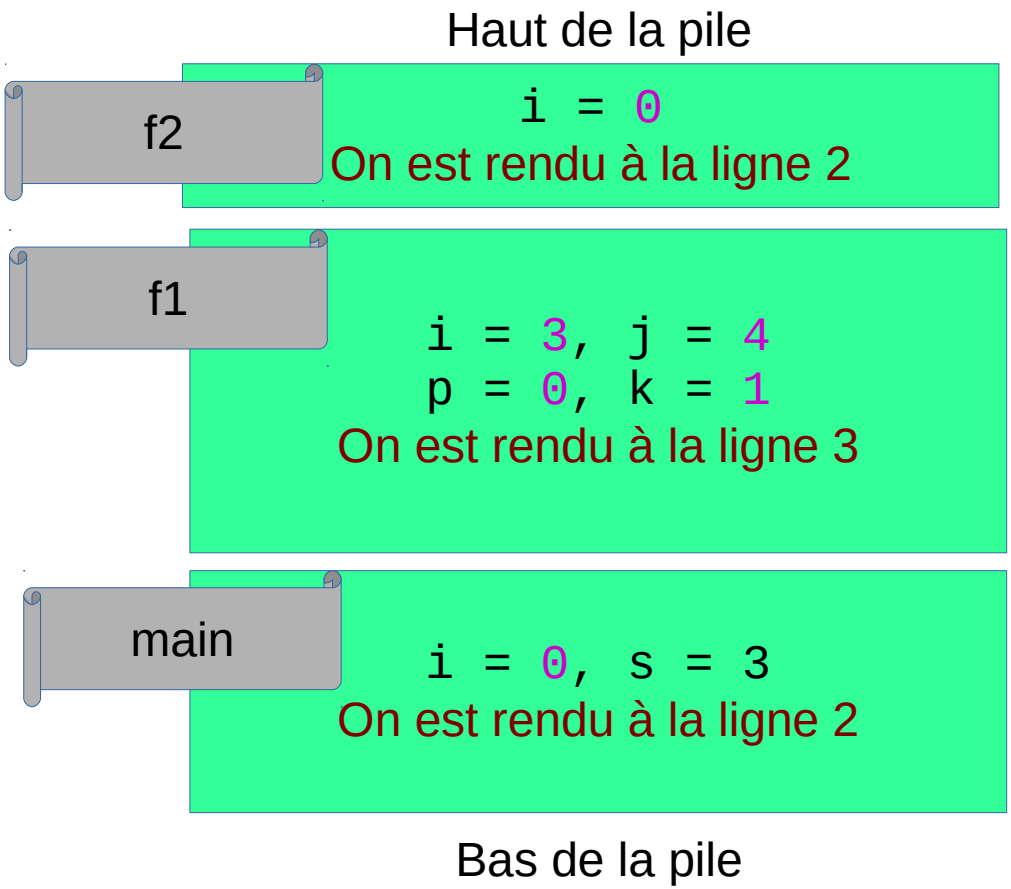


La pile d'exécution – À retenir

À la base de la pile est le `main`.

Chaque appel de fonction empile une case sur la pile. Cette case contient toutes les variables locales.

Au `return`, la case du dessus est effacée et le pointeur d'exécution se retrouve dans la case directement en dessous.



Elle ne suffit pas pour tous les programmes...

Revenons aux tableaux...

Essayons de retourner une variable...

```
double moyenne(int tableau[], int taille){
    int m;
    ...
    return m;
}

int[] renverser(int source[], int taille){
    int dest[taille];
    ...
    return dest;
}
```

retourne la valeur de `m`.

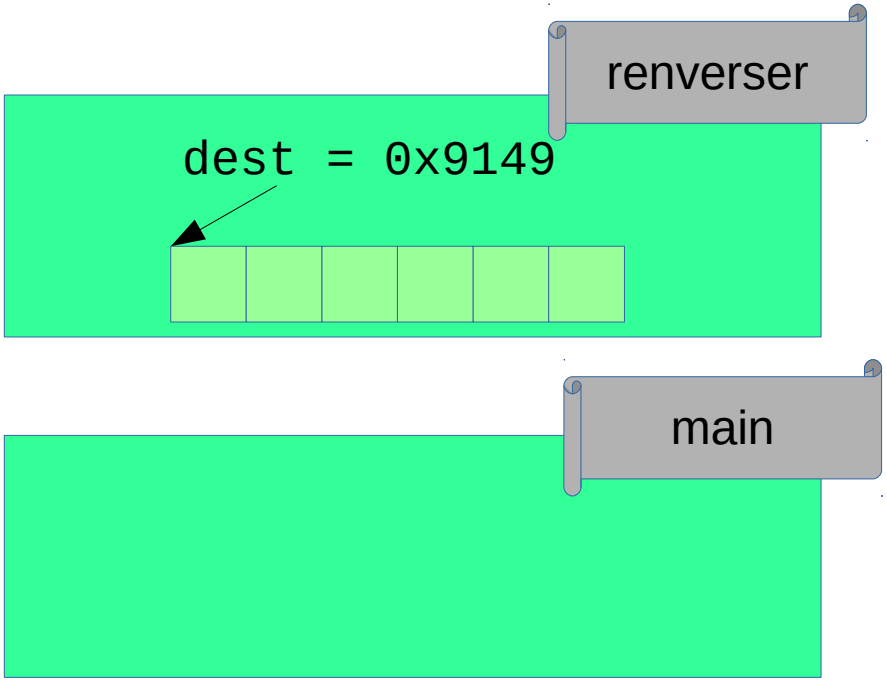
retourne la valeur de `dest` Mais ...

La pile et les pointeurs

```
int* renverser(int source[], int taille){
  int dest[taille];
  for (int i = 0; i < taille; i++)
    dest[i] = source[taille - i – 1];
  return dest;
}

int main(){
   int t[] = {55, 2, 3, 1, 49, 9};
   int* r = renverser(t, 6);

   // Imprimer t et r
}
```

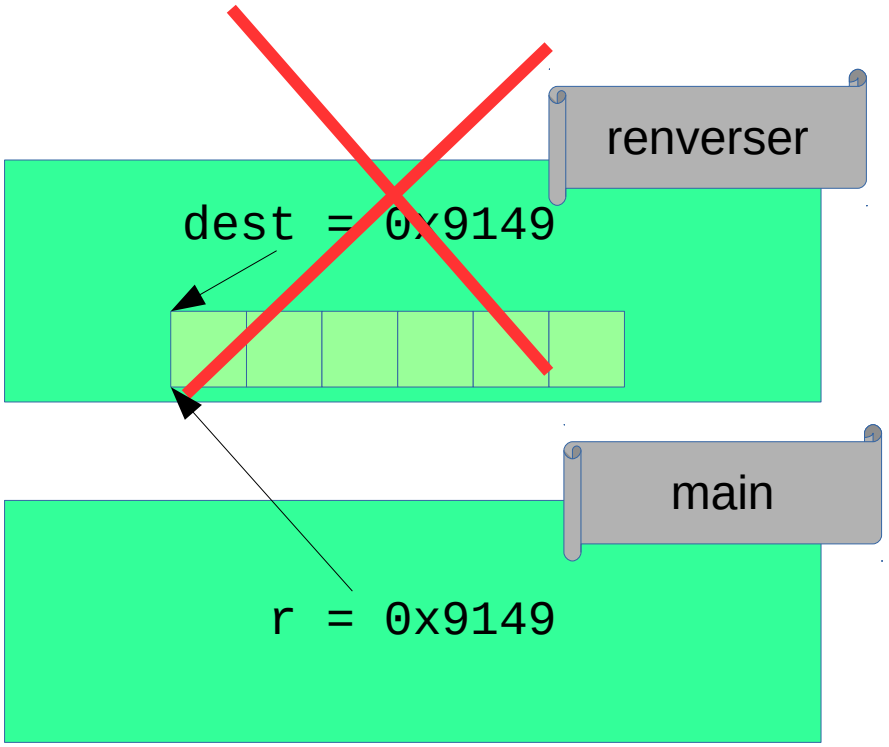


La pile et les pointeurs

```
int* renverser(int source[], int taille){
  int dest[taille];
  for (int i = 0; i < taille; i++)
    dest[i] = source[taille - i – 1];
  return dest;
}

int main(){
   int t[] = {55, 2, 3, 1, 49, 9};
   int* r = renverser(t, 6);

   // Imprimer t et r
}
```

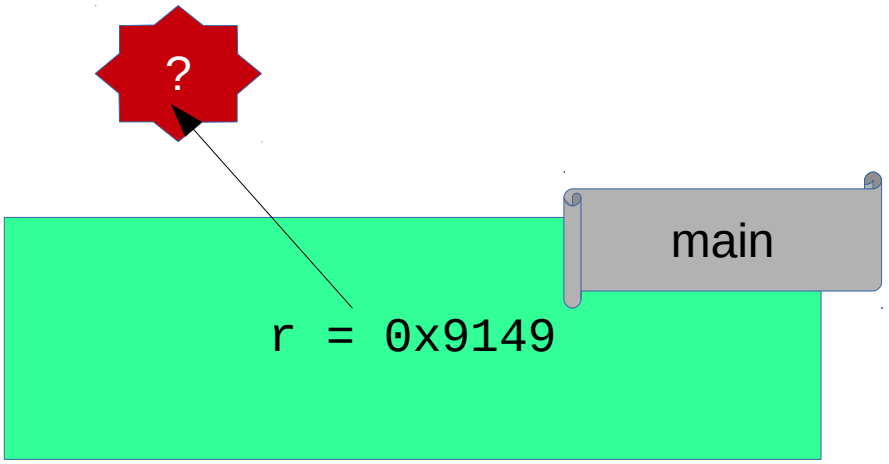


La pile et les pointeurs

```
int* renverser(int source[], int taille){
  int dest[taille];
  for (int i = 0; i < taille; i++)
    dest[i] = source[taille - i – 1];
  return dest;
}

int main(){
   int t[] = {55, 2, 3, 1, 49, 9};
   int* r = renverser(t, 6);

   // Imprimer t et r
}
```

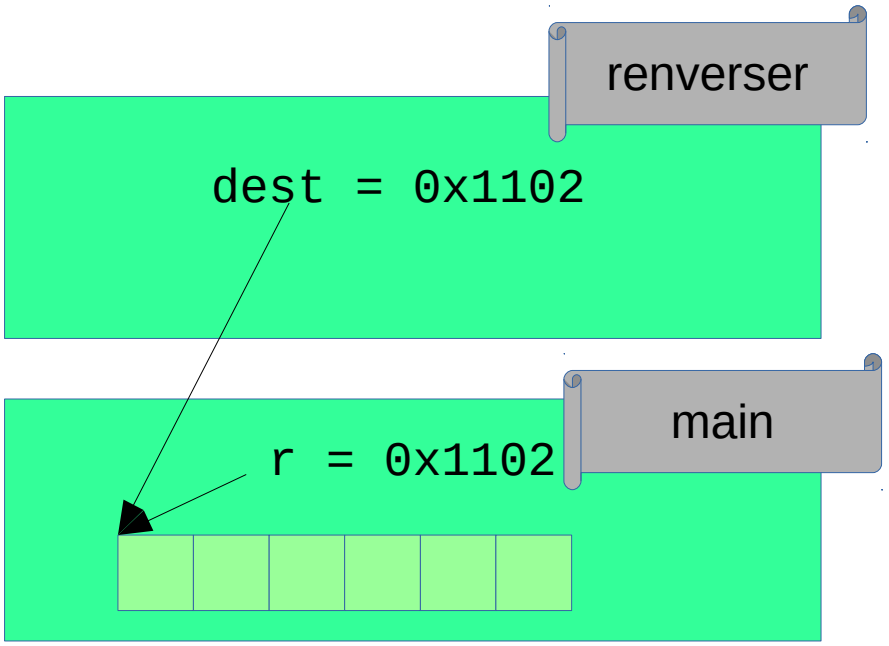


On ne peut pas pointer vers une case plus haut dans la pile...

Solution 1 : On passe le tableau par référence.

```
void renverser(int source[], int dest[], int taille){
  for (int i = 0; i < taille; i++)
    dest[i] = source[taille - i - 1];
}

int main(){
   int t[] = {55, 2, 3, 1, 49, 9};
   int r[6];
   renverser(t, r, 6);
   // Imprimer r et t
}
```



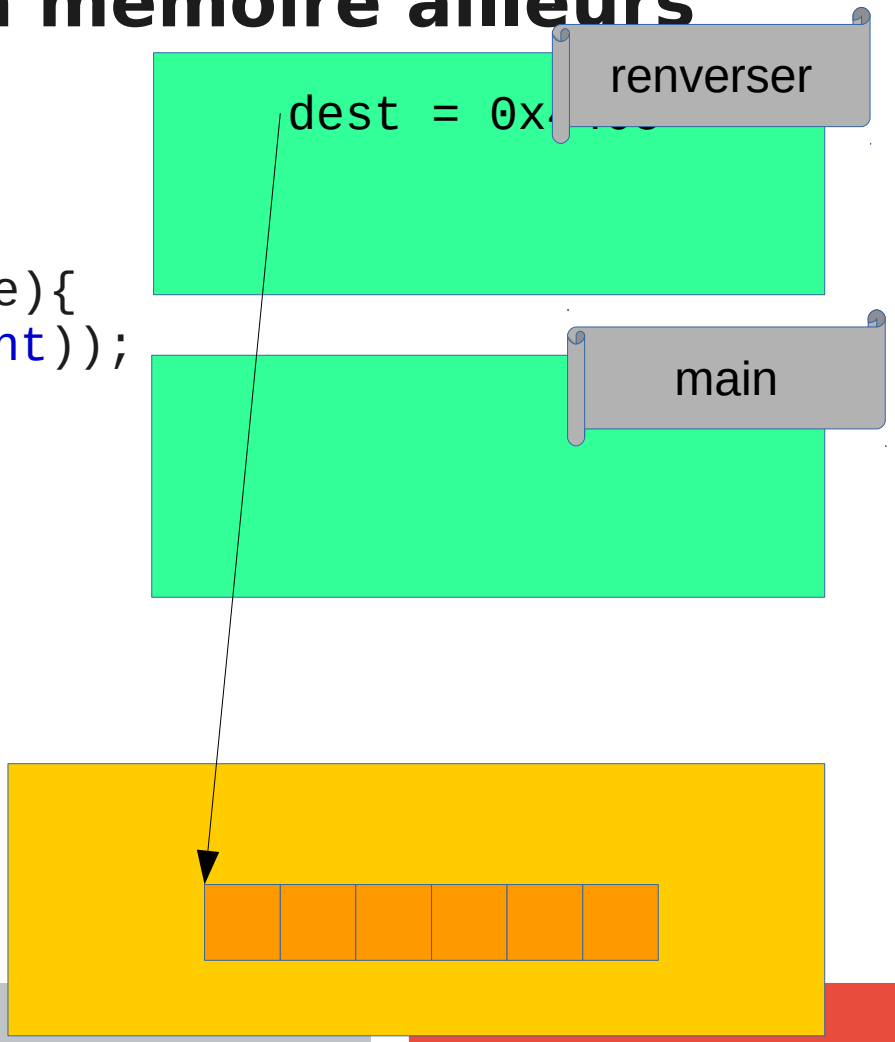
On ne peut pas pointer vers une case plus haut dans la pile...

Solution 2 : On réserve de la mémoire ailleurs que dans la pile. Le tas!

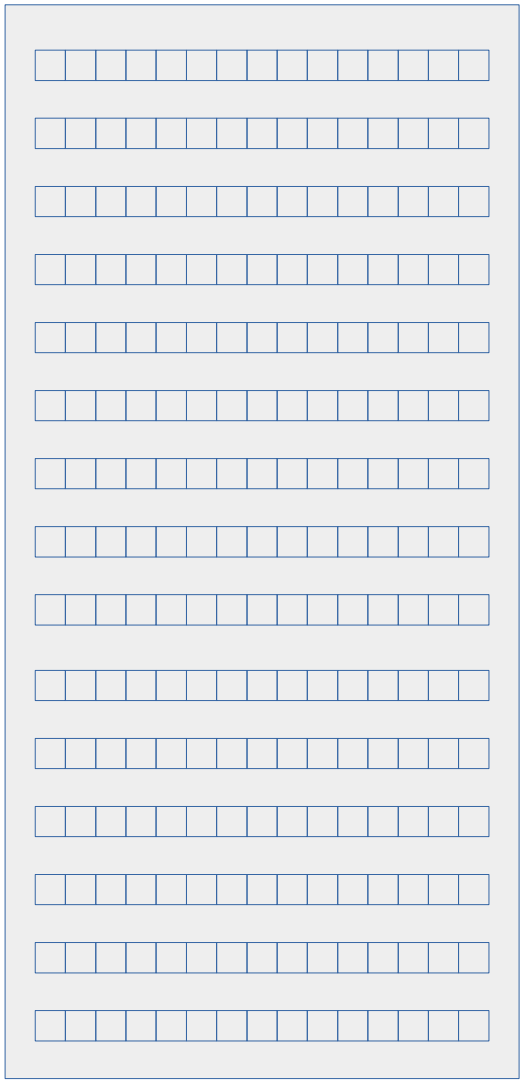
```
int* renverser(int source[], int taille){
  int* dest = malloc(taille * sizeof(int));
  for (int i = 0; i < taille; i++)
    dest[i] = source[taille - i – 1];
  return dest;
}

int main(){
   int t[] = {55, 2, 3, 1, 49, 9};
   int* r = renverser(t, 6);

   // Imprimer t et r
}
```

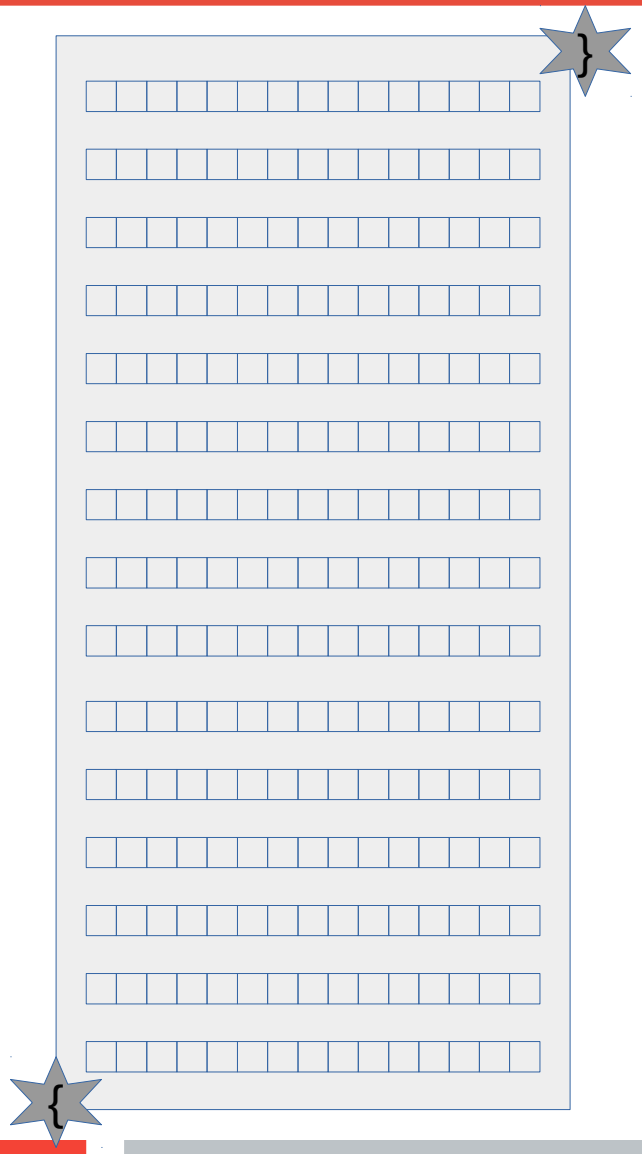


La mémoire…



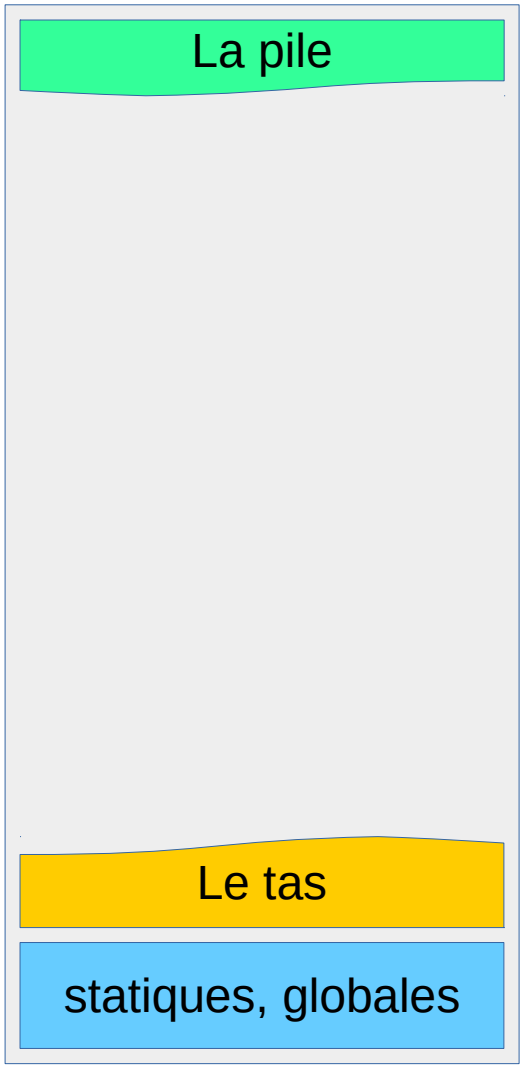
Au début de l'exécution d'un programme, l'OS alloue un espace mémoire pour notre programme.

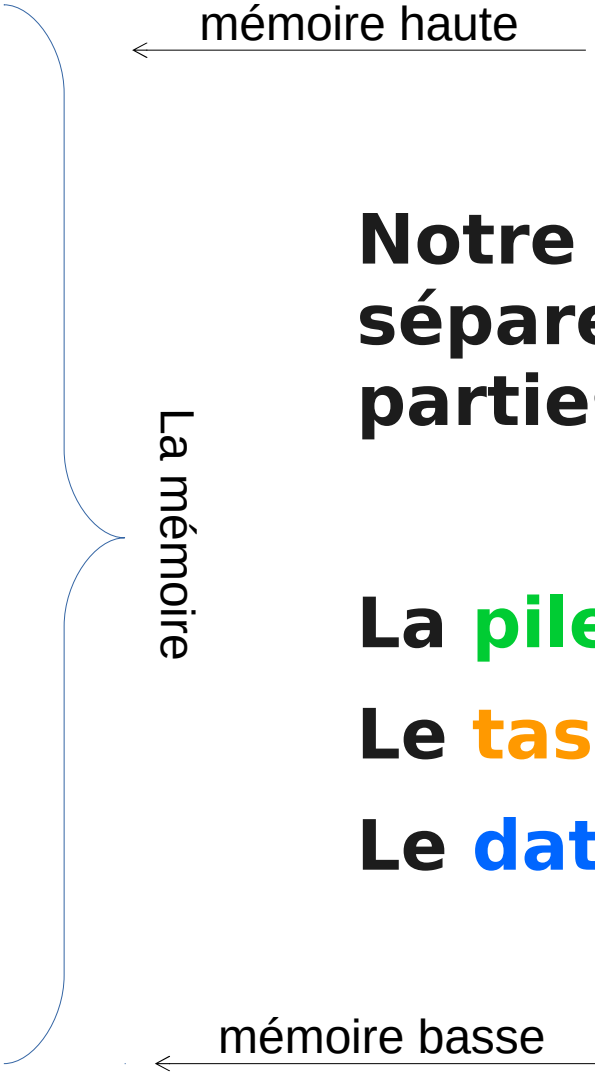
La mémoire…



Le programme connaît l'adresse de la première case et de la dernière case.

La mémoire...





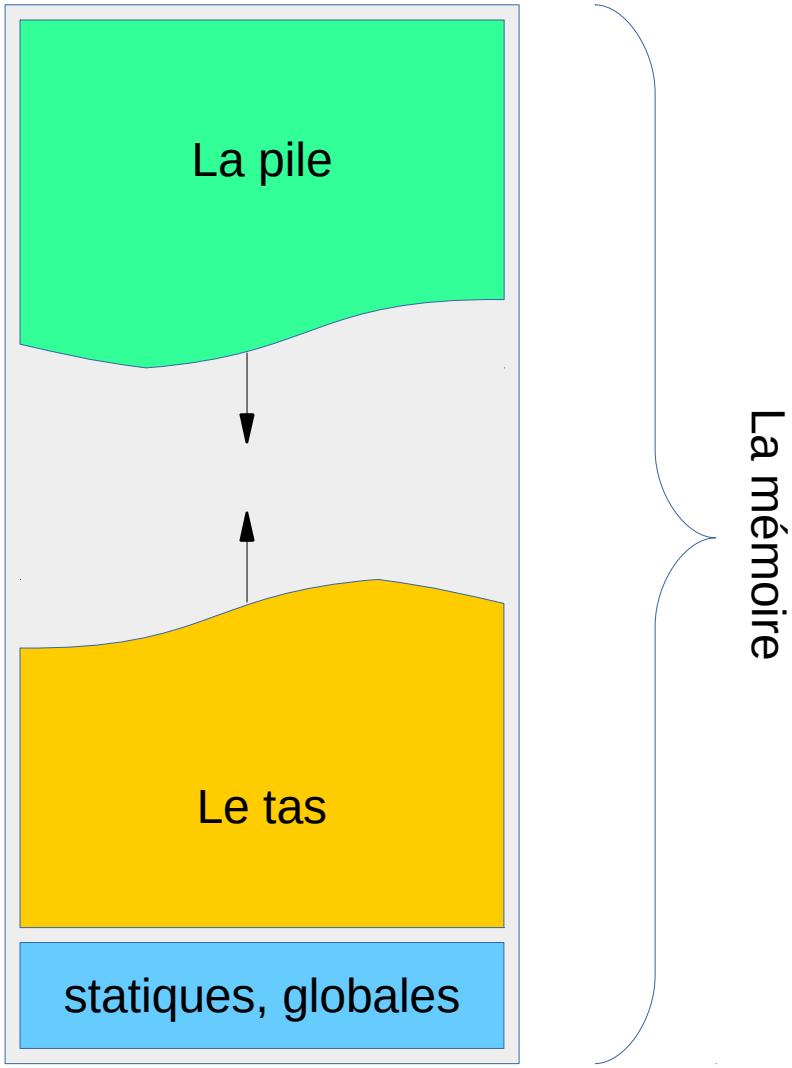
Notre programme va la séparer en trois parties.

La pile (*stack*),

Le tas (*heap*) et

Le data statique.

La pile… le tas…



La pile : on la connaît, c'est les variables locales.

Taille dynamique

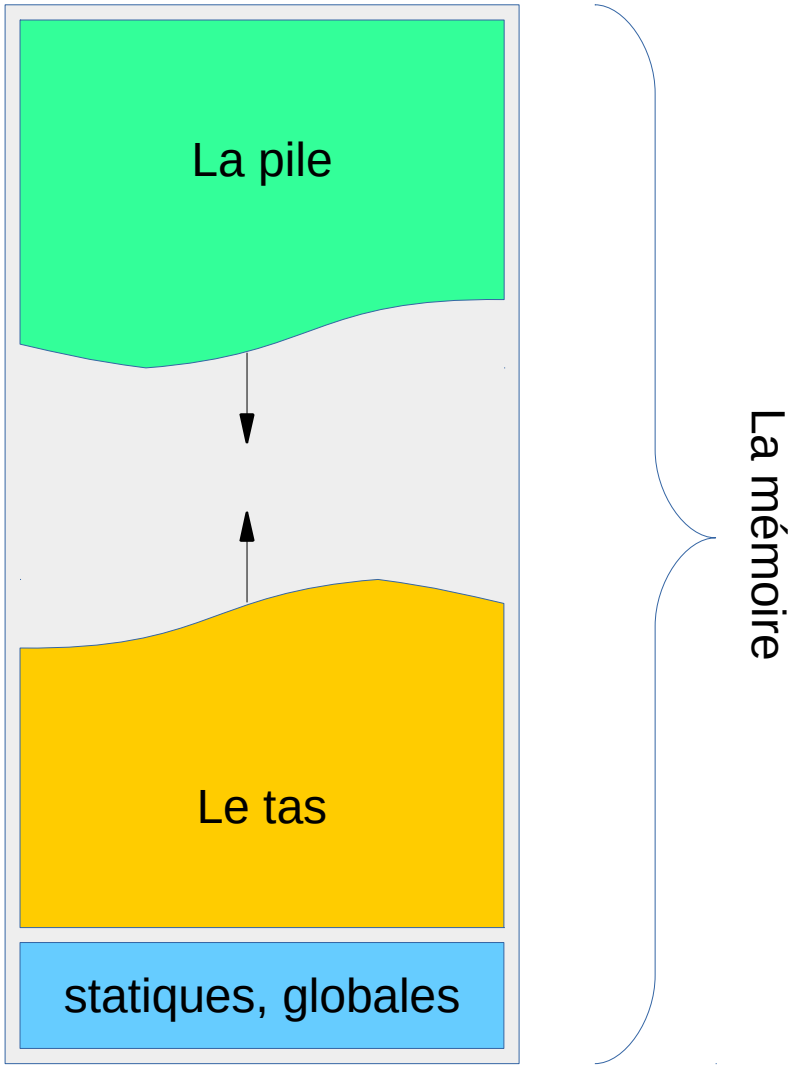
Le tas : ce sont les variables allouées dynamiquement (avec un `malloc`)

Taille dynamique

Le data : (entre autres) les instructions, les constantes et les variables globales.

Taille connue d'avance

La pile… le tas…

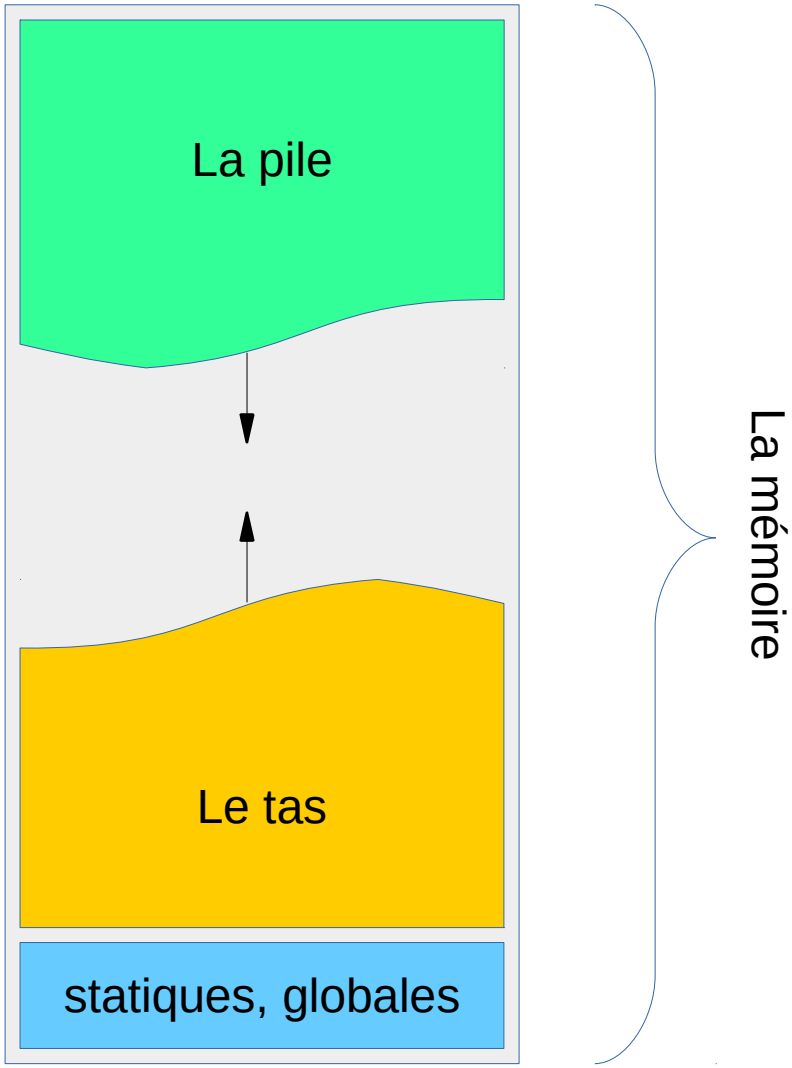


La pile : on la connaît, c'est les variables locales.

Le tas : ce sont les variables allouées dynamiquement (avec un `malloc`)

Le data : (entre autres) les instructions, les variables globales.

La pile… le tas…



La pile : on la connaît, c'est les variables locales.

Taille dynamique

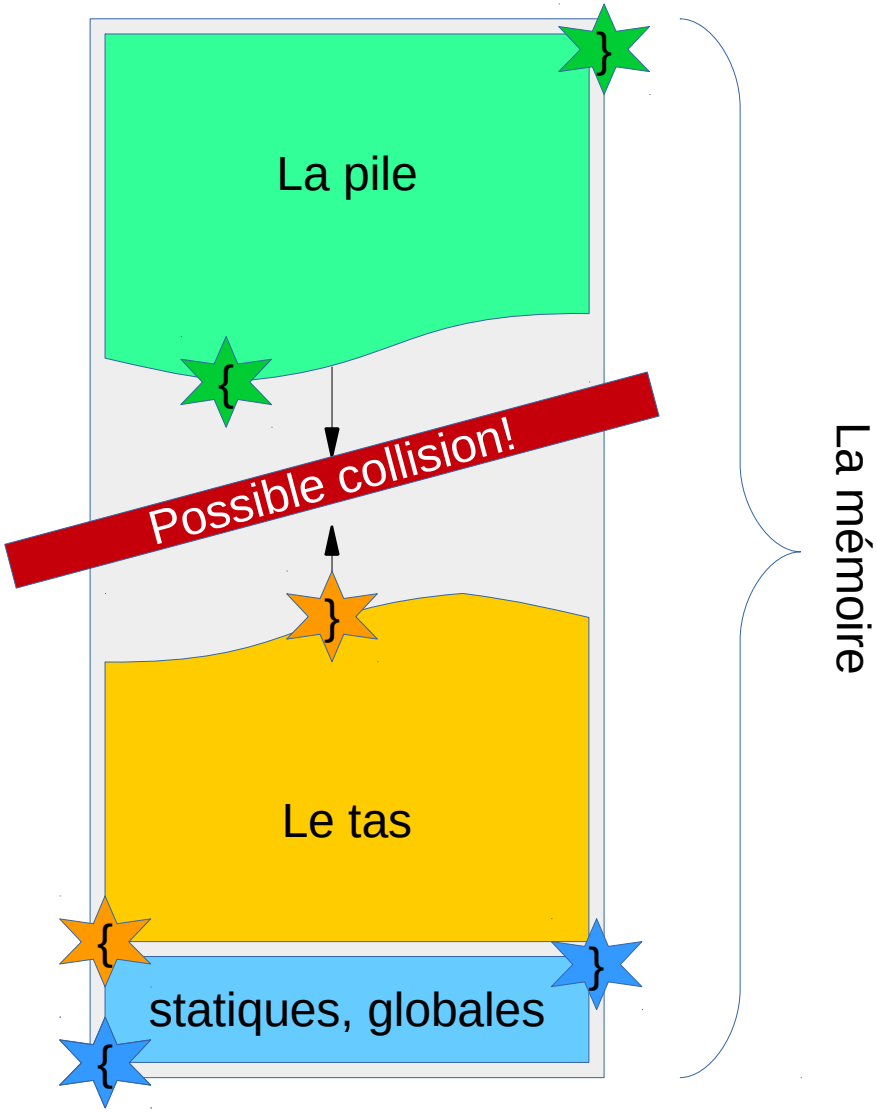
Le tas : ce sont les variables allouées dynamiquement (avec un `malloc`)

Taille dynamique

Le data : (entre autres) les instructions, les constantes et les variables globales.

Taille connue d'avance

La pile… le tas…



La pile : on la connaît, c'est les variables locales.

Taille dynamique

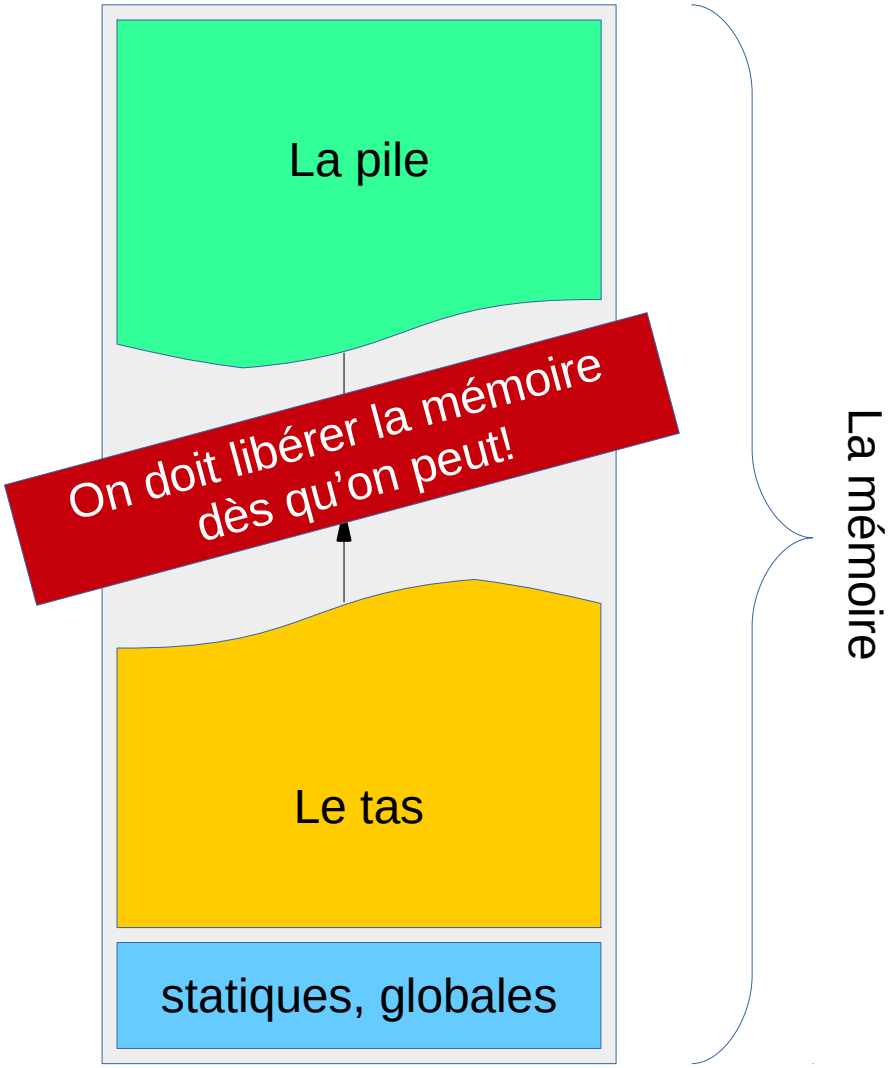
Le tas : ce sont les variables allouées dynamiquement (avec un `malloc`)

Taille dynamique

Le data : (entre autres) les instructions, les constantes et les variables globales.

Taille connue d'avance

La pile… le tas…



La pile : on la connaît, c'est les variables locales.

Mémoire libérée au `return`.

Le tas : ce sont les variables allouées dynamiquement (avec un `malloc`)

Mémoire libérée avec la fonction `free` (ou à la fin de l'exécution).

Le data : (entre autres) les instructions, les constantes et les variables globales.

Mémoire libérée à la fin de l'exécution du programme.

la fonction `malloc` plus en détail…

***m**emory **alloc**ation*.
(Java : *new*)

```
void* malloc(size_t taille);
```

Retourne un pointeur vers n'importe quoi (`void`) et réserve `taille` octets à partir de cette adresse.

Par exemple :

`sizeof` retourne le nombre d'octets que prend le type

```
#include <stdlib.h>
int* a = (int*)malloc(sizeof(int));
int* b = (int*)malloc(sizeof(int) * 6);
int* c = (int*)malloc(24);
```

la fonction `malloc` plus en détail...

```
void* malloc(size_t taille);
```

Si l'allocation a échoué, le pointeur sera `NULL`.

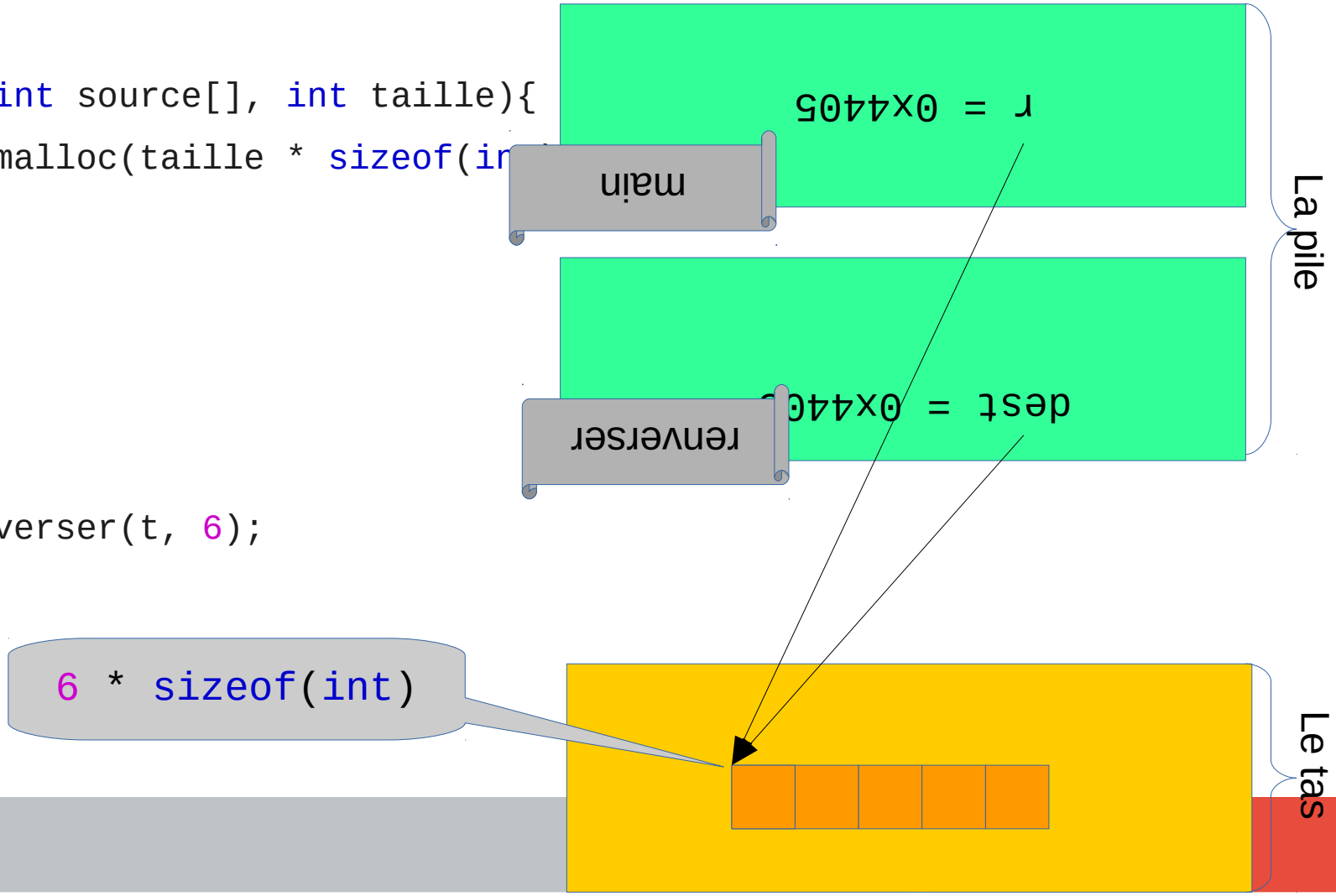
Par exemple :

```
int* a = (int*)malloc(sizeof(int));
if (a == NULL){
   fprintf(stderr, "Erreur de mémoire!");
   return -1;
}
```

La libération de la mémoire

Libérons la mémoire réservée dans le tas

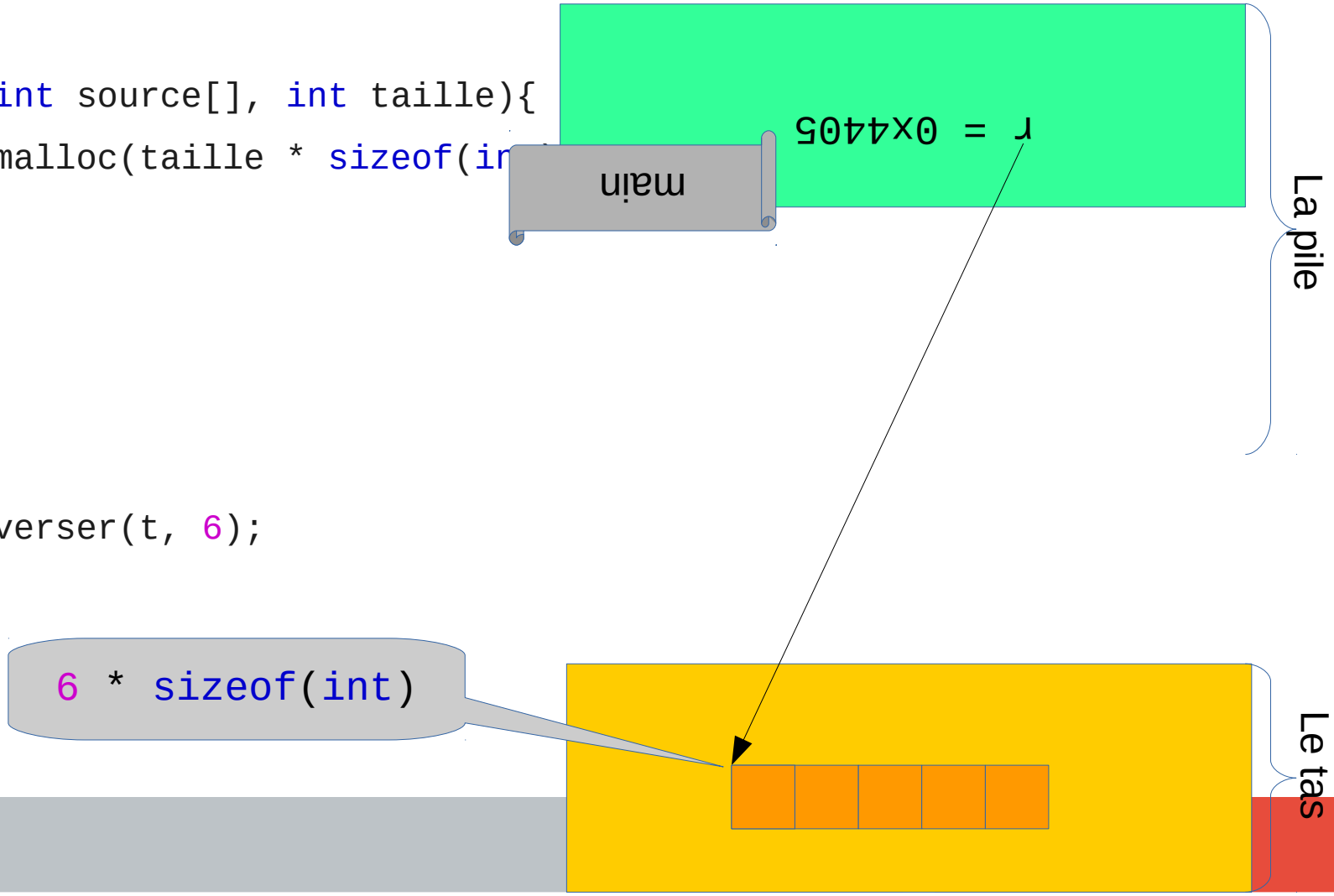
```
int* renverser(int source[], int taille){
    int* dest = malloc(taille * sizeof(in
    ...
    return dest;
}
int main(){
    ...
    int* r = renverser(t, 6);
    ...
    free(r);
}
```



La libération de la mémoire

Libérons la mémoire réservée dans le tas

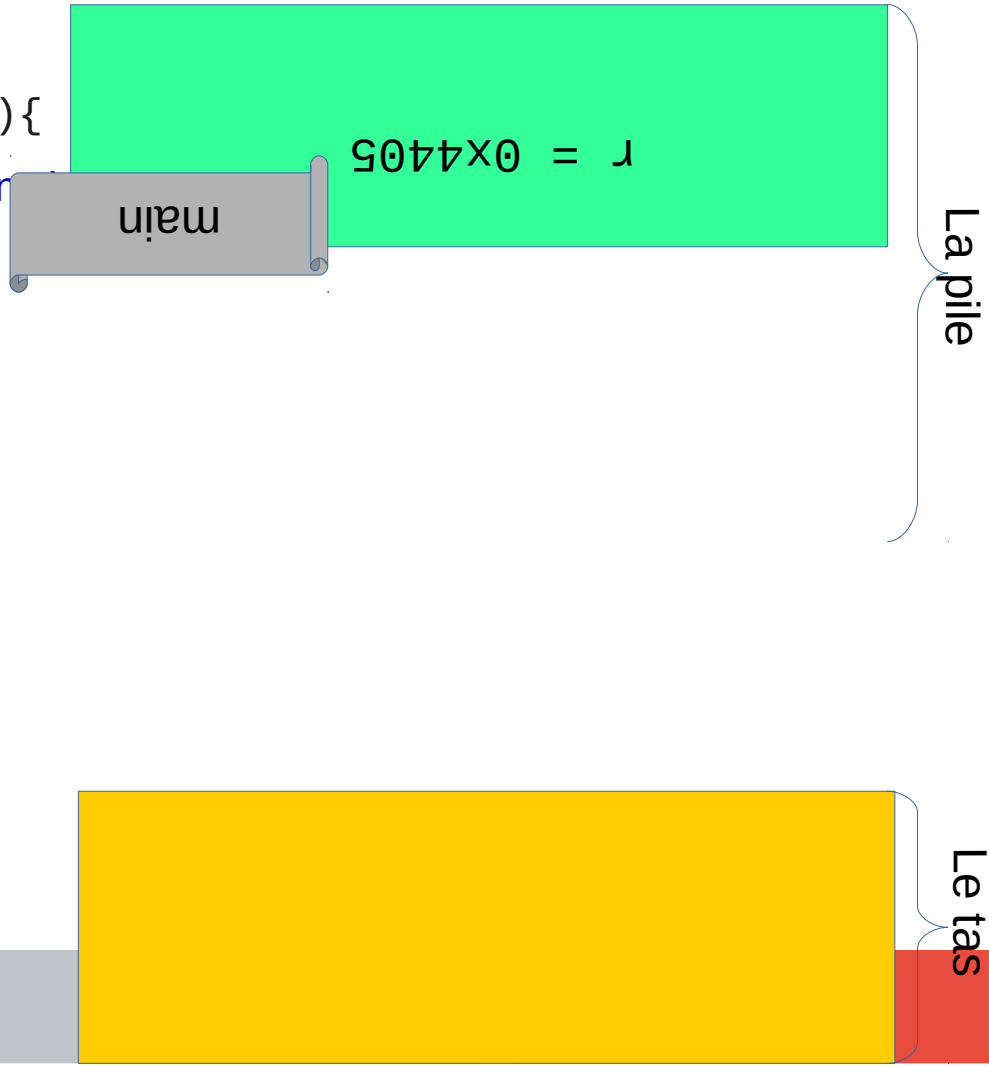
```
int* renverser(int source[], int taille){
    int* dest = malloc(taille * sizeof(in
    ...
    return dest;
}
int main(){
    ...
    int* r = renverser(t, 6);
    ...
    free(r);
}
```



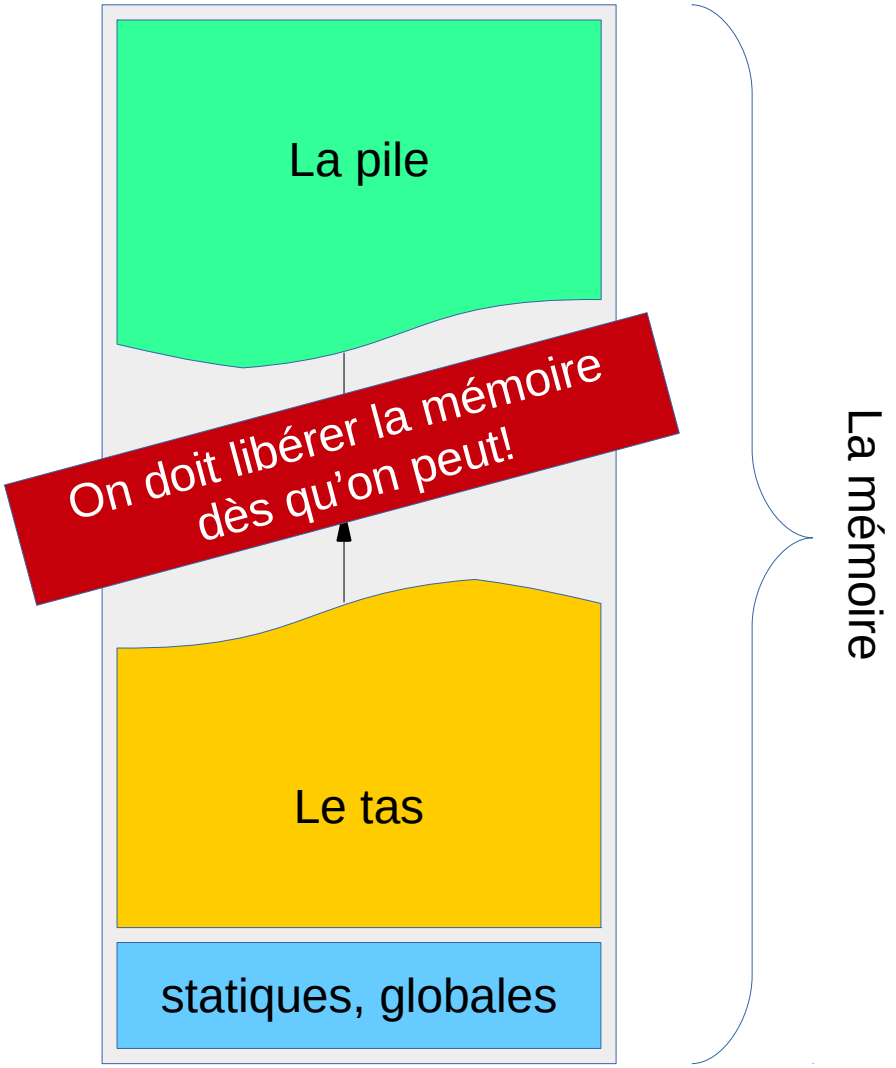
La libération de la mémoire

Libérons la mémoire réservée dans le tas

```
int* renverser(int source[], int taille){
   int* dest = malloc(taille * sizeof(in
   ...
   return dest;
}
int main(){
   ...
   int* r = renverser(t, 6);
   ...
   free(r);
}
```



La pile… le tas…



La pile : on la connaît, c'est les variables locales.

Mémoire libérée au `return`.

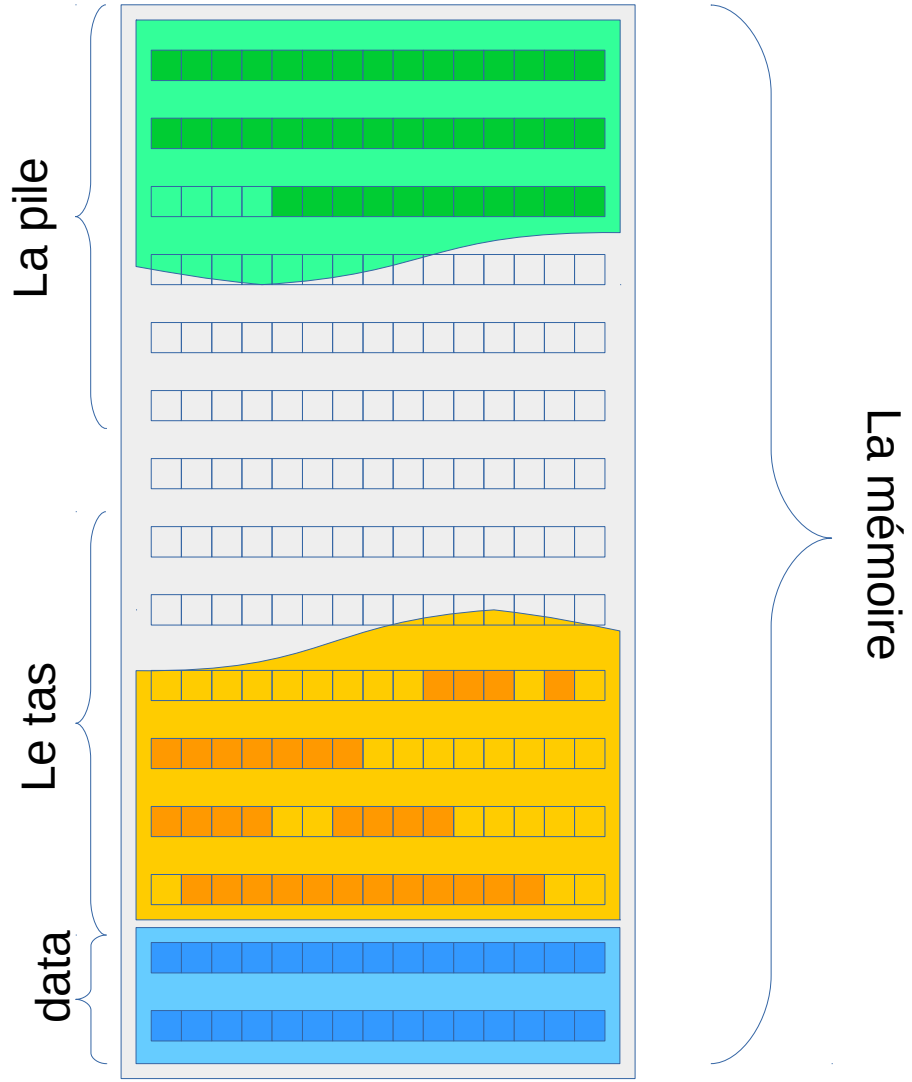
Le tas : ce sont les variables allouées dynamiquement (avec un `malloc`)

Mémoire libérée avec la fonction `free` (ou à la fin de l'exécution).

Le data : (entre autres) les instructions, les constantes et les variables globales.

Mémoire libérée à la fin de l'exécution du programme.

La mémoire, en résumé.



La pile

Elle est dense.

Sa taille s'ajuste au cours de l'exécution

La mémoire est automatique.

Son accès est rapide.

Le tas

Il n'a pas de structure particulière.

Sa taille s'ajuste au cours de l'exécution

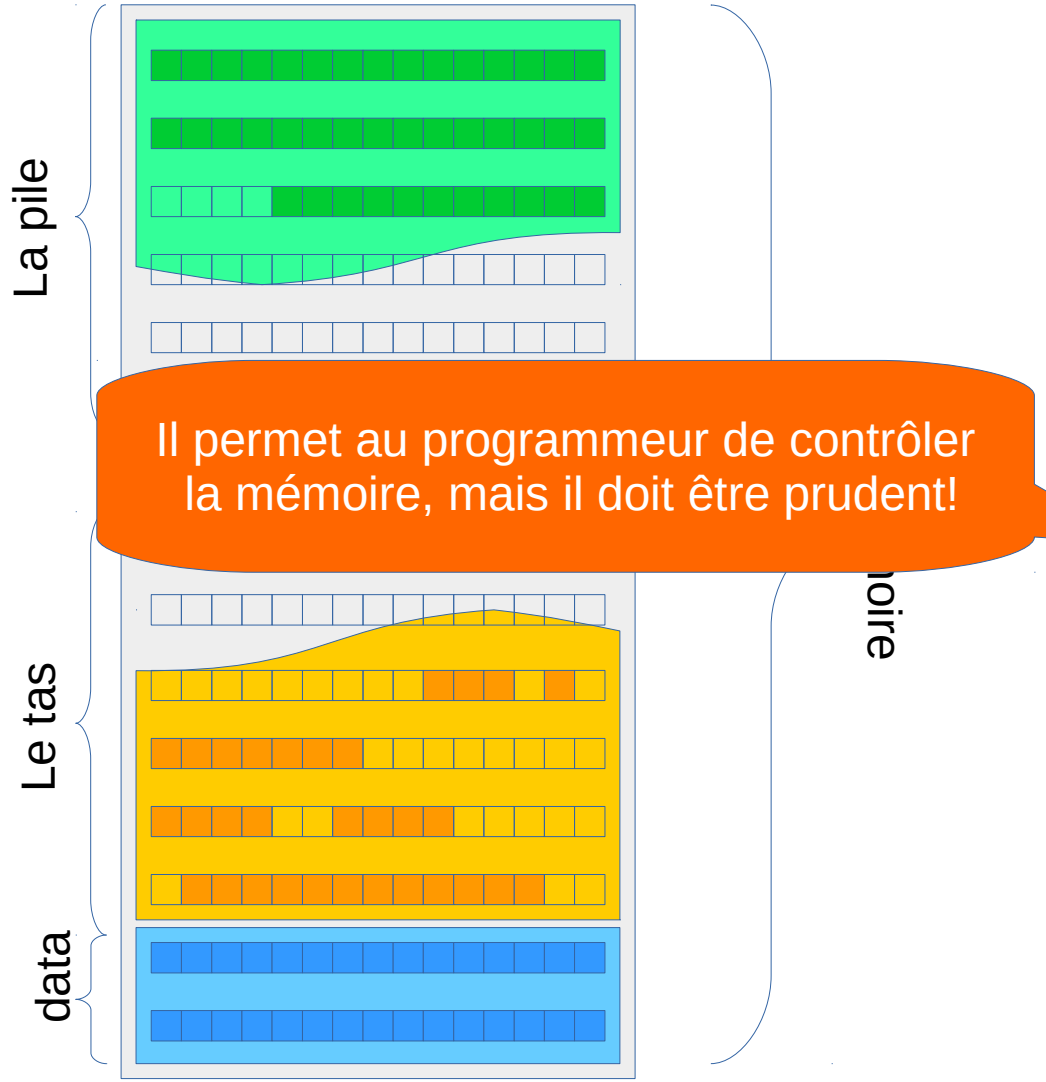
La mémoire est dynamique.

Le data

Rapide, compact et statique.

Sa taille est fixée à la compilation.

La mémoire, en résumé.



La pile

Elle est dense.

Sa taille s'ajuste au cours de l'exécution

La mémoire est automatique.

Son accès est rapide.

Le tas

Il n'a pas de structure particulière.

Sa taille s'ajuste au cours de l'exécution

La mémoire est dynamique.

Le data

Rapide, compact et statique.

Sa taille est fixée à la compilation.

Où se trouvent les données?

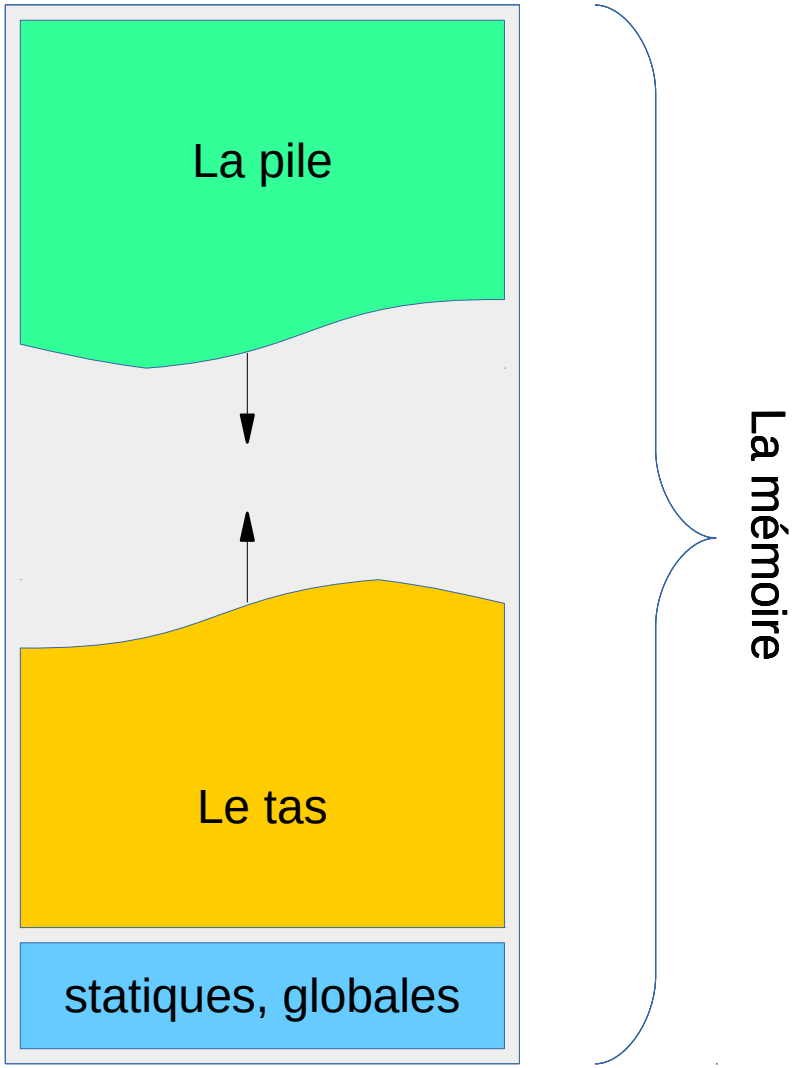
Ouvrez `dynamique.c`

A) Quelle est la valeur du pointeur `tabPile`?

B) Quelle est la valeur du pointeur `tabTas`?

C) Quelle est l'adresse de la fonction `imprimerTableau`?

D) Quelle est l'adresse de la variable globale `TAILLE`?



stdlib

La librairie `stdlib` vient avec toutes sortes de fonctions et de macros pour utiliser le tas :

`size_of` (c'est le type retourné par `sizeof`)

`NULL` (macro)

```
void* calloc(size_t nitems, size_t size);
void  free(void* ptr);
void* malloc(size_t size);
void* realloc(void* ptr, size_t size);
```

et bien d'autres...

Tableau de `nitems` cases de `size` octets chacun et initialise chaque case à `0`.

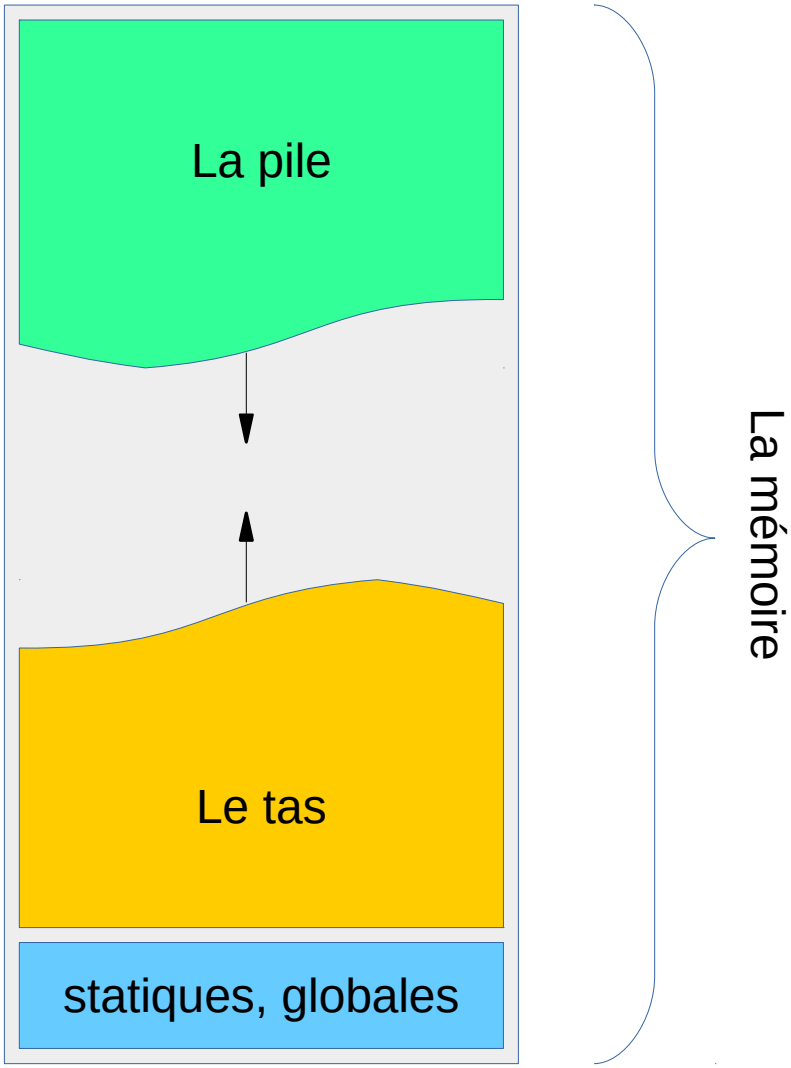
Libère la mémoire utilisée par `ptr` dans le tas.

Modifie la taille réservée pour par `ptr` dans le tas pour la nouvelle taille `size`.

Les fonctions de `stdlib`

E) Combien il faut-il d'octets pour stocker une adresse? (utilisez `sizeof`)

F) À quelle valeur les bits sont initialisés dans le tas? Dans la pile?

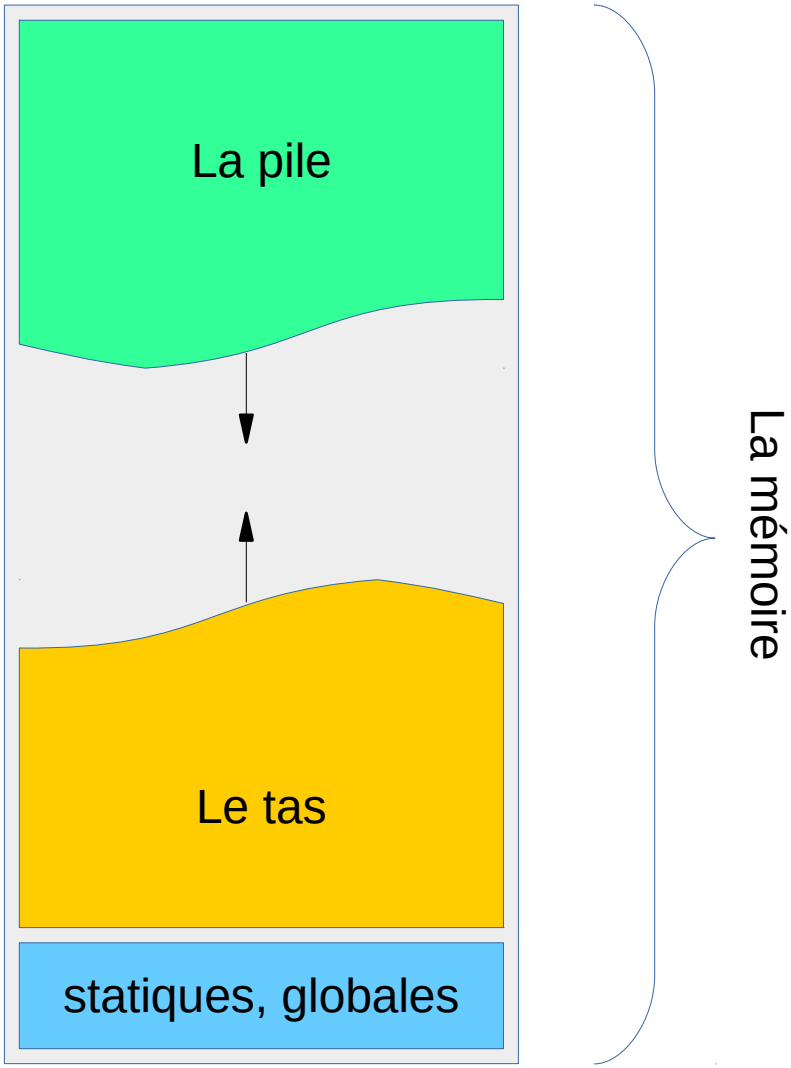


La pile, le tas et les variables.

G) Que se passera-t-il si on ne libère la mémoire utilisée par `tableauTas` à même la fonction `fabriquerTableauTas`?

H) Quelle est la portée d'une variable dans la pile? Sa durée de vie?

I) Quelle est la portée d'une variable dans le tas? Sa durée de vie?



Un exercice de programmation

Au TP1, on aura besoin de stocker un nombre arbitraire de **Client**s.

```
typedef struct Client Client;
struct Client{
    int instantArrivee; // L'instant où le client s'est ajouté à une file
    int nbArticles;     // Nb d'articles dans le panier du client
    // Vous pouvez ajouter des membres
};
```

Programmez la fonction

```
Client* creerClient(int instant, int nbArticles);
```

Appelez cette fonction dans le **main**.

N'oubliez pas de libérer la mémoire du tas!

« Ramassez les ordures! »

Un exercice de programmation (suite et fin)

Quelle est la taille maximale que votre système d'exploitation permet d'allouer à la pile?

Sur UNIX : `ulimit -a`

Essayez de faire déborder la pile.
Essayez de faire déborder le tas.

Déclarez de grandes variables
Faites trop d'appels récursifs.

Demandez un très grand nombre d'octets à `malloc` ou `calloc`.