Introduction

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

Introduction

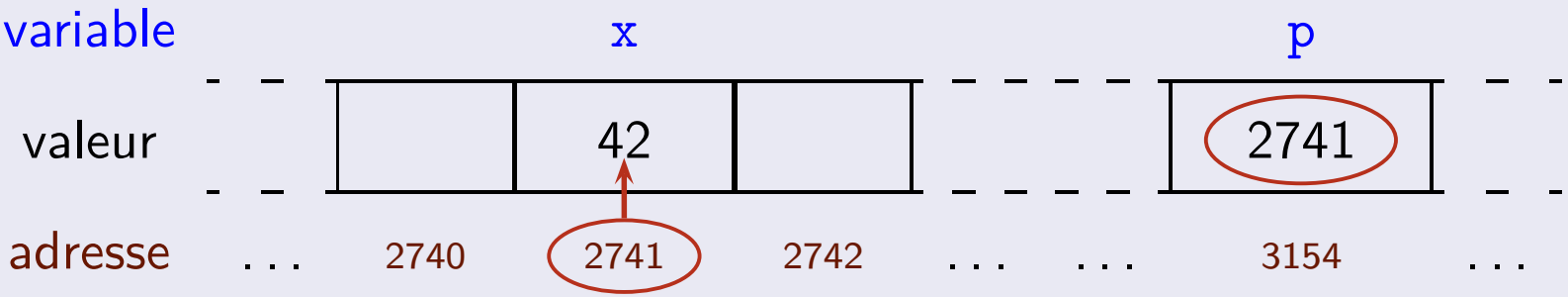
- De façon schématique, la mémoire d'un ordinateur s'apparente à un immense tableau dont chaque case a une adresse.
- Un pointeur est une variable contenant une de ces adresses.

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.

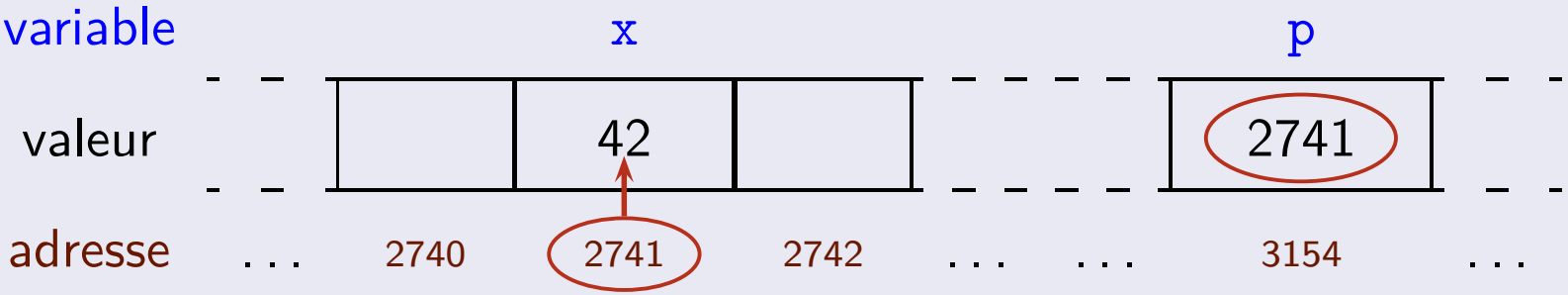
Introduction

- 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.



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.



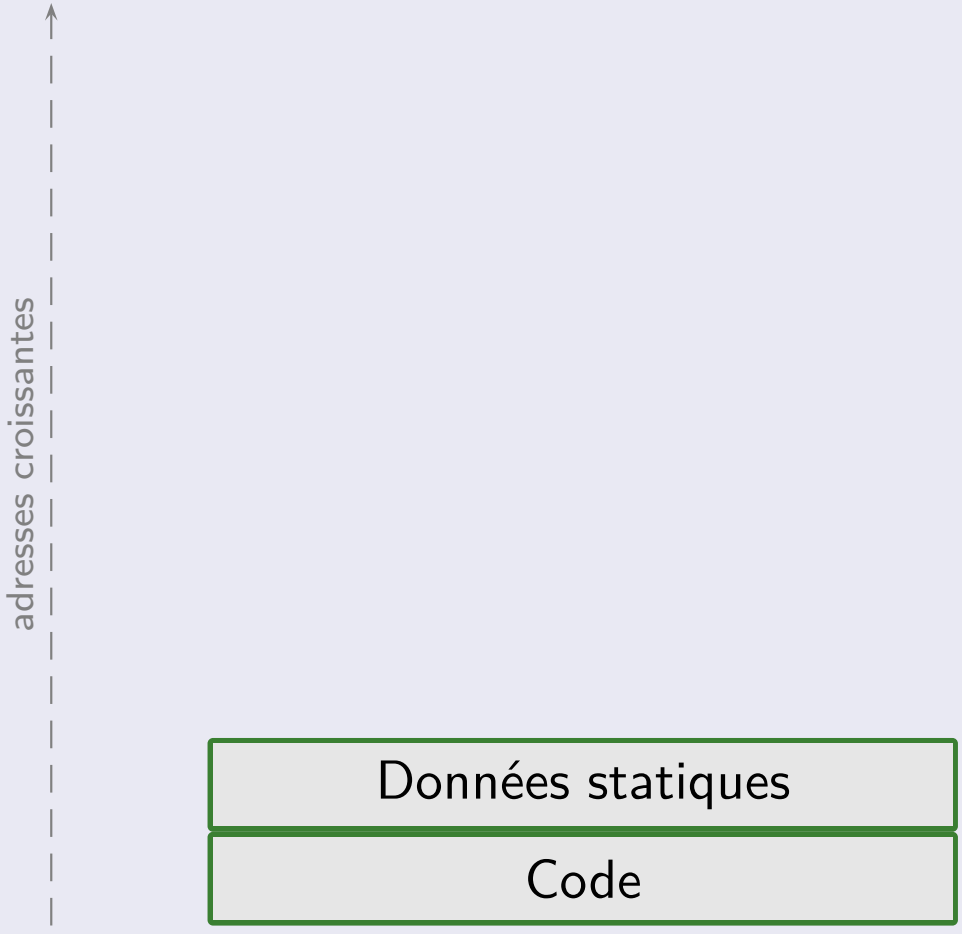
- En C, la gestion de la mémoire n'est pas totalement automatique (comme en Python ou en OCaml). Certains aspects reviennent au programmeur, ce qui impose de comprendre le modèle mémoire du C.

Schéma de l'organisation de la mémoire en C

Schéma de l'organisation de la mémoire en C

Schéma de l'organisation de la mémoire en C

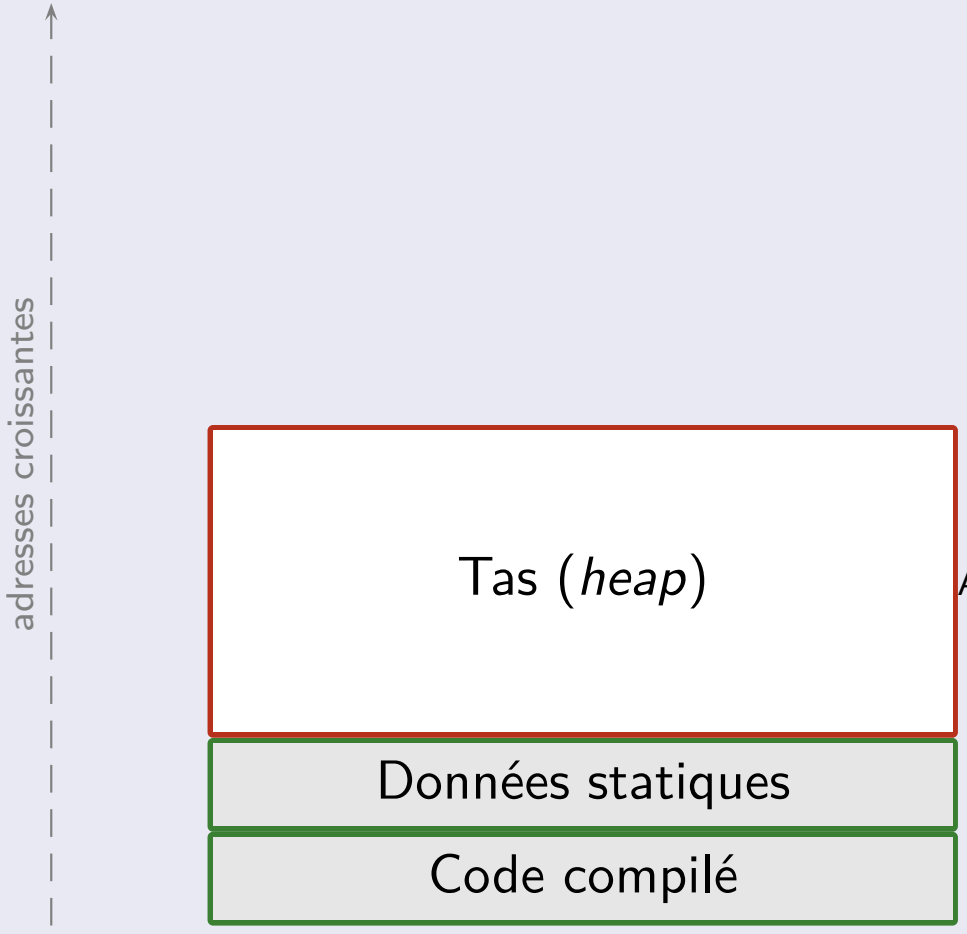
Schéma de l'organisation de la mémoire en C



Taille connue à la compilation
initialisées ou non.

En lecture seule

Schéma de l'organisation de la mémoire en C

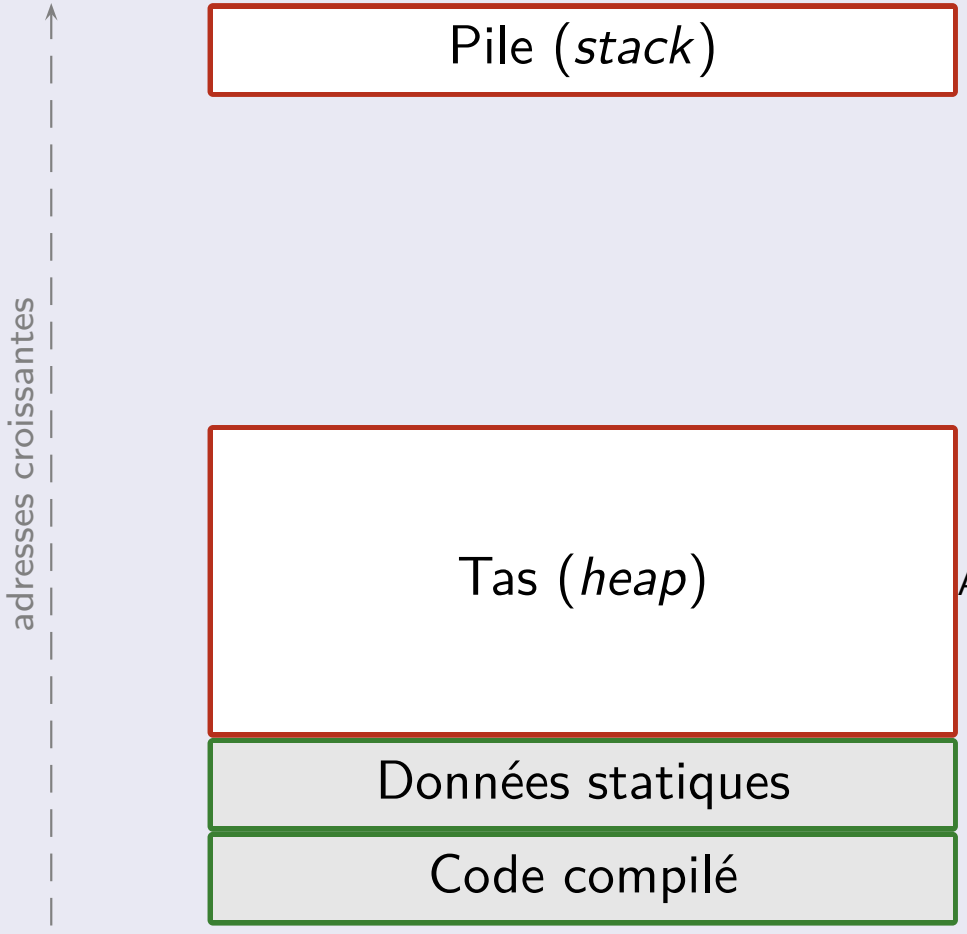


Taille variable
Allouée dynamiquement par le programmeur avec `malloc` et libéré avec `free`

Taille connue à la compilation
initialisées ou non.

En lecture seule

Schéma de l'organisation de la mémoire en C



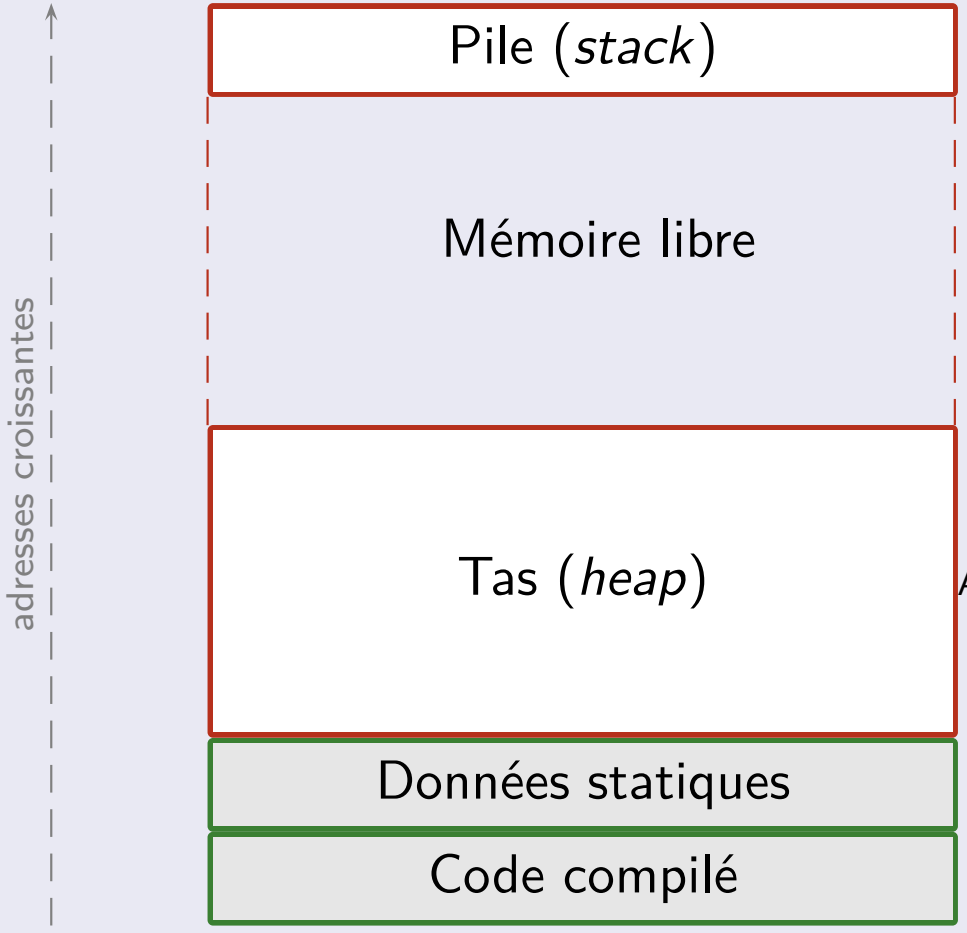
Empile les contextes d'appel des fonctions (variables locales, adresse de retour, ...)

Taille variable
Allouée dynamiquement par le programmeur avec `malloc` et libéré avec `free`

Taille connue à la compilation
initialisées ou non.

En lecture seule

Schéma de l'organisation de la mémoire en C



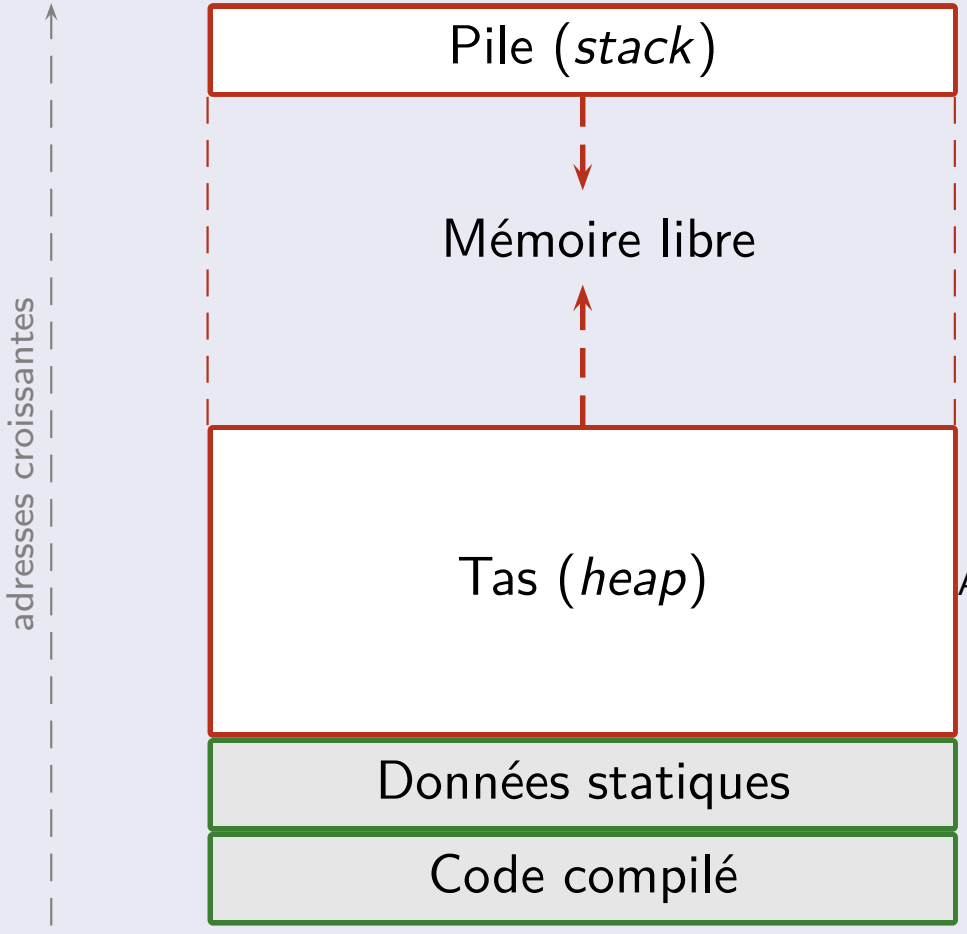
Empile les contextes d'appel des fonctions (variables locales, adresse de retour, ...)

Taille variable
Allouée dynamiquement par le programmeur avec `malloc` et libéré avec `free`

Taille connue à la compilation
initialisées ou non.

En lecture seule

Schéma de l'organisation de la mémoire en C



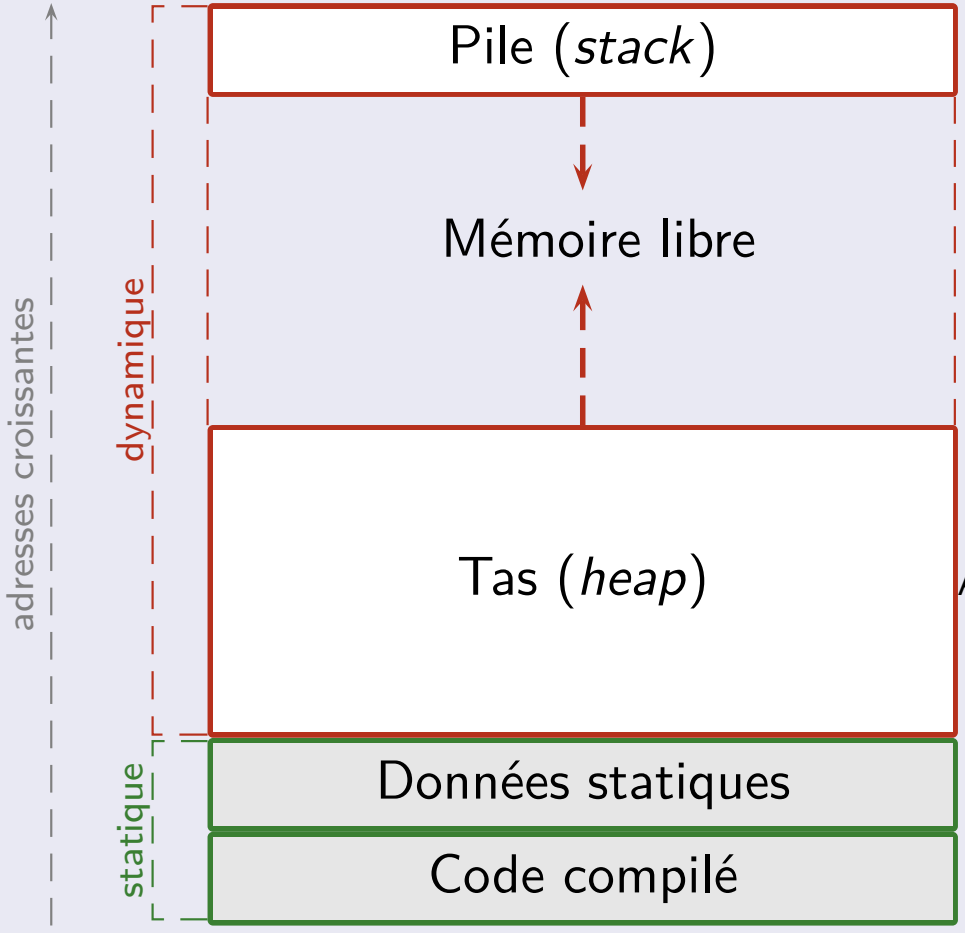
Empile les contextes d'appel des fonctions (variables locales, adresse de retour, ...)

Taille variable
Allouée dynamiquement par le programmeur avec `malloc` et libéré avec `free`

Taille connue à la compilation
initialisées ou non.

En lecture seule

Schéma de l'organisation de la mémoire en C



Empile les contextes d'appel des fonctions (variables locales, adresse de retour, ...)

Taille variable
Allouée dynamiquement par le programmeur avec `malloc` et libéré avec `free`

Taille connue à la compilation
initialisées ou non.

En lecture seule

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*).

Un premier exemple

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

1. Rappeler la taille d'un `double`, en déduire la taille du tableau `big_array`

Un premier exemple

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

1. Rappeler 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 premier exemple

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

1. Rappeler 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 d'appel sur cet ordinateur.

Un premier exemple

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

1. Rappeler 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 d'appel sur cet ordinateur.
4. Comment résoudre ce problème ?

Un premier exemple

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

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

Un premier exemple

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

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

Un premier exemple

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

1. Rappeler la taille d'un `double`, en déduire la taille du tableau `big_array`
   Un double occupe 8 octects, donc ce tableau $8 \times 1,5 = 12$ Mo.
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 ?
   La taille du tableau dépasse celle de la pile sur laquelle il est alloué.
3. 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)
4. Comment résoudre ce problème ?

Un premier exemple

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

1. Rappeler la taille d'un `double`, en déduire la taille du tableau `big_array`
   Un double occupe 8 octects, donc ce tableau $8 \times 1,5 = 12$ Mo.
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 ?
   La taille du tableau dépasse celle de la pile sur laquelle il est alloué.
3. 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)
4. Comment résoudre ce problème ?
   La mémoire occupée par le tableau doit être alloué sur le tas.