Algorithmique et structures de données 2

Chapitre 2

Memory check et révisions S1

1. Correction TP1

2. Fuites mémoire
   - Valgrind
   - Analyse de programme

3. Retour ASD1
   - Révision notions algorithmique
   - Révisions notions de C
     - Passage de paramètres

Tri à bulles

i couvre le tableau de 1 à n, j couvre la partie basse

```
1  int tri_a_bulle(int T[], int n)
2  {
3      printf("~~~~~~~~~~~~~~ Tri ~~~~~~~~~~~~~~\n");
4      int temp;
5      for (int i=n ; i > 0 ; i--){
6          for(int j = 0 ; j < i-1 ; j++) {
7              printf("tri : j = %d\n", j);
8              if(T[j] > T[j+1]){
9                  echange(&T[j], &T[j+1]);
10             }
11         }
12     }
13     return 0;
14 }
```

Optimisation tri à bulles

Meilleur cas : linéraire

- tableau trié : un seul parcours à faire
- idée : interrompre le tri si le tableau est trié

Optimisation tri à bulles

Meilleur cas : linéraire

- tableau trié : un seul parcours à faire
- idée : interrompre le tri si le tableau est trié

```
1  int trie;
2  for (int i=n ; i > 0 ; i--){
3      trie = 1;
4      for(int j = 0 ; j < i-1 ; j++) {
5          printf("tri opti : j = %d\n", j);
6          if(T[j] > T[j+1]){
7              echange(&T[j], &T[j+1]);
8                  trie = 0;
9          }
10     }
11     if (trie == 1){
12         return 0;
13     }
14 }
```

Comparer le nombre d'itérations

```
int tableau[5] = {9,2,3,3,1} ;

tri_a_bulle(tableau,5); # affiche  0 1 2 3 0 1 2 0 1 0
tri_a_bulle_opti(tableau,5); # affiche 0 1 2 3
```

Comparer le nombre d'itérations

```
int tableau[5] = {9,2,3,3,1} ;

tri_a_bulle(tableau,5); # affiche  0 1 2 3 0 1 2 0 1 0
tri_a_bulle_opti(tableau,5); # affiche 0 1 2 3
```

```
int tableau[5] = {9,2,3,3,1} ;

tri_a_bulle_opti(tableau,5); # affiche 0 1 2 3 0 1 2 0 1 0
```

foo.c

```c
#include <stdio.h>

int foo(void){
        // déclaration cpt

        printf("Appel à foo() numéro %d\n", cpt);
        return 0;
}

int main(void){
        int i ;
        for (i = 0; i<10; i++){
                foo();
        }
}
```

1. Correction TP1

2. Fuites mémoire
   - Valgrind
   - Analyse de programme

3. Retour ASD1
   - Révision notions algorithmique
   - Révisions notions de C
     - Passage de paramètres

1. Correction TP1

2. Fuites mémoire
   - Valgrind
   - Analyse de programme

3. Retour ASD1
   - Révision notions algorithmique
   - Révisions notions de C
     - Passage de paramètres

Fuite mémoire / *memory leak* 🛸

Quel est le problème ?

- pas d'erreur ni de warning à la compilation
- pas de bug à l'exécution
- gdb ne signale pas la fuite

Risques

- ralentissement du système ⏳
- saturation de la mémoire 🚫

Fuite mémoire 🛸

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int fuite() {
    static int* pIntNull;
    static int* pInt;

    pIntNull = malloc(sizeof(int));
    pIntNull = NULL;

    pInt = malloc(sizeof(int));
    return 0;
}

int main()
{
    fuite();
    return 0;
}
```

/vælgrɪnd/

« [...] The "grind" is pronounced with a short 'i' -- ie. "grinned" (rhymes with "tinned") rather than "grined" (rhymes with "find").
Don't feel bad : almost everyone gets it wrong at first... »

1

1. « Fantasy painting of computer scientist with a computer trying to pass Valgrind, the main entrance to Valhalla », généré par `https://labs.openai.com`.

« Valgrind is the name of the main entrance to Valhalla (the Hall of the Chosen Slain in Asgard). Over this entrance there resides a wolf and over it there is the head of a boar and on it perches a huge eagle, whose eyes can see to the far regions of the nine worlds. Only those judged worthy by the guardians are allowed to pass through Valgrind. All others are refused entrance. »

Fuite mémoire 🛸 : l'outil valgrind (2002)

```
$ valgrind --leak-check=full ./a.out
==153317== HEAP SUMMARY:
==153317==     in use at exit: 8 bytes in 2 blocks
==153317==   total heap usage: 2 allocs, 0 frees, 8 bytes allocated
==153317==
==153317== 4 bytes in 1 blocks are definitely lost in loss record 2 of 2
==153317== at 0x4848899: malloc (in /usr/libexec/[...]-linux.so)
==153317== by 0x10915A: fuite (main.c :11) *
==153317== by 0x1091A1: main (main.c:19)
==153317==
==153317== LEAK SUMMARY:
==153317==    definitely lost: 4 bytes in 1 blocks
==153317==    indirectly lost: 0 bytes in 0 blocks
==153317==      possibly lost: 0 bytes in 0 blocks
==153317==    still reachable: 4 bytes in 1 blocks
==153317==         suppressed: 0 bytes in 0 blocks
```

* le numéro de ligne est accessible parce que la compilation a été réalisée avec l'option -g.

Fuite mémoire 🛸

valgrind : `https://doc.ubuntu-fr.org/valgrind`

```
$ valgrind [—options=valeur] ./a.out
```

Toutes les entrées / sorties mémoires sont analysées et vérifiées minutieusement en interceptant tous les appels à `malloc` et `free`

- exécute le programme
- dresse un diagnostic

Fuite mémoire

```
$ valgrind --leak-check=full --show-reachable=yes ./a.out
==154396== HEAP SUMMARY:
==154396==     in use at exit: 8 bytes in 2 blocks
==154396==   total heap usage: 2 allocs, 0 frees, 8 bytes allocated
==154396==
==154396== 4 bytes in 1 blocks are still reachable in loss record 1 of
    2
==154396==    at 0x4848899: malloc (in /usr/libexec/[...]-linux.so)
==154396==    by 0x10916B: fuite (main.c :12)
==154396==    by 0x1091A1: main (main.c:19)
==154396==
==154396== 4 bytes in 1 blocks are definitely lost in loss record 2 of 2
==154396==    at 0x4848899: malloc (in /usr/libexec/[...]-linux.so)
==154396==    by 0x10915A: fuite (main.c :9)
==154396==    by 0x1091A1: main (main.c:19)
==154396==
==154396== LEAK SUMMARY:
==154396==    definitely lost: 4 bytes in 1 blocks
==154396==    indirectly lost: 0 bytes in 0 blocks
==154396==      possibly lost: 0 bytes in 0 blocks
==154396==    still reachable: 4 bytes in 1 blocks
==154396==         suppressed: 0 bytes in 0 blocks
```

Fuite mémoire 🛸

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int fuite() {
    static int* pIntNull;
    static int* pInt;

    pIntNull = malloc(sizeof(int));
    pIntNull = NULL; // definitely lost

    pInt = malloc(sizeof(int)); // still reachable
    return 0;
}

int main()
{
    fuite();
    return 0;
}
```

Fuite mémoire 🛸

Solution ✅

```
$ gcc -g -Wall main.c
$ valgrind --leak-check=full --show-reachable=yes ./a.out
```

... et corriger le code.

Fuite mémoire 🛸

Libérer la mémoire avec `free`

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int fuite() {
    static int* pIntNull;
    static int* pInt;

    pIntNull = malloc(sizeof(int));
    pIntNull = NULL;

    pInt = malloc(sizeof(int));

    free(pIntNull); // ajout free
    free(pInt); // ajout free
    return 0;
}

int main()
{
    fuite();
    return 0;
}
```

Fuite mémoire

Libérer la mémoire avec `free`

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int fuite() {
    static int* pIntNull;
    static int* pInt;

    pIntNull = malloc(sizeof(int));
    pIntNull = NULL;

    pInt = malloc(sizeof(int));

    free(pIntNull); // que se passe-t-il ici ?
    free(pInt);
    return 0;
}

int main()
{
    fuite();
    return 0;
}
```

Fuite mémoire

```
$ valgrind --leak-check=full --show-reachable=yes ./a.out
==155231== HEAP SUMMARY:
==155231==     in use at exit: 4 bytes in 1 blocks
==155231==   total heap usage: 2 allocs, 1 frees, 8 bytes allocated
==155231==
==155231== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==155231==    at 0x4848899: malloc (in /usr/libexec/[...]-linux.so)
==155231==    by 0x10917A: fuite (main.c:9)
==155231==    by 0x1091DF: main (main.c:21)
==155231==
==155231== LEAK SUMMARY:
==155231==    definitely lost: 4 bytes in 1 blocks
==155231==    indirectly lost: 0 bytes in 0 blocks
==155231==      possibly lost: 0 bytes in 0 blocks
==155231==    still reachable: 0 bytes in 0 blocks
==155231==         suppressed: 0 bytes in 0 blocks
```

The free function causes the space pointed to by ptr to be deallocated, that is, made available for further allocation. If ptr is a null pointer, no action occurs.

(source C standard, 7.20.3.2/2 from ISO-IEC 9899 : lien)

Pointeur NULL

Dans quel cas un pointeur peut-il valoir `NULL` ?

Pointeur NULL

Dans quel cas un pointeur peut-il valoir `NULL` ?

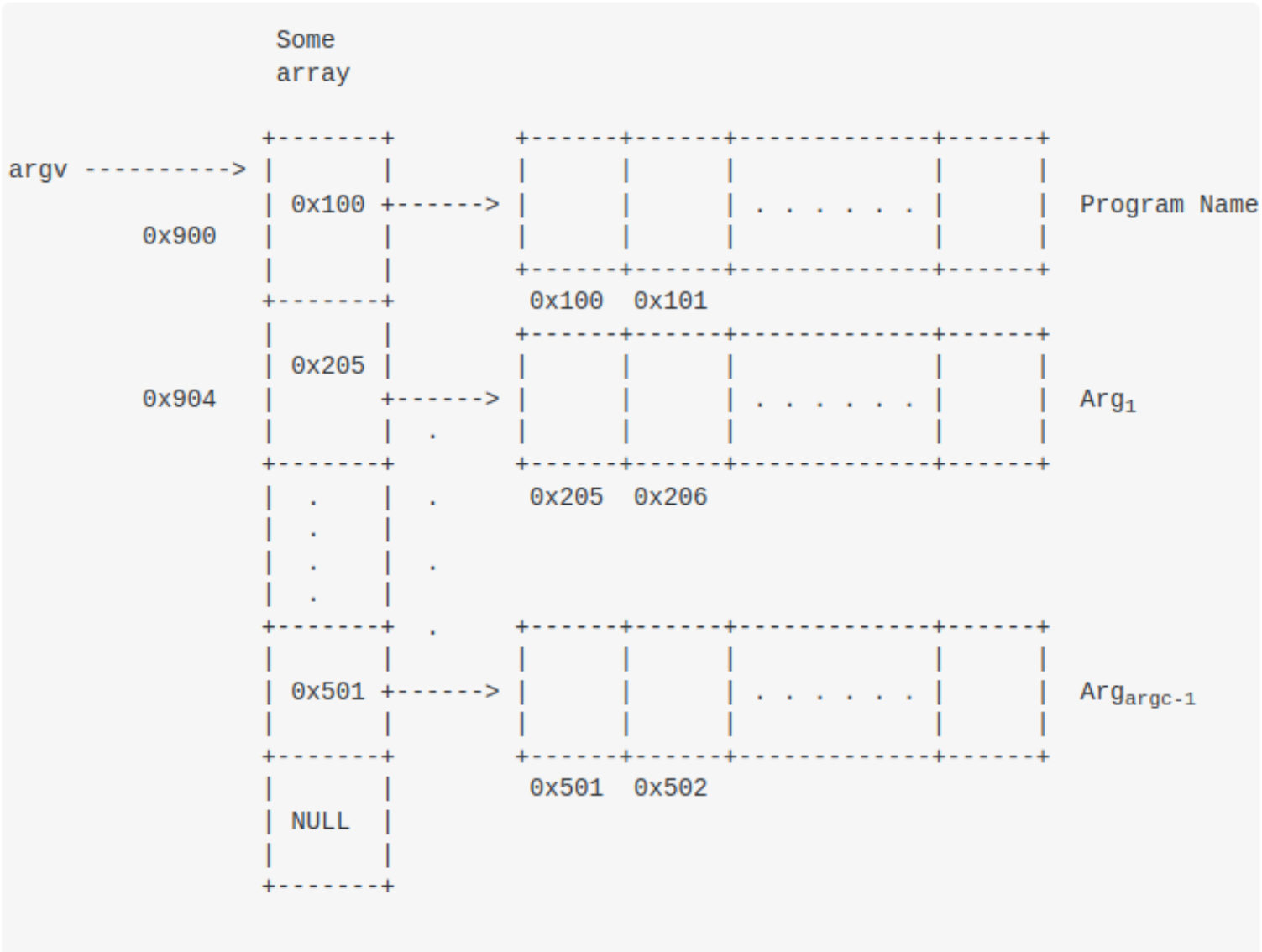
- initialisation à `NULL` : une variable non initialisée peut avoir un contenu *indéterminé* $\rightarrow$ il **faut** initialiser ses pointeurs ;
- échec d'allocation ;

1. Correction TP1

2. Fuites mémoire
   - Valgrind
   - Analyse de programme

3. Retour ASD1
   - Révision notions algorithmique
   - Révisions notions de C
     - Passage de paramètres

argc / argv



0. https://stackoverflow.com/questions/39095850/what-is-the-type-of-command-line-argument-argv-in-c

Analyse de programme : `leak.c`

```
1 int main(int argc, char** argv)
2 {
3     char tab[8];
4     printf("%i argument(s)\n", argc);
5     if (argc > 1){
6             strcpy(tab, argv[1]);
7             printf("Argument 1 : %s\n", tab);
8     }
9 }
```

- que fait ce programme ?

Analyse de programme : leak.c

```c
int main(int argc, char** argv)
{
    char tab[8];
    printf("%i argument(s)\n", argc);
    if (argc > 1){
            strcpy(tab, argv[1]);
            printf("Argument 1 : %s\n", tab);
    }
}
```

- que fait ce programme ?

```
$ ./a.out hello
2 argument(s)
Argument 1 : hello
```

🥪 Take away 🥤

1. quel est l'**outil de débuggage** le mieux adapté en fonction du type de bug ?
   - `gcc` : suivre pas à pas l'exécution du programme
   - `valgrind` : diagnostiquer les fuites mémoires
2. **pointeurs et bonnes pratiques**
   - initialiser à `NULL` si valeur inconnue
   - test valeur avant utilisation
   - appel à `malloc` pour l'initialisation
   - appel à `free` pour libérer l'espace mémoires

1. Correction TP1

2. Fuites mémoire
   - Valgrind
   - Analyse de programme

3. Retour ASD1
   - Révision notions algorithmique
   - Révisions notions de C
     - Passage de paramètres

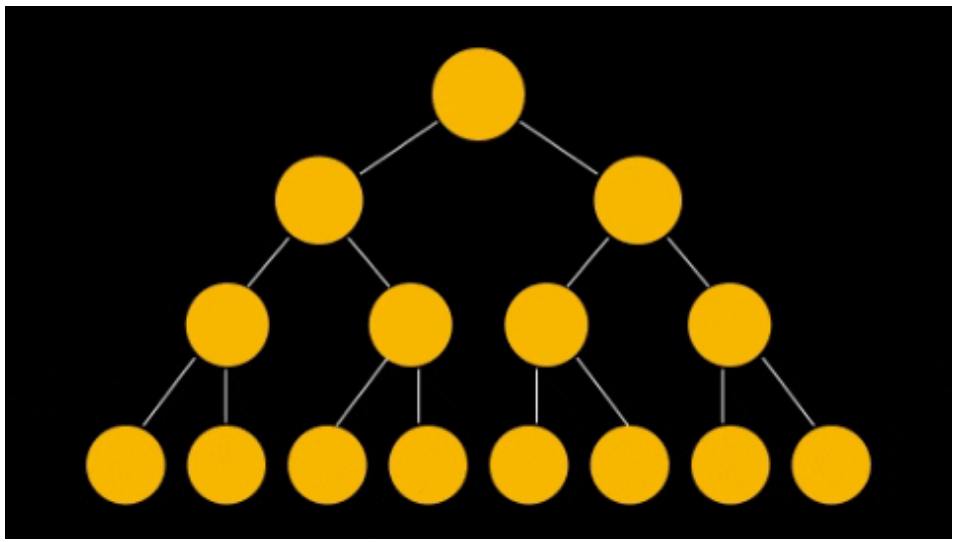
1. Correction TP1

2. Fuites mémoire
   - Valgrind
   - Analyse de programme

3. Retour ASD1
   - Révision notions algorithmique
   - Révisions notions de C
     - Passage de paramètres

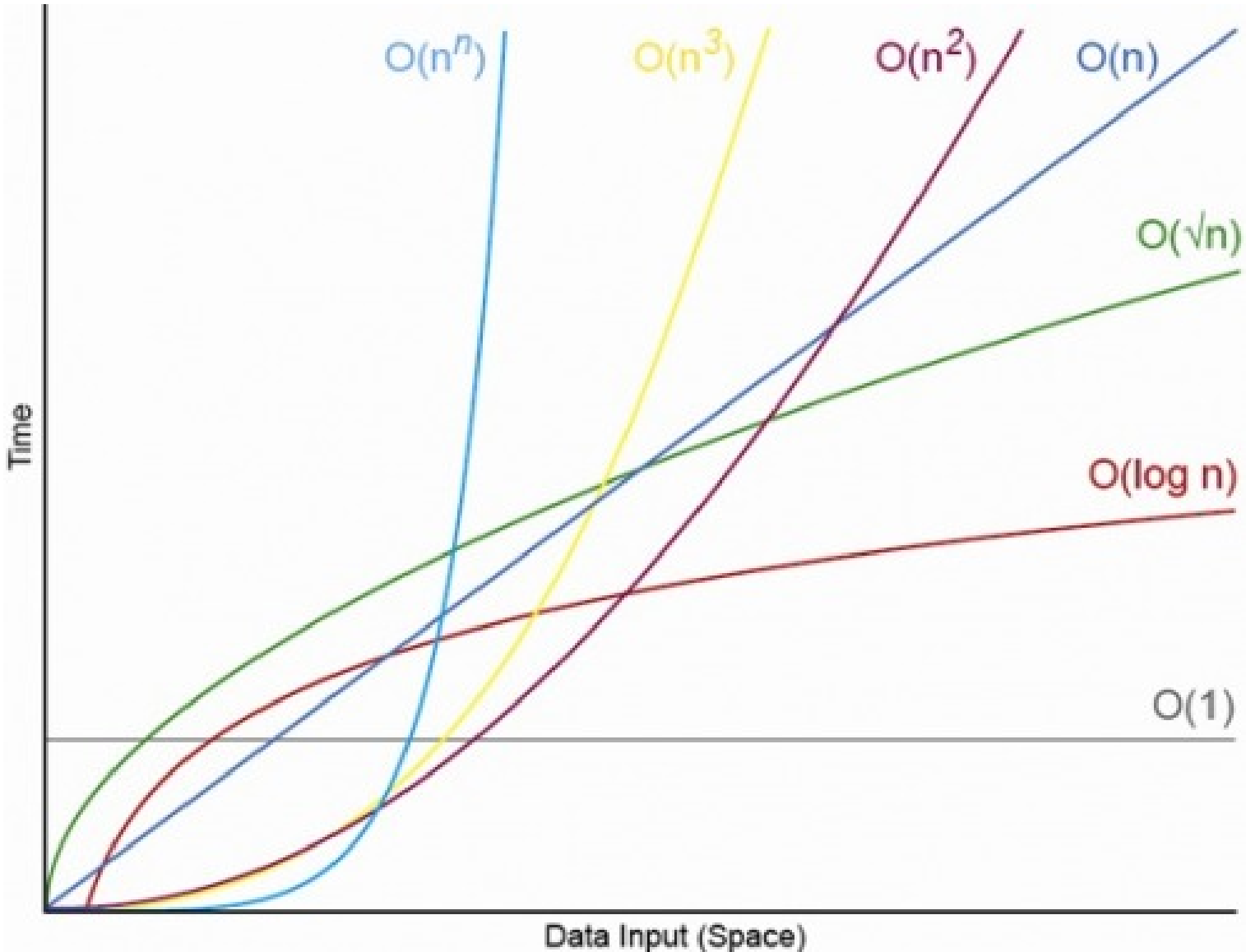
Parcours en profondeur vs largeur

Depth *first* // Breadth *first*



Comparaison de suites

Qui croît plus vite que qui ?



1. Correction TP1

2. Fuites mémoire
   - Valgrind
   - Analyse de programme

3. Retour ASD1
   - Révision notions algorithmique
   - Révisions notions de C
     - Passage de paramètres

Différents modes de passages des paramètres

- Passage par valeur : la **valeur** de l'expression passée en paramètre est copiée dans une variable locale.
- Passage par variable : la **variable** elle-même est passée en paramètre.

  conséquence :

  *Toute **modification du paramètre** dans la fonction appelée entraîne la **modification de la variable** passée en paramètre*.

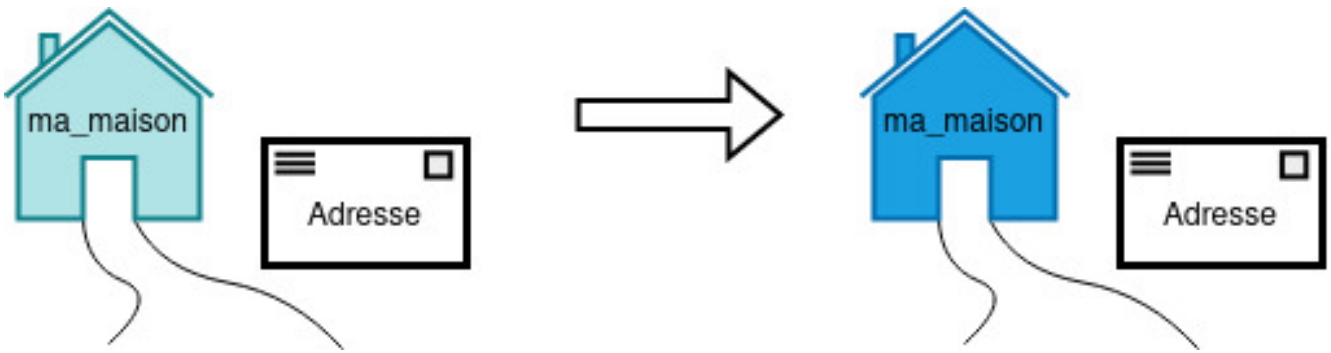
Différents modes de passages des paramètres

- Passage par valeur : la **valeur** de l'expression passée en paramètre est copiée dans une variable locale.
- Passage par variable : la **variable** elle-même est passée en paramètre.
  conséquence :
  *Toute **modification du paramètre** dans la fonction appelée entraîne la **modification de la variable** passée en paramètre*.

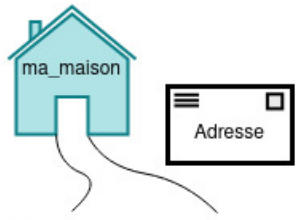
C n'autorise **pas** le passage par variable

- Passage par référence : l'**adresse** de la variable est passée en paramètre.

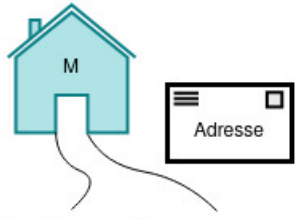
Exemple : repeindre une maison



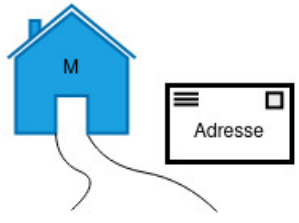
Passage de paramètre par VARIABLE



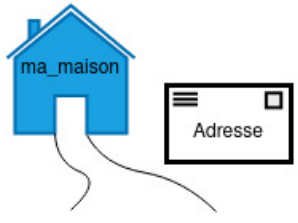

```
def peindre(M, bleu) :
```



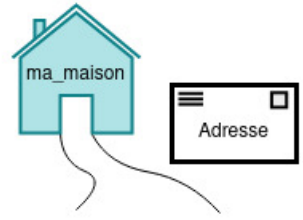

```
M.couleur = bleu
```



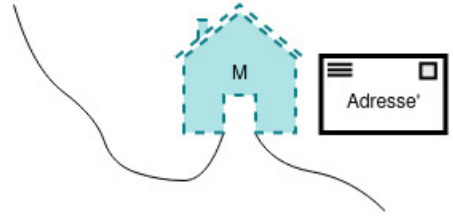

```
peindre(ma_maison, bleu)
```



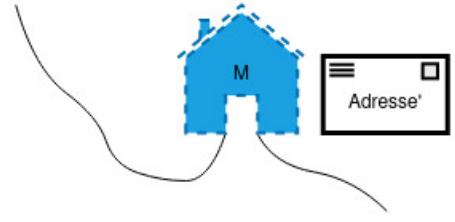
Passage de paramètre par VALEUR



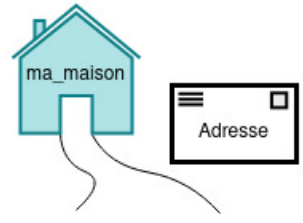

```
void peindre(Maison M, Couleur bleu) {
```




```
M.couleur = bleu
```

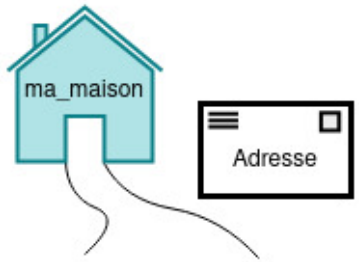



```
}
peindre(ma_maison, bleu) ;
```

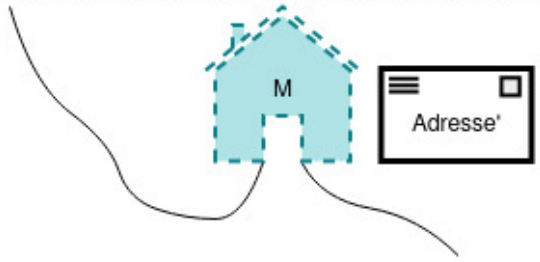


Passage de paramètre par VALEUR

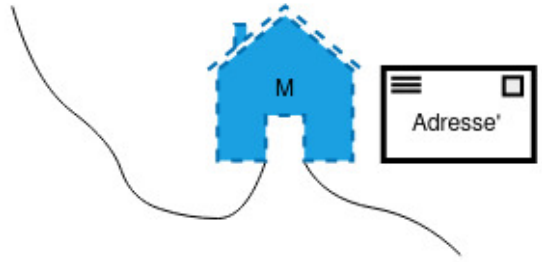
une solution



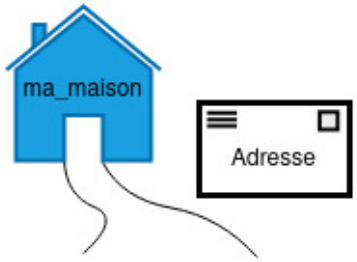

```
Maison peindre(Maison M, Couleur bleu) {
```



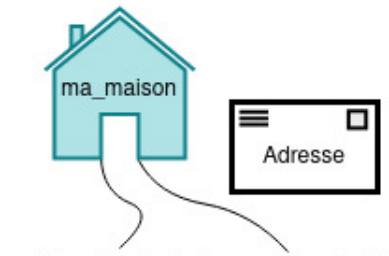

```
M.couleur = bleu;
```



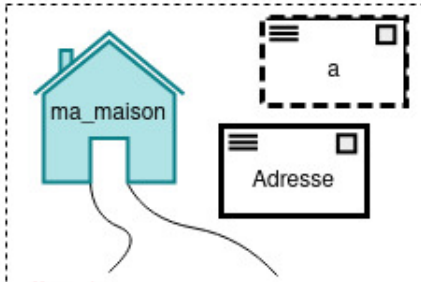

```
return M ;
}
ma_maison = peindre(ma_maison, bleu)
```



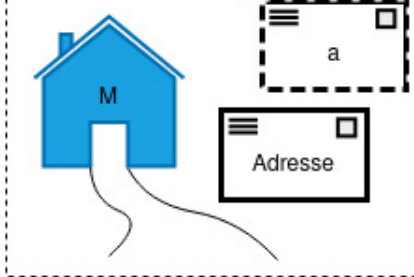
Passage de paramètre par REFERENCE



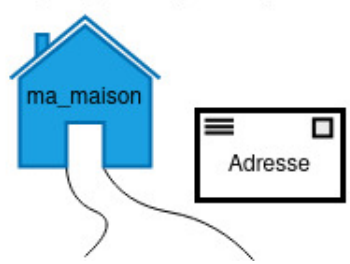

```
void peindre(adresse a, Couleur bleu) {
```




```
M = *a
M.couleur = bleu;
```




```
}
peindre(&ma_maison, bleu)
```



Passage par valeur *vs* passage par référence

- Quel est l'avantage du passage par référence par rapport au passage par valeur en C ?

Passage par valeur *vs* passage par référence

- Quel est l'avantage du passage par référence par rapport au passage par valeur en C ?

Le passage par valeur permet de transmettre une (ou plusieurs) valeur(s) de retour **supplémentaire(s)** dans une fonction.

Exemple ?

Passage par valeur *vs* passage par référence

- Quel est l'avantage du passage par référence par rapport au passage par valeur en C ?

Le passage par valeur permet de transmettre une (ou plusieurs) valeur(s) de retour **supplémentaire(s)** dans une fonction.

Exemple ?

```
1 int scanf("%i", &premier); // 2 val de retour
2 int scanf("%i %i", &premier, &deuxieme); // 3 val de retour
3 int scanf("%i %i %i", &premier, &deuxieme, &troisieme); // 4 val de
      retour
```