

# TD 2 : Mémoire

Algorithmes et programmation.

2023

Dans tout le TD, on supposera, pour simplifier que l'espace requis en mémoire pour encoder les différents types est le suivant :

Type	Taille (nombre d'octets)
int	1
long	2
float	1
double	2
char	1
void *	1

## Exercice 1 — *Quelques exemples*

Représentez ce qu'il se passe en mémoire si on exécute les codes suivants.

### 1. Manipulations classiques

```
1 void test (int* p, int* r){
2     int a;
3     double b;
4     int *q;
5     a = 42;
6     b = 73.5;
7     p = &a;
8     q = (int*) malloc(sizeof(int));
9     *p = b + a ;
10    *q = *p + 5;
11    *r = *p + *q;
12    q = &a;
13    p = (int*) malloc(sizeof(int));
14    *p = *q + a;
15    *r = *r + *p;
16 }
17 void main(void){
18     int x;
19     test(&x, &x);
20     printf("%p %d", &x, x);
21 }
22
```

## 2. Chaînes, pointeurs et tableaux.

```
1  #define N 3
2
3  void test(char** t){
4      for(int i = 0; i < N; i++){
5          char c[4];
6          if(i % 3 == 0){
7              c[0] = 'a';
8              c[1] = 'b';
9              c[2] = 'c';
10             c[3] = '\\0';
11             t[i] = c;
12         }
13         else if(i % 3 == 1){
14             t[i] = "def";
15         }
16         else{
17             c[0] = 'g';
18             c[1] = 'h';
19             c[2] = 'i';
20             c[3] = '\\0';
21             t[i] = c;
22         }
23     }
24 }
25
26 int main(void){
27     char ** t = malloc(N * sizeof(char*));
28     test(t);
29     for(int i = 0; i < N; i++){
30         printf("t[%d] = %s\\n", i, t[i]);
31     }
32     free(t);
33     return 0;
34 }
35
36
37
38
```

Comment corriger ce programme ?

### 3. Structures, pointeurs et realloc.

```
1  #define N1 2
2  #define N2 4
3
4  struct s_rec{
5      long value;
6      struct s_rec * first;
7  };
8  typedef struct s_rec rec;
9
10 int main(void){
11
12     rec* t = (rec *) malloc(N1 * sizeof(rec));
13     for(int i = 0; i < N1; i++){
14         t[i].value = 0;
15         t[i].first = t;
16     }
17     malloc(sizeof(int));
18     t = realloc(t, N2 * sizeof(rec));
19     for(int i = N1; i < N2; i++){
20         t[i].value = i;
21         t[i].first = t;
22     }
23
24     for(int i = 0; i < N2; i++)
25         t[i].first->value = 165449;
26     *t.value = 134789;
27
28     for(int i = 0; i < N2; i++)
29         printf("%d : %d\n", i, t[i].first->value);
30
31     return 0;
32 }
33
34
35
36
```

Comment corriger ce programme ?

## Exercice 2 — Défragmentation

1. On suppose qu'on dispose d'une mémoire de  $2N$  octets sur le tas (la zone où sont alloués les espaces réservés par `malloc`).

Que se passe-t-il avec le code suivant ? Représentez le tas en mémoire.

```
1  int* p[2*N];
2  for(int i = 0; i < 2*N ; i++){
3      p[i] = (int*) malloc(sizeof(int));
4  }
5  for(int i = 0; i < 2*N ; i = i + 2){
6      free(p[i]);
7  }
8  double * q = (double*) malloc(sizeof(double));
9
```

2. Supposons que le système soit capable de défragmenter : si un `malloc` devait renvoyer `NULL` par manque de place, les pointeurs précédemment alloués sont repositionnés sur le tas de sorte à mettre tout l'espace disponible à la fin.  
Représentez le tas en mémoire à l'issue du code précédent.
3. En supposant la défragmentation effectuée, sur quoi pointe `p[1]`. En déduire que ce modèle de défragmentation n'est pas utilisable dans des langages qui manipulent sans contrainte des variables de type pointeur.
4. On veut implémenter une fonction `malloc2` et une fonction `free2` qui utiliseront `malloc` et `free` et qui défragmentent quand il y a besoin. Pour simplifier on suppose qu'on ne travaille qu'avec des entiers. On fait les hypothèses suivantes :
  - En haut du fichier sont définis, avec un `define`, deux tailles `S` et `P`.
  - On dispose d'un tableau global `t` de `P` pointeurs sur entiers.
  - On dispose d'un tableau global `size` de `P` entiers; où `size[i]` contient le nombre d'entiers sur lesquels pointe `t[i]`.
  - On dispose d'un entier global `s` qui donne la taille effective de `t` et `size`.
  - Un pointeur de `t` ne peut pointer sur plus de `S` entiers.
  - (a) Implantez un algorithme `int* malloc2(int size)` qui alloue la mémoire pour un pointeur sur `size` entiers, si `size <= S`, et ajoute ce pointeur à `t`. S'il n'y a pas la place nécessaire, on appelle une fonction qui défragmente la mémoire avant de réessayer.
  - (b) Implantez un algorithme `void free2(int* p)` qui libère la mémoire d'un pointeur allouée par la fonction `malloc2`.

On fait maintenant les hypothèses suivantes :

- On suppose qu'aucune autre fonction du programme ne fait appel à `malloc` ou à `free`.
- On suppose que `malloc` attribue toujours la première place disponible en mémoire : l'adresse la plus petite permettant d'allouer un bloc mémoire de la taille demandée.
- On dispose d'une fonction `sort` qui trie le pointeurs de `t` de la plus petite à la plus grande adresse.

- (c) Implantez un algorithme `void defragmente()` qui défragmente la mémoire.