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

```c
void test (int* p, int* r){
  int a;
  double b;
  int *q;
  a = 42;
  b = 73.5;
  p = &a;
  q = (int*) malloc(sizeof(int));
  *p = b + a ;
  *q = *p + 5;
  *r = *p + *q;
  q = &a;
  p = (int*) malloc(sizeof(int));
  *p = *q + a;
  *r = *r + *p;
}
void main(void){
  int x;
  test(&x, &x);
  printf("%p %d", &x, x);
}

```

▶ **Correction**

La correction est donné avec le fichier corr1.

2. Chaînes, pointeurs et tableaux.

```c
#define N 3

void test(char** t){
  for(int i = 0; i < N; i++){
    char c[4];
    if(i % 3 == 0){
      c[0] = 'a';
      c[1] = 'b';
      c[2] = 'c';
      c[3] = '\0';
      t[i] = c;
    }
    else if(i % 3 == 1){
      t[i] = "def";
    }
    else{
      c[0] = 'g';
      c[1] = 'h';
      c[2] = 'i';
      c[3] = '\0';
      t[i] = c;
    }
  }
}

int main(void){
  char ** t = malloc(N * sizeof(char*));
  test(t);
  for(int i = 0; i < N; i++){
    printf("t[%d] = %s\n", i, t[i]);
  }
  free(t);
  return 0;
}
```

Comment corriger ce programme ?

▶ Correction

La correction est donné avec le fichier corr2. Pour corriger, il faudrait par exemple créer `c` ou `t[0]`, `t[1]` et `t[2]` avec un `malloc` pour s'assurer que la zone mémoire occupée par ces cases ne soit pas réutilisée.

3. Structures, pointeurs et realloc.

```c
#define N1 2
#define N2 4

struct s_rec{
  long value;
  struct s_rec * first;
};
typedef struct s_rec rec;

int main(void){

  rec* t = (rec *) malloc(N1 * sizeof(rec));
  for(int i = 0; i < N1; i++){
    t[i].value = 0;
    t[i].first = t;
  }
  malloc(sizeof(int));
  t = realloc(t, N2 * sizeof(rec));
  for(int i = N1; i < N2; i++){
    t[i].value = i;
    t[i].first = t;
  }

  for(int i = 0; i < N2; i++)
    t[i].first->value = 165449;
    *t.value = 134789;

  for(int i = 0; i < N2; i++)
    printf("%d : %d\n", i, t[i].first->value);

  return 0;
}
```

Comment corriger ce programme ?

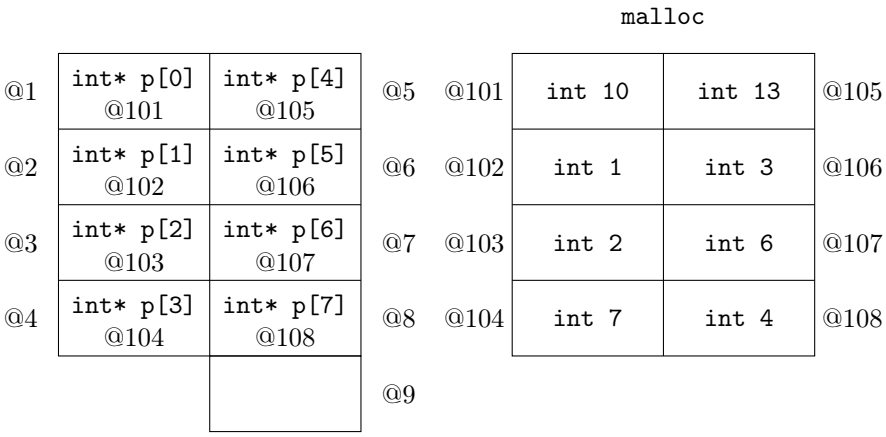
Exercice 2 — *Défragmentation*

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.

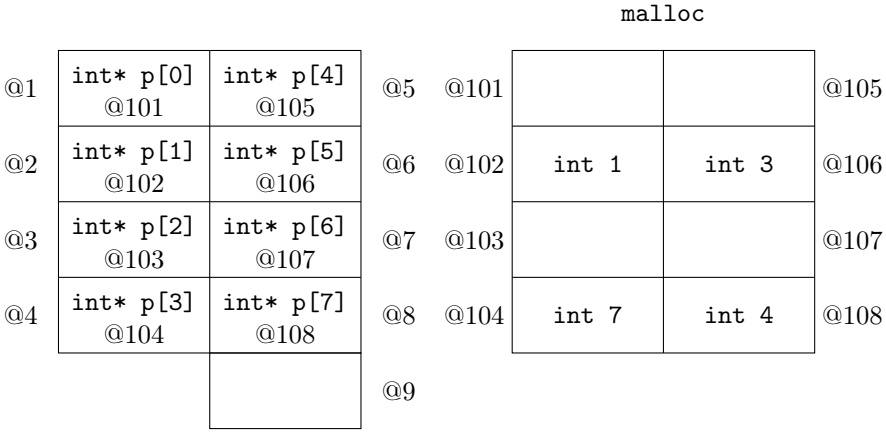
```
1       int* p[2*N];
2       for(int i = 0; i < 2*N ; i++){
3         p[i] = (int*) malloc(sizeof(int));
4       }
5       for(int i = 0; i < 2*N ; i = i + 2){
6         free(p[i]);
7       }
8       double * q = (double*) malloc(sizeof(double));
9
```

▶ **Correction**

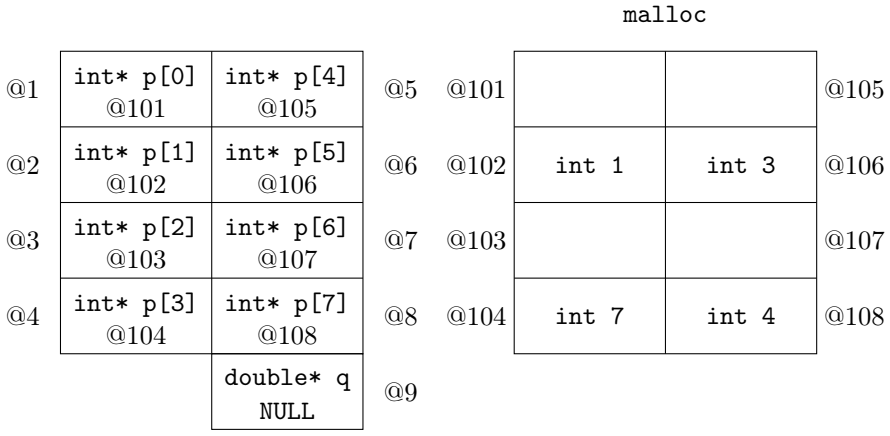
Voici un exemple avec $N = 4$. On représente ici toutes les zones, y compris celle contenant p et q.
Au début, on réserve la mémoire (4 premières lignes). Les valeurs des entiers de la zone `malloc` sont quelconques (et auraient pu être remplacés par `???`, des valeurs sont données explicitement pour expliquer la question suivante).



On libère la mémoire (lgines 5, 6, et 7). (Les valeurs sont effacées sur le dessin, mais, dans la mémoire, ce qui était présent dans les cases @101, @103, @105 et @107 reste tant qu'on ne les a pas écrasées ; par contre les cases ne sont plus réservées).



On réserve de la place pour q. Mais comme q est un pointeur sur `double`, il faut réserver 2 cases consécutives. On ne dispose pas de telle place mémoire, la réservation ne se fait donc pas.



On réserve $2N$ cases pour $2N$ pointeurs sur des entiers. Puis on libère, une case sur 2.

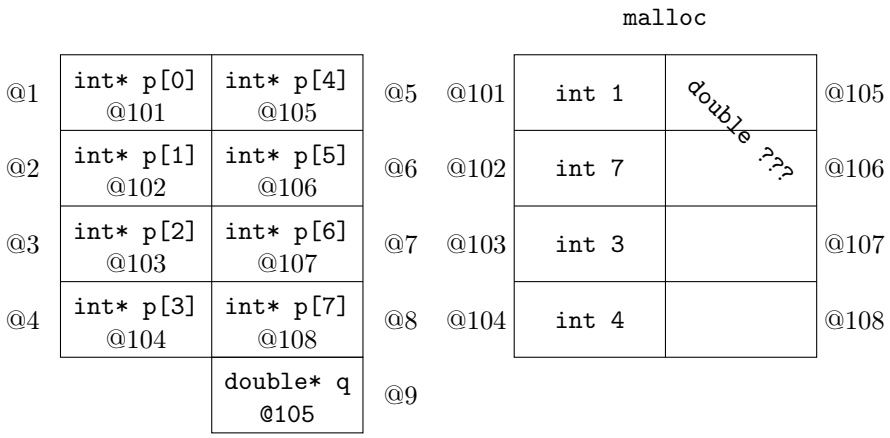
Lorsqu'on libère la mémoire, on se retrouve avec N cases libérées, pourtant, il n'est pas possible de trouver de la place pour y mettre un `double`. En effet, la valeur pointée par q prend 2 cases et il n'y a aucun endroit disponible avec 2 cases consécutives. Donc le malloc de la dernière ligne renvoie NULL.

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.

► Correction

On aurait donc la mémoire suivante :



On peut remarquer que p n'a pas changé, on en discute à la question suivante.

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.

► Correction

`p[2*N-1]` pointe sur @108. La valeur de cette case n'a pas changée, car aucune autre opération n'a modifié le contenu de cette case en mémoire. Cependant, la case est libre, on pourrait donc

écraser cette valeur ultérieurement, ce qui changerait la valeur `*p[2*N-1]`. Autre problème, on voit que `p[0], p[1], p[2], p[3]` ont eux changé de valeur puisqu'on a réécrit le contenu des cases entre les adresse `@101` et `@104`.

Il y a un petit problème avec cette défragmentation. Elle suppose qu'on puisse librement déplacer tout le contenu des cases réservées par `malloc`. Mais, problème, ces cases ont une adresse qui est pointée par des variables de type pointeur (ici `p` et `q`). On peut voir que `p` n'est plus du tout cohérent avec les cases sur lesquelles il pointait avant. Changer l'emplacement de ces cases implique donc de devoir modifier l'adresse pointée par les cases de `p` et par `q`.

On pourrait imaginer une solution assez naturelle, consistant à dire que, lorsqu'un `malloc` est effectuée, on enregistre l'information correspondant au pointeur où a été rangée l'adresse réservée par le `malloc`. Par exemple ici, après la défragmentation, la valeur de `p[1]` devrait être changée en `@101` au lieu de `@102`.

Cependant, on peut manipuler ce pointeur librement : on peut copier l'adresse contenue dans un pointeur vers un autre pointeur, il faudrait donc aussi changer ce nouveau pointeur. Si par exemple on exécute une ligne `int* r = p[1]` avant la défragmentation, alors `r` contient l'adresse `@102`. Après la défragmentation, on devrait aussi changer `r` en `@101`.

Pire, on peut écraser l'adresse contenue dans un pointeur avec une autre adresse, dans ce cas, il ne faut pas qu'une défragmentation touche à l'adresse contenue dans ce pointeur. Par exemple si on a, avant la défragmentation les lignes `int* r = p[1]` et `p[1] = &x`, et qu'on défragmente, il ne faut pas mettre `p[1]` à l'adresse `@101` (par contre il faut le faire pour `r`).

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.

   ▶ **Correction**

   Pour la défragmentation, on va utiliser la fonction `defragmente` qui est demandée à la dernière question.

```
1   int* malloc2(int msize){
2     // On verifie s'il reste de la place dans t et size et
    si msize n'est pas trop grand
3     if(s >= P)
4       return NULL;
5     if(msize > S)
6       return NULL;
7
8     // On cree le pointeur avec malloc
9     int * p = (int*)malloc(size*sizeof(int));
10
11    // On verifie s'il y a de la place pour p, sinon on
   defragmente et on recommence.
12    // Quand il n'y a plus de place, malloc renvoie NULL
13    if(p == NULL){
14      defragmente();
15      p = (int*)malloc(size*sizeof(int));
16    }
17
18    // Si malloc renvoie toujours NULL, alors il n'y a
   plus de place du tout, (malgre la defragmentation).
19    if(p == NULL){
20      return p;
21    }
22
23    // On enregistre p dans t et msize dans size
24    t[s] = p;
25    size[s] = msize;
26    s = s + 1;
27
28    return p;
29  }
30
```

(b) Implantez un algorithme `void free2(int* p)` qui libère la mémoire d'un pointeur allouée par la fonction `malloc2`.

► **Correction**

```
int* free2(int* p){
  // On commence par rechercher p dans la liste t
  int index = -1;
  for(int i = 0; i < s; i++){
    if(t[i] == p){
      index = i;
      break;
    }
  }

  // Si p n'est pas dans t, il n'a pas ete cree avec malloc2
  if(index == -1){
    return;
  }

  // On libere p
  free(p);

  // On efface la case t[i] et on decale toutes les cases t[j] et size[j], pour j > i, vers le haut.
  for(int i = index; i < s - 1; i++){
    t[i] = t[i + 1];
    size[i] = size[i + 1];
  }
  s = s - 1;
}

```

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.

▶ **Correction**

```
void defragmente(){
  // On trie les pointeurs de t
  sort();

  // On peut maintenant liberer chaque pointeur un par un, puis reallouer la memoire en debut de zone.
  // Puisque t[0] est la plus petite adresse des pointeurs de t, alors il n'y a pas de risque a deplacer t[0] vers une adresse plus petite. Une fois qu'il est deplace, on peut deplacer t[1], et ainsi de suite.

  // On va utiliser un tableau intermediaire pour stocker les pointeurs le temps de defragmenter.
  int save[S];

  // Pour chaque pointeur
  for(int i = 0; i < s; i++){

    // On sauvegarde le pointeur dans save
    for(int j = 0; j < size[i]; j++)
      save[j] = t[i][j];

    // On deplace le pointeur (on le libere et on le re-reserve)
    free(t[i]);
    t[i] = malloc(size[i] * sizeof(int));

    // On utilise la sauvegarde pour redonner la bonne valeur aux cases pointees par le pointeur.
    for(int j = 0; j < size[i]; j++)
      t[i][j] = save[j];
  }

}

```

Attention, on voit que cette solution ne règle pas tous les problèmes. En particulier, si un pointeur de t est copié, il n'est pas garanti que la copie pointe au bon endroit suite à une défragmentation. Aussi, il ne faut pas écraser les pointeurs pointés par t ou les pointeurs renvoyés par `malloc2`.