

# TP4: Correction

MP2I Lycée Pierre de Fermat  
guillaume.rousseau@ens-lyon.fr

## Exercice 1

L'archive du TP contient deux fichiers `quarante.txt` et `quarante.pas.txt`. Le premier contient le nombre 40 encodé en ASCII, c'est à dire le caractère '4' suivi du caractère '0'. Le second contient le nombre 40 encodé en entier signé sur 4 octets. L'archive contient également le code `ecrire_40.c` utilisé pour générer ce fichier. Il utilise des fonctions que l'on n'a pas encore vu, mais vous pouvez déjà l'ouvrir et constater que l'on recopie bien le contenu d'un `int` valant 40.

- Q1.** Les trois octets `0x34`, `0x30` et `0x0a` correspondent respectivement au code ASCII du caractère '4', au code ASCII du caractère '0', et au code ASCII du retour à la ligne.
- Q2.** La commande `od -t x1 -A x quarante.pas.txt` affiche que le fichier contient **quatre** octets : `0x28`, `0x00`, `0x00`, et `0x00`. `0x28` vaut  $2 \times 16 + 8 = 40$ , on reconnaît donc l'écriture de 40 sur 4 octets, en **petit-boutiste**.

## Exercice 2

Effectivement, le fichier a bien été créé. Si jamais on inspecte son contenu avec la commande `od` de l'exercice précédent, on y voit les codes ASCII des différents caractères. Il est important de noter que lorsque l'on écrit un entier  $n$  avec le format `%d`, on n'écrit pas l'encodage de  $n$ , mais la suite des encodages individuels de ses chiffres en base 10. En particulier, si l'on avait écrit un plus grand nombre comme 123456, on aurait bien vu 6 octets et pas 4.

### Exercice 3

Un exemple de main permettant de tester la fonction :

```
1 int main(){
2     char nom_fichier[51];
3     // récupérer le nom du fichier
4     printf("Veuillez entrer un nom de fichier:\n");
5     scanf("%s", nom_fichier);
6
7     int s = somme2(nom_fichier);
8     printf("Le fichier %s contient deux entiers "
9           "de somme %d\n", nom_fichier, s);
10
11     return 0;
12 }
```

Pour tester le programme, on peut aussi créer un fichier texte contenant deux entiers :

```
49 150
```

le programme lit bien les deux entiers et affiche leur somme, 199.

### Exercice 4

On adapte le main de l'exercice précédent :

```
1 int main(){
2     char nom_fichier[51];
3     // récupérer le nom du fichier
4     printf("Veuillez entrer un nom de fichier:\n");
5     scanf("%s", nom_fichier);
6
7     int s = somme(nom_fichier);
8     printf("Le fichier %s contient des entiers "
9           "de somme %d\n", nom_fichier, s);
10
11     return 0;
12 }
```

Puis on crée un fichier contenant plusieurs entiers :

```
45 10
63
47 12 -5
```

On peut alors tester la fonction : elle lit bien dans le fichier, entier par entier, et s'arrête lorsque la fin du fichier est atteinte. La somme affichée est 172, ce qui est juste.

## Exercice 5

- Q1.** Pour la fonction `int premier_zero(char* filename)`, on peut commencer par reprendre le code de la fonction `somme` de l'exercice précédent, et l'adapter. On peut directement mettre le `fscanf` en condition de boucle, ce qui simplifie un peu la logique du code : si l'on rentre dans le corps de la boucle, c'est que la lecture a réussi et donc que `x` existe bien :

```

1  /* Renvoie le nombre d'entiers strictement positifs lus dans le fichier
2     `nom_fichier` avant d'y lire un 0. Renvoie -1 si le fichier ne contient
3     aucun 0.
4     Précondition: `nom_fichier` ne doit contenir que des entiers. */
5  int premier_zero(char* nom_fichier){
6     FILE* f = fopen(nom_fichier, "r"); // ouverture en mode lecture
7     assert(f != NULL);
8     int x = 0; // entier lu dans le fichier
9     int tot_positif = 0; // nombre d'entiers > 0 lus
10    while (fscanf(f, "%d", &x) != EOF){
11        if (x == 0){
12            fclose(f);
13            return tot_positif;
14        } else if (x > 0){
15            tot_positif++;
16        }
17    }
18    // si l'on sort de la boucle, c'est qu'aucun 0 n'a été croisé
19    fclose(f);
20    return -1;
21 }

```

Notons que dans cette fonction, il y a deux endroits dans le code où l'on peut `return` et sortir de la fonction : il faut penser à fermer le fichier ouvert avant **chacun** des `return` !

Dans le main, en plus d'une version interactive du programme où l'on demande un nom de fichier à l'utilisateur et où l'on affiche le résultat, on peut écrire une version "test", où l'on appelle la fonction sur plusieurs fichiers qui ont été créés au préalable :

5 33 1 816 8 0 48 12 0 3

FIGURE 1 – Fichier `test1.txt`

9 -5 7 -2 6 0

FIGURE 2 – Fichier `test2.txt`

60 42 -12 17 198 -5

FIGURE 3 – Fichier `test3.txt`

Les trois fichiers servent à tester la fonction dans les cas suivants :

- que des entiers positifs suivi d'un 0, et éventuellement d'autres données après, pour vérifier que le programme s'arrête au premier 0 ;
- un mélange d'entiers positifs et négatifs suivi d'un 0, pour vérifier que le programme ne compte pas les entiers négatifs ;
- pas de 0, pour vérifier que le programme renvoie bien -1 comme demandé.

D'où la fonction main :

```

1  int main(){
2      // TESTS
3      assert(premier_zero("test1.txt") == 5);
4      assert(premier_zero("test2.txt") == 3);
5      assert(premier_zero("test3.txt") == -1);
6
7      // MODE INTERACTIF
8
9      char nom_fichier[51];
10     // récupérer le nom du fichier
11     printf("Veuillez entrer un nom de fichier:\n");
12     scanf("%s", nom_fichier);
13
14     int p = premier_zero(nom_fichier);
15     if (p == -1){
16         printf("Le fichier ne contient pas de 0\n");
17     } else {
18         printf("Le fichier contient %d entiers > 0 avant le premier 0\n", p);
19     }
20     return 0;
21 }
```

**Q2.** Commençons par écrire la fonction `void renverser(char* s)` suggérée par l'énoncé, qui renverse le sens d'une chaîne donnée. Pour cela, notons  $L$  la longueur de  $s$ . Il faut échanger :

- $s[0]$  et  $s[L - 1]$  ;
- $s[1]$  et  $s[L - 2]$  ;
- ...
- $s[i]$  et  $s[L - 1 - i]$  ;
- ...

en s'arrêtant une fois avoir traité la moitié du mot, i.e. quand  $i \geq \frac{L}{2}$ . Utilisons la librairie `<string.h>` :

```

1  /* Renverse la chaîne s */
2  void renverser(char* s){
3      int l = strlen(s);
4      for (int i = 0; i < l/2; ++i){
5          char temp = s[l-i-1];
6          s[l-i-1] = s[i];
7          s[i] = temp;
8      }
9  }
```

On teste dans le main : ici pour choisir des bons tests, on peut prendre une chaîne de longueur impaire et une de longueur paire :

```

1  char s1[40] = "bonjour";
2  char s2[40] = "UnTexteDansLeBonSens";
3  renverser(s1);
4  assert(strcmp(s1,"ruojnob") == 0);
5
6  renverser(s2);
7  assert(strcmp(s2,"sneSnoBeLsnaDetxeTnU") == 0);

```

On en déduit une fonction permettant de renverser chaque mot individuel d'un fichier : il suffit de lire mot par mot avec `%s`, de renverser, et de réécrire dans le fichier de sortie :

```

1  /* Renverse chaque mot de in_fn, et écrit dans out_fn */
2  void renverser_fichier(char* in_fn, char* out_fn){
3      FILE* in_f = fopen(in_fn, "r");
4      FILE* out_f = fopen(out_fn, "w");
5      assert(in_f != NULL);
6      assert(out_f != NULL);
7
8      char buf[51]; // mot lu
9      while (fscanf(in_f, "%s", buf) != EOF){
10         renverser(buf);
11         fprintf(out_f, "%s ", buf);
12     }
13     fclose(out_f);
14     fclose(in_f);
15 }

```

## Exercice 6

- Q1.** On voit bien que le programme affiche chacun des arguments qu'on lui fournit. De plus, il affiche systématiquement son propre nom : on peut donc en conclure que `argv[0]` est le nom de l'exécutable.
- Q2.** En suivant l'indication du sujet : on lit caractère par caractère, en recopiant chaque caractère lu depuis le fichier d'entrée directement dans le fichier de sortie :

```

1  /* Copie in_fn dans out_fn */
2  void copieur(char* in_fn, char* out_fn){
3      FILE* in_f = fopen(in_fn, "r");
4      FILE* out_f = fopen(out_fn, "w");
5      assert(in_f != NULL);
6      assert(out_f != NULL);
7      char c; // caractère lu
8      while (fscanf(in_f, "%c", &c) != EOF){
9         fprintf(out_f, "%c", c);
10     }
11     fclose(out_f);
12     fclose(in_f);
13 }

```

Puis, dans le main, il suffit d'appeler cette fonction avec les deux premiers arguments du programme. Si l'utilisateur n'a pas fourni assez d'arguments, on affiche une erreur :

```

1  int main(int argc, char** argv){
2      if (argc < 3){

```

```
3     printf("Veuillez fournir deux noms de fichiers.\n");
4     return 0;
5 }
6 copieur(argv[1], argv[2]);
7 return 0;
8 }
```

## Exercice 7

Problème d'énoncé : il y a deux questions Q1...

**Q1.** Il faut vérifier les tableaux case par case, on ne peut pas se contenter de tester si `t1 == t2`. En effet, `t1 == t2` teste l'égalité des adresses mémoires, mais pas le contenu des tableaux!

```
1 /* Renvoie true si les n premiers éléments
2    de t1 et t2 sont égaux */
3 bool egaux(int* t1, int* t2, int n){
4     for (int i = 0; i < n; ++i){
5         if (t1[i] != t2[i]){
6             return false;
7         }
8     }
9     return true;
10 }
```

**Q2.** On peut remplir le tableau avec une boucle :

```
1 int* q2 = malloc(5*sizeof(int));
2 for (int i = 0; i < 5; ++i){
3     q2[i] = i;
4 }
5 free(q2);
```

**Q3.** La fonction :

```
1 /* Renvoie un tableau de taille n, nul */
2 int* zeros(int n){
3     int* t = malloc(n * sizeof(int));
4     for (int i = 0; i < n; ++i){
5         t[i] = 0;
6     }
7     return t;
8 }
```

Un exemple de test unitaire dans le main :

```
1 int* q3 = zeros(7);
2 int test_q3[7] = {0, 0, 0, 0, 0, 0, 0};
3 assert(egaux(q3, test_q3, 7));
```

**Q4.** La fonction :

```
1 /* Renvoie un tableau de n false puis m true */
2 bool* zeros_uns(int n, int m){
3     bool* res = malloc((n + m) * sizeof(int));
4     for (int i = 0; i < n; ++i){
5         res[i] = false;
```

```

6   }
7   for (int i = n; i < n+m; ++i){
8       res[i] = true;
9   }
10  return res;
11 }

```

Attention, si on veut tester cette fonction comme la précédente, on ne peut pas directement utiliser la fonction `egaux` car les types ne sont pas compatibles. On peut créer une autre fonction identique mais avec des `bool*` en entrée :

```

1  /* Renvoie true si les n premiers éléments
2     de t1 et t2 sont égaux */
3  bool egalx_bool(bool* t1, bool* t2, int n){
4      for (int i = 0; i < n; ++i){
5          if (t1[i] != t2[i]){
6              return false;
7          }
8      }
9      return true;
10 }

```

On pourrait techniquement utiliser la fonction `egalx_bool` pour tester l'égalité entre des tableaux d'`int` car le type `bool` fait exactement un octet : un tableau de  $n$  `bool` fait donc la même taille qu'un tableau de  $4n$  `bool`, et tester l'égalité d'un tableau revient à tester l'égalité de tous les octets.

On pourrait aussi utiliser `egaux` pour les tableaux de booléens, mais seulement s'ils ont une taille multiple de 4 !

## Exercice 8

- Q1.** Le problème de ce fichier est que la mémoire allouée pour `t` ne fait que 4 octets, alors qu'on en veut  $4 \times 4 = 16$ . Pour corriger, il suffit d'utiliser `sizeof` pour modifier la première ligne :

```

1  int* t = malloc(4 * sizeof(int));

```

- Q2.** L'erreur ici vient du fait qu'à la ligne 8, on essaie d'accéder à `*p`, qui est une zone mémoire qui a été libérée juste avant. Il faut échanger l'ordre des deux dernières lignes, afin de libérer les deux zones dans un ordre cohérent. Notons encore une fois que l'ordre correspond à un bon parenthésage :

```

1  int main(){
2      int** p = malloc(sizeof(int*));
3      *p = malloc(sizeof(int));
4      **p = 75;
5      free(*p);
6      free(p);
7  }

```

- Q3.** Ici, pas d'erreur à l'exécution, mais on a une fuite mémoire, car le premier `malloc` n'a jamais été `free`. Il suffit de libérer la mémoire avant de faire le deuxième `malloc` :

```

1  int main(){
2      float* a = malloc(sizeof(float));

```

```

3   *a = 5;
4   free(a);
5   a = malloc(2*sizeof(float));
6   a[0] = 5;
7   a[1] = 7;
8   free(a);
9 }

```

Si l'on dessinait la mémoire dans le programme original, on verrait que la zone mémoire correspondant au premier malloc est "orpheline" : plus aucune variable de la pile ne permet d'y accéder.

**Q4.** Il manque le return à la fin de la fonction, et le free à la fin du main !

```

1  /* Renvoie un tableau de n entiers aléatoires entre 1 et 100 */
2  int* random_tab(int n){
3      int* t = malloc(n * sizeof(int));
4      for (int i = 0; i < n; ++i){
5          t[i] = rand()%100 + 1;
6      }
7      return t;
8  }
9
10 int main(){
11     int* t = random_tab(10);
12     for (int i = 0; i <= 10; ++i){
13         printf("%d\n", t[i]);
14     }
15     free(t);
16 }

```

Ce qu'il faut retenir de cet exercice, c'est que si l'on exécute les 4 programmes sans utiliser valgrind, ils ont de fortes chances de fonctionner sans problème **apparent**. Pourtant, ils ont tous des erreurs qui risquent de les faire dysfonctionner. Utiliser valgrind pour tester régulièrement les fonctions que vous écrivez permet de repérer ces erreurs invisibles et d'éviter de se retrouver à devoir les gérer plus tard lorsque votre programme a pris de l'ampleur.

Les messages de valgrind sont cryptiques au début, mais en vous entraînant à l'utiliser, vous verrez qu'ils sont très informatifs !

## Exercice 9

Dans cet exercice, on veut lire dans un fichier jusqu'à ce que l'une des deux conditions suivantes soit atteinte :

- On a lu  $n$  entiers
- On a atteint la fin du fichier

Autrement dit, on lit **tant que** les deux conditions sont fausses. Cela donner une solution possible :

```

1  /* Renvoie un tableau d'au plus *n entiers lus
2     dans le fichier filename. Stocke dans *n le
3     nombre d'entiers effectivement lus. */
4  int* lire_entiers(char* filename, int* n){
5      FILE* f = fopen(filename, "r");

```



```
6  assert(f != NULL);
7  int i = 0; // nb d'entiers lus
8  int* t = malloc(*n * sizeof(int)); // entiers lus
9
10 bool fini_de_lire = false;
11 while(!fini_de_lire && i < *n){
12     int nb_lus = fscanf(f, "%d", &t[i]);
13     if (nb_lus == EOF){
14         fini_de_lire = true; // termine la boucle
15     } else {
16         i++;
17     }
18 }
19 // en sortie: soit on a atteint EOF, soit on a lu n entiers.
20 // Dans tous les cas, i contient le nombre d'entiers lus
21 *n = i;
22 fclose(f);
23 return t;
24 }
```

Un point important : quand vous ouvrez un fichier, pensez à **vérifier qu'il n'est pas NULL** : si le fichier est NULL, c'est que l'ouverture a échoué (typiquement, vous avez mal écrit le nom du fichier, ou vous ne vous trouvez pas au même endroit dans le terminal et vous n'avez pas mis le chemin menant au fichier).

Dans les tests, on peut penser à trois cas à tester :

- un cas où la fonction atteint la fin du fichier avant d'avoir lu  $n$  entiers;
- un cas où la fonction lit  $n$  entiers avant d'avoir atteint la fin du fichier;
- un fichier vide.

(Voir archive du corrigé).

Une remarque : dans la solution proposée, on renvoie un tableau de la taille demandée par l'utilisateur, y compris si on a lu beaucoup moins d'entiers. On pourrait imaginer ajouter quelques instructions à la fin de la fonction pour allouer un deuxième tableau faisant pile la bonne taille, et y recopier les valeurs :

```
1  // réallouer un autre tableau faisant pile la bonne taille
2  int* res = malloc(i*sizeof(int));
3  for (int j = 0; j < i; ++j){
4      res[j] = t[j];
5  }
6  free(t);
7  fclose(f);
8  return res;
```

Vous pouvez aussi vous renseigner sur la fonction `realloc` (en tapant `man realloc` dans le terminal par exemple, ou en regardant sur internet) et voir comment on aurait pu l'utiliser dans cet exercice !

## Exercice 10

Les fonctions des premières questions ne sont pas trop compliquées, vous pouvez regarder dans l'archive du corrigé pour voir les solutions proposées : ce sont des implémentations directes des formules mathématiques pour la somme et du produit.

Pour l’affichage, on peut afficher chaque coefficient avec `%d`, mais on peut aussi utiliser le format `%6d`, qui fonctionne comme `%d` mais oblige l’affichage à prendre 6 espaces. Ceci permet d’aligner les cases verticalement, que les coefficients prennent 1 ou 6 chiffres, ce qui rend l’affichage plus lisible.

Pour la question 6, une solution possible est de fournir en entrée de la fonction deux pointeurs qui seront modifiés, de la même manière que l’on modifie `n` dans l’exercice 9 :

```
1 void min_moy(int** g, int n, int m, int* pi0, int* pj0){
2     int i0 = 0;
3     int j0 = 0;
4     ...
5     *pi0 = i0;
6     *pj0 = j0;
7 }
```

Une autre possibilité est de renvoyer un tableau de taille 2 dans lequel on stocke les valeurs de  $i_0$  et  $j_0$ , l’inconvénient étant qu’on alloue de la mémoire dans le tas et qu’il ne faut pas oublier de la libérer plus tard.

## Exercice 11

Il y avait quelques erreurs d’énoncé :

- La fonction `getline` renvoie `EOF` si elle est appelée alors que la tête de lecture est à la fin du fichier, ce n’est pas précisé dans le sujet.
- La fonction prend en entrée un **unsigned long int**, mais dans l’exemple de trois lignes donnés, c’est un **int** qui est utilisé : ça devrait être un unsigned long int.

**Q1.** Si le premier argument était de type `char*`, ce serait un pointeur vers une zone de la mémoire déterminée, où un nombre fixé de cases ont été réservées via `malloc`. La fonction ne pourrait pas réallouer de mémoire. En mettant un `char**`, la fonction peut trouver de la mémoire autre part. Un exemple :

```
1 void realloue(int** p){
2     free(*p);
3     *p = malloc(100 * sizeof(int));
4 }
5
6 int main(){
7     int* p = malloc(2 * sizeof(int));
8     realloue(&p);
9 }
```

**Q2.** Puisque `getline` conserve le caractère de retour à la ligne, si on veut éviter d’afficher une ligne blanche entre chaque ligne, on doit remplacer le `'n'` par un `'0'`. A part ça, la difficulté principale de la question vient de l’utilisation de la fonction `getline`, qui est assez complexe : elle lit dans un fichier, modifie des arguments ET renvoie une valeur !

```
1 int main(int argc, char** argv){
2     assert(argc >= 2);
3     char* nom_fichier = argv[1];
4     FILE* f = fopen(nom_fichier, "r");
```

```

5
6 char* buffer = NULL;
7 unsigned long int n = 0;
8
9 bool fini = false;
10 int i = 1; // numéro de la ligne
11 while(!fini){
12     int len = getline(&buffer, &n, f);
13     if (len == EOF){
14         fini = true;
15     } else {
16         // enlever l'éventuel \n de fin
17         if (buffer[len-1] == '\n'){
18             buffer[len-1] = '\0';
19         }
20         printf("%d. %s\n", i, buffer);
21         i++;
22     }
23 }
24
25 free(buffer);
26 fclose(f);
27 return 0;
28 }

```

Notons que dans la solution proposée, le buffer est **réutilisé** : si on lit une ligne de taille 20 puis une ligne de taille 10, la fonction `getline` n'aura pas besoin de réallouer de la mémoire, car elle sait quelle taille fait la zone mémoire qu'on lui donne (grâce à l'argument `&n`). Ceci permet aussi de n'avoir qu'à mettre un seul `free`, à la toute fin de programme. On voit alors un avantage au fait que l'on a le droit de `free(NULL)` : le programme fonctionnera même pour un fichier vide !

**Q3.** On ne connaît pas à priori le nombre de lignes d'un fichier, mais on peut écrire une fonction qui le calcule, en reprenant la question précédente :

```

1 /* Renvoie le nombre de lignes de `nom_fichier` */
2 int compter_lignes(char* nom_fichier){
3     char* buffer = NULL;
4     unsigned long int n = 0;
5     FILE* f = fopen(nom_fichier, "r");
6     int i = 0;
7     bool fini = false;
8     while(!fini){
9         int len = getline(&buffer, &n, f);
10        if (len == EOF){
11            fini = true;
12        } else {
13            i++;
14        }
15    }
16    fclose(f);
17    free(buffer);
18    return i;
19 }

```

Ensuite, on peut écrire une fonction qui renvoie un tableau contenant toutes les lignes d'un fichier :

```

1 /* Renvoie un tableau contenant toutes les lignes de `nom_fichier`.

```

```

2   Modifie n pour y mettre le nombre de lignes lues */
3 char** toutes_lignes(char* nom_fichier, int* n){
4   *n = compter_lignes(nom_fichier);
5   FILE* f = fopen(nom_fichier, "r");
6   char** res = malloc(*n * sizeof(char*));
7   int i = 0;
8   bool fini = false; // EOF atteint ?
9   while(!fini){
10    char* buffer = NULL;
11    unsigned long int n = 0;
12    int len = getline(&buffer, &n, f);
13    if (len == EOF){
14      fini = true;
15      free(buffer);
16    } else {
17      // enlever l'éventuel \n de fin
18      if (buffer[len-1] == '\n'){
19        buffer[len-1] = '\0';
20      }
21      res[i] = buffer;
22      i++;
23    }
24  }
25  }
26  fclose(f);
27  return res;
28 }

```

Notons qu'il ne faut rien free ici, car on veut conserver les chaînes lues et les stocker dans le tableau renvoyé! Il faut tout de même free la toute dernière zone, qui est réservée par l'appel à `getline` qui atteint la fin du fichier (la fonction `getline` semble systématiquement commencer par réserver 120 octets au début de son exécution, même si rien n'est lu ensuite).

Un très bon exercice : dessinez la mémoire au cours de l'exécution de la fonction `toutes_lignes`!

Une fois qu'on a la liste des lignes, il suffit d'appliquer un algorithme de tri. On en verra plein dans le prochain chapitre, celui utilisé dans la solution proposée est le tri par sélection : il consiste à chercher le plus grand élément du tableau à trier, et à le mettre à la fin, puis chercher le plus grand élément restant, à le mettre à l'avant dernière case, et ainsi de suite :

```

1  /* trie t, tableau de taille n contenant des chaînes
2   de caractères. */
3 void trier(char** t, int n){
4   for (int i = 0; i < n; ++i){
5     // trouver le max de t[0], ..., t[n-1-i]
6     int i_max = 0;
7     for (int j = 0; j < n-i; ++j){
8       if (strcmp(t[j], t[i_max])>0){
9         i_max = j;
10      }
11    }
12    // échanger t[i_max] et t[n-1-i]
13    char* tmp = t[i_max];
14    t[i_max] = t[n-1-i];
15    t[n-1-i] = tmp;

```

```

16 }
17 }

```

**Q4.** Il y beaucoup d'aspects à gérer dans cette question car la fonction est assez complexe. Commençons par l'en-tête :

```

1 int ma_getline(char** buffer, unsigned long int* n, FILE* f){

```

En suivant les indications du sujet, on peut introduire quelques variables :

```

1 int i = 0; // nb d'octets lus
2 bool fini = false;
3 char c = '\0';
4 // Invariants:
5 // - *buffer a *n octets alloués
6 // - i octets ont été lus et stockés dans *buffer
7 // - c est le caractère lu à l'étape précédente

```

Dans la boucle, il faut bien penser à réallouer de la mémoire lorsque l'on atteint la taille maximale :

```

1 while(!fini && c != '\n'){
2     if (fscanf(f, "%c", &c) == EOF){
3         fini = true;
4     } else {
5         // augmenter la taille si nécessaire
6         if (i+1 >= *n){
7             *buffer = realloc(*buffer, 2 * (*n));
8             *n = 2 * (*n);
9         }
10        (*buffer)[i] = c;
11        i++;
12    }
13 }

```

Le +1 dans la condition pour augmenter la taille correspond au fait qu'il faudra mettre un caractère nul à la fin de la chaîne une fois la lecture finie.

En sortie de boucle, si le **premier** `fscanf` a renvoyé EOF, il faut que la fonction renvoie EOF, sinon elle renvoie le nombre d'octets lus :

```

1 (*buffer)[i] = '\0';
2 if (fini && i == 0){ // EOF atteint immédiatement
3     return EOF;
4 } else {
5     return i;
6 }
7 }

```

**Q5.** Pour que la fonction fonctionne dans les deux cas de figure, c'est assez simple : si en entrée de la fonction le buffer n'est pas alloué, on lui alloue une taille initiale (le sujet indique 30, la vraie fonction getline semble allouer 120) :

```

1 if (*n == 0 && *buffer == NULL){
2     *buffer = malloc(30*sizeof(char));
3     *n = 30;
4 }

```

On peut alors tester en remplaçant les appels à getline dans les programmes des questions précédentes par notre version homemade : tout fonctionne pareil !

**Exercice 12**

Voir archive du corrigé.

---