TP4: Fichiers et allocation dynamique

MP2I Lycée Pierre de Fermat

Consignes

Ce TP est à rendre avant TODO. Vous trouverez sur Cahier de Prépa une archive contenant plusieurs fichiers pour le TP. Comme d'habitude : **commentaires**, **assertions**, **tests** !
Les exercices et questions **optionnels**, généralement un peu plus durs, doivent être sautés au premier passage, afin d'y revenir une fois que vous avez fini la partie principale du TP.

1 Fichiers (Pas besoin d'utiliser malloc dans cette partie)

Un fichier est une suite d'octets stockée sur le disque dur de l'ordinateur. Dans le nom d'un fichier, l'extension (`.txt`, `.mp3`, etc...) permet d'indiquer à l'utilisateur le type de données stockée, et permet donc aussi d'indiquer à l'ordinateur quel logiciel utiliser par défaut pour lire et interpréter les octets du fichier. Les fichiers texte sont des fichiers utilisant l'encodage ASCII (1 octet par symbole), l'encodage UTF-8 (entre 1 et 4 octets par symbole), ou tout autre encodage standard pour représenter du texte.
La plupart des éditeurs de texte standards sont capable de détecter automatiquement le type d'encodage d'un fichier et de s'y adapter.
Dans le terminal, la commande `od -t x1 -A x mon_fichier` permet d'afficher le contenu de `mon_fichier` **octet par octet**, en hexadécimal. Cette commande devrait être par défaut sur toutes vos installations : machine virtuelle, Mac, Linux, WSL, etc...

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. Dans un terminal, tapez la commande `od -t x1 -A x quarante.txt`. Cette commande affiche :

```
00000000  34 30 0a
00000003
```

Le fichier contient donc 3 octets : `0x34`, `0x30` et `0x0a`. A quoi correspond chacun ?

Q2. En utilisant la même commande sur le fichier `quarante.pas_txt`, donner le nombre d'octets du fichier, lister ces octets dans l'ordre d'apparition, et expliquer ce que l'on observe.

En C, il existe des fonctions permettant de lire/écrire des octets purs dans un fichier, mais aussi des fonctions permettant spécifiquement de lire/écrire du texte. Ce sont ces dernières qui sont au programme de MP2I, dans la suite on s'intéressera donc principalement à des fichiers textuels. La manipulation de fichiers se fait via un nouveau type, le type `FILE`, que l'on manipule avec les 4 fonctions suivantes :

- `FILE* fopen(char* filename, char* mode)` permet d'ouvrir le fichier de nom `filename`, dans le mode spécifié. Les trois modes à connaître sont :
  - `"w"` (comme write) : ouvre le fichier en mode écriture, le crée s'il n'existe pas, et **le vide intégralement s'il existe**.
  - `"r"` (comme read) : ouvre le fichier en mode lecture. Si le fichier n'existe pas, la fonction échoue.
  - `"a"` (comme append) : ouvre le fichier en mode écriture, le crée s'il n'existe pas, et positionne la tête d'écriture à la fin du fichier. Ce mode permet donc d'écrire en rajoutant les données à la fin de celles déjà présentes dans le fichier.

  Si `fopen` échoue, alors elle renvoie le pointeur NULL. Il faudra toujours vérifier que le pointeur renvoyé n'est pas NULL avant de commencer à utiliser le fichier. Notamment, ceci vous permettra de détecter facilement les moments où vous ouvrez en lecture un fichier qui n'existe pas !
- `int fclose(FILE* f)` permet de fermer le fichier `f` une fois qu'on a fini de l'utiliser. Cette fonction renvoie un code d'erreur (0 si tout s'est bien passé), que l'on ignorera en TP.
- `fprintf` et `fscanf` fonctionnent exactement comme `printf` et `scanf`, mais prennent en premier argument le fichier dans lequel écrire/lire. `fscanf` renvoie le nombre d'objets lus.

Exercice 2

Le programme suivant ouvre le fichier `hello.txt` et y écrit "Hello World 2024 !". Si le fichier n'existe pas, il est créé :

```c
#include <stdio.h>
#include <assert.h>

int main(){
  FILE* f = fopen("hello.txt", "w"); // ouverture en mode écriture
  assert(f != NULL);
  fprintf(f, "Hello World %d!\n", 2024);
  fclose(f);
}
```

Recopiez ce programme, compilez et exécutez-le, et vérifiez qu'il fonctionne, c'est à dire qu'il crée bien le fichier `hello.txt` dans le dossier où vous exécutez le programme.

Exercice 3

La fonction suivante prend en entrée un nom de fichier, tente de lire deux entiers dedans, et renvoie la somme. Elle échoue si le fichier n'existe pas où s'il ne contient pas deux entiers.

```c
/* Renvoie la somme des deux premiers entiers lus dans le fichier
   `nom_fichier`.
   Précondition: `nom_fichier` doit commencer par au moins
   deux entiers
*/
int somme2(char* nom_fichier){
  FILE* f = fopen(nom_fichier, "r"); // ouverture en mode lecture
  assert(f != NULL);
  int x, y;
  int nb_lus = fscanf(f, "%d %d", &x, &y); // lire deux entiers dans f
  assert(nb_lus == 2);
  fclose(f);
  return x + y;
}
```

Recopiez la fonction précédente dans un fichier C, et testez-la et créant quelques fichiers texte et en y écrivant des entiers à la main via le bloc note ou autre éditeur de texte. La commande `touch` du terminal permet de créer un fichier, par exemple en tapant `touch test.txt`. Laissez dans le dossier du TP les fichiers créés lorsque vous le rendez.

La fonction fscanf renvoie une valeur particulière lorsqu'elle atteint la fin du fichier : `EOF`. Ceci permet d'écrire des programmes qui lisent **jusqu'à la fin** d'un fichier.

Exercice 4

La fonction suivante renvoie la somme de tous les entiers lus dans un fichier :

```c
/* Renvoie la somme des entiers lus dans le fichier `nom_fichier`.
   Précondition: `nom_fichier` ne doit contenir que des entiers. */
int somme(char* nom_fichier){
  FILE* f = fopen(nom_fichier, "r"); // ouverture en mode lecture
  assert(f != NULL);
  int x = 0; // pour lire dans le fichier
  int tot = 0; // pour stocker la somme
  int nb_lus = 0;
  while (nb_lus != EOF){
    tot += x;
    nb_lus = fscanf(f, "%d", &x); // nb_lus vaut 1 ou EOF
  }
  fclose(f);
  return tot;
}
```

Recopiez la fonction suivante dans un nouveau fichier, et testez-la.

Exercice 5

Créez un nouveau fichier C pour cet exercice, et écrivez-y les fonctions suivantes, en n'oubliant pas de bien les tester au fur et à mesure.

Q1. Écrire une fonction `int premier_zero(char* filename)` qui va lire dans un fichier, supposé contenir uniquement des entiers, et compte le nombre d'entiers strictement positifs lus avant de lire un zéro. Cette fonction renverra -1 si elle lit tout le fichier sans jamais rencontrer de 0.

Q2. (**Optionnelle**) Écrire une fonction qui prend en entrée deux noms de fichiers `in_fn` et `out_fn`, qui recopie le premier dans le deuxième en renversant chaque mot. Par exemple, ci-dessous : à gauche un fichier d'entrée, et à droite le fichier de sortie correspondant.

```
MP2 Tous des Dieux                2PM suoT sed xueiD
ruojnoB a setuot te suot          Bonjour a toutes et tous
```

On pourra supposer qu'aucun mot ne fait plus que 50 caractères de long. *Indication : commencez par écrire une fonction* `void renverser(char* s)` *qui renverse une chaîne de caractères.*

Exercice 6

Vous avez pu remarquer que certaines commandes du terminal prennent des arguments, alors que jusqu'ici, en C, lorsque l'on a compilé un exécutable `prog`, on le lance en tapant simplement `./prog`. En réalité, on peut taper `./prog argument1 argument2 ...`, et l'on va voir dans cet exercice comment accéder aux arguments dans le programme.

La fonction main prend en fait deux paramètres :

```c
int main(int argc, char** argv){
```

`argv` est le **vecteur d'argument**, c'est un tableau de chaînes de caractères contenant tous les arguments fournis lors de l'exécution du programme. `argc` est le **compte d'argument**, c'est simplement la longueur du vecteur d'arguments.

Par exemple, le programme suivant, très simple, affiche chacun de ses arguments :

```c
#include <stdio.h>
int main(int argc, char** argv){
  for (int i = 0; i < argc; i++){
    printf("%d-ème argument: %s\n", i, argv[i]);
  }
}
```

Q1. Recopiez, compilez et exécutez ce programme en lui donnant des arguments, par exemple en tapant `./a.out bla 31 mp2i`. Recompilez le programme en lui donnant un autre nom avec `-o`, et réexécutez. Que peut-on en déduire sur `argv[0]` ?

Q2. Écrire un **programme** `./copieur` qui prend en argument deux noms de fichiers, et recopie le contenu du premier dans le deuxième. Par exemple, si l'on tape dans le terminal `./copieur a.txt b.txt`, le contenu de `a.txt` est recopié dans `b.txt`, écrasant ce qui s'y trouvait avant, et créant le fichier s'il n'existe pas.

On pourra utiliser le format `"%c"` qui permet de lire /écrire un `char` à la fois.

Vous venez de recoder la commande `cp` du terminal !

2 Allocation dynamique

On rappelle que la fonction `malloc` permet d'allouer de la mémoire de manière dynamique au cours de l'exécution d'un programme. Pour l'utiliser, on précise en argument le nombre **d'octets** à réserver. On utilise généralement l'opérateur `sizeof`, pour ne pas à avoir à écrire explicitement la taille des types, par exemple :

```
1 int* t = malloc(10 * sizeof(int)); // réserve 10 cases mémoires de type int
```

On rappelle que l'intérêt principal de `malloc` est qu'elle permet de réserver de la mémoire dans le tas, de manière permanente. En particulier, un pointeur obtenu par `malloc` peut être renvoyé sans problème par une fonction. Cependant, **il faut libérer la mémoire allouée vous même** en utilisant la fonction `free`.

Exercice 7

Créez un unique fichier C pour cet exercice.

Q1. Implémentez une fonction `bool egaux(int* t1, int* t2, int n)` qui renvoie un booléen indiquant si les tableaux t_1 et t_2, supposés de taille n, contiennent les mêmes valeurs.

Vous pouvez utiliser la fonction `egaux` pour vos tests. Par exemple, si une fonction `int* premiers_nats(int n)` est sensée renvoyer un tableau contenant les n premiers entiers naturels, on pourra la tester comme suit.

```
1 int* t = premiers_nats(5);
2 int test[5] = {0,1,2,3,4};
3 assert(egaux(t, test, 5));
4 free(t);
```

Q1. Dans le main, créez un tableau de 5 entiers stockés dans le tas. Écrivez-y 0, 1, 2, 3, 4, puis libérez la mémoire.

Q2. Implémentez une fonction `int* zeros(int n)` qui renvoie un tableau de n cases, toutes nulles. Testez-la dans votre main (n'oubliez pas de libérer la mémoire).

Q3. Implémentez une fonction `bool* zeros_uns(int n, int m)` qui renvoie un tableau de booléens, dont les n premières valeurs sont fausses, et dont les m suivantes sont vraies. Testez-la dans votre main.

Débug **Valgrind** est un outil de débug très puissant qui peut détecter les fuites mémoires mais aussi tout un tas d'erreurs, comme des accès à des zones invalide, l'utilisation de valeurs non-initialisées, l'oubli d'un return dans une fonction, etc... Lorsque l'on compile un programme C, on peut utiliser l'option `-g` pour rajouter à l'exécutable des informations utiles à valgrind. On peut ensuite exécuter le programme **dans** valgrind. Par exemple :

```
fredfrigo@ubuntu:~/TP4$ gcc mon_programme.c -o mon_prog -g
fredfrigo@ubuntu:~/TP4$ valgrind ./mon_prog
```

La deuxième commande va lancer valgrind, et simuler l'exécution de `./mon_prog`, en signalant les erreurs et les fuites mémoires. Les rapports d'erreur faits par valgrind sont longs et difficiles à lire au départ, mais avec un peu d'entraînement, ils permettent de débugger les programmes de manière très efficace. Une règle d'or : comme pour GCC, il faut toujours lire et régler les

erreurs de haut en bas.
Valgrind est installé par défaut sur les machines virtuelles. Pour ce premier TP, travaillez sur la machine virtuelle, et installez valgrind sur votre ordinateur personnel pour la suite de l'année. Si vous êtes sous Linux ou sur WSL, vous pouvez installer valgrind en tapant `sudo apt install valgrind` dans le terminal. Si vous êtes sur MacOS, selon le modèle, votre ordinateur peut ne pas être compatible avec valgrind, il faudra venir me voir.

Exercice 8

L'archive du TP contient un dossier `erreurs/` contenant trois fichiers C qui compilent, mais qui causent des erreurs à l'exécution ou des fuites mémoires. En utilisant valgrind, déterminez les sources des erreurs/fuites et corrigez-les.
Profitez-en pour vérifier qu'il n'y a pas d'erreurs ou de fuites mémoires dans les exercices précédents !

Dans la suite du TP, et pour tout le reste de l'année, **utilisez valgrind** pour vous assurer que vous n'avez pas de fuites mémoires ou d'erreurs !

Exercice 9

Implémentez une fonction `int* lire_entiers(char* filename, int* n)` qui lit dans le fichier `filename` jusqu'à $*n$ entiers et les renvoie sous la forme d'un tableau. De plus, la fonction va stocker dans l'adresse pointée par n le nombre d'entiers effectivement lus, si jamais la lecture atteint la fin du fichier avant d'avoir lu $*n$ entiers.

Exercice 10

On rappelle que pour représenter des matrices / grilles de nombres, on utilise des **tableaux de tableaux**, c'est à dire des **pointeurs de pointeurs**.

Q1. Écrire une fonction `int** zeros(int n, int m)` qui renvoie une matrice nulle de dimensions $n \times m$, ainsi qu'une fonction `int** rand_mat(int n, int m, int a, int b)` qui renvoie une matrice de dimensions $n \times m$ dont chaque valeur est dans $[\![a, b]\!]$.

Q2. Écrire deux fonctions `free_mat(int** g, int n)` et `print_mat(int**g, int n, int m)` qui servent respectivement à libérer la mémoire allouée pour une matrice et à afficher le contenu d'une matrice.

Q3. Écrire une fonction `int** somme_mat(int** g1, int** g2, int n, int m)` qui renvoie la matrice $g_1 + g_2$. On supposera que g_1 et g_2 sont des matrices $n \times m$.

Q4. Écrire une fonction faisant le **produit** de deux matrices (pas forcément carrées).

Q5. Écrire une fonction `void min_moy(int**g, int n, int m)` qui cherche dans une grille les indices i_0, j_0 tels que la moyenne des valeurs de $g[i_0][j_0]$ et de ses voisines est minimale (on comptera comme voisines les cases se touchant par un côté ou par un coin). Cette fonction **affichera** les indices (i_0, j_0) correspondants, ainsi que la moyenne obtenue. Il pourra être pratique d'écrire une fonction auxiliaire calculant la moyenne pour une position (i, j) donnée.

Q6. Modifier la fonction précédente pour qu'elle permette de **récupérer** les valeurs de i et j trouvées, de façon à pouvoir utiliser ces valeurs dans le main par exemple.

Exercice 11

Lorsque l'on utilise `scanf` avec le format `%s`, on lit mot par mot, en s'arrêtant aux espaces. Il existe dans `stdio.h` une fonction

```
1 int getline(char** buffer, unsigned long int* n, FILE* f);
```

qui permet de lire une ligne entière d'un coup. Plus précisément :

- La fonction lit dans le fichier f jusqu'à y rencontrer un retour à la ligne, stocke la chaîne lue dans la chaîne pointée par `buffer`.
- Si en entrée, `buffer` pointe vers un pointeur nul, i.e. si `*buffer == NULL`, la fonction **alloue** de la place pour y stocker la chaîne de caractère, et va stocker dans `*n` la taille mémoire de la zone allouée.
- Si en entrée, `buffer` pointe vers un pointeur non nul, alors `n` doit pointer vers une case contenant la taille de la zone mémoire allouée pour `*buffer`. Dans ce cas, la fonction va directement stocker la chaîne de caractère à cet endroit, et s'il n'y a pas assez de place, elle **réalloue** de la mémoire et modifie en conséquence la valeur pointée par `n`.
- Dans tous les cas, la fonction **renvoie** un entier indiquant le nombre de caractères lus, y compris le retour à la ligne.

Par exemple, en supposant que l'on a ouvert un fichier `FILE* f` en mode écriture :

```
1 char* ligne = NULL;
2 int n = 0;
3 int len = getline(&ligne, &n, f);
```

Si l'on exécute ce code et que le fichier ouvert contient "`Une phrase quelconque`" sur la première ligne, alors `len` vaudra 22 (21 caractères + le retour à la ligne), et la valeur de `ligne` aura été modifiée, ça sera désormais l'adresse mémoire d'une zone du tas d'au moins 23 octets contenant `Une phrase quelconque\n` (plus le caractère nul). De plus, `n` aura aussi été modifié et vaudra au moins 23 (en pratique, `getline` réserve souvent un peu plus d'espace que nécessaire).

Q1. Expliquer pourquoi le premier argument de getline est de type `char**` et pas simplement de type `char*`.

Q2. En utilisant `getline`, écrire un programme C qui lit dans un fichier et affiche toutes les lignes, en préfixant à chaque fois par un numéro de ligne. On voudra l'utiliser comme suit :

```
fredfrigo@ubuntu:~/TP4$ ./lignes bla.txt
1. Ceci est un fichier
2.
3. Il y a trois lignes dedans, la deuxième est vide.
```

Q3. (**Optionnelle**) Écrire un programme C qui permet de trier les lignes d'un fichier par ordre alphabétique, et de stocker le résultat dans un autre fichier.

Nous allons maintenant implémenter une version simplifiée de `getline`, dans laquelle on suppose que `buffer` pointe forcément vers un pointeur nul, et donc que c'est à nous d'allouer la mémoire. Notons que l'on ne peut pas allouer directement la bonne taille, car on ne connaît pas au préalable le nombre de caractères que l'on va lire. Il va donc falloir **réallouer** de la mémoire au fur et à mesure. L'algorithme que nous allons employer est le suivant :

1. Allouer une zone mémoire d'une taille arbitraire (par exemple 30 octets) ;
2. Lire caractère par caractère dans le fichier ;
3. A chaque fois qu'on dépasse la taille autorisée, réallouer une zone mémoire plus grande.

La fonction `T* realloc(T* p, unsigned int size)` (où T est un type quelconque) est de la même famille que malloc et permet de réallouer de la mémoire, en libérant la zone mémoire pointée par p et en allouant 'size' octets autre part dans le tas. Voici un exemple d'utilisation :

```
1  int* a = malloc(8*sizeof(int));
2  a[0] = 1; a[1] = 2; ...; a[7] = 8;
3
4  // ah en fait je voudrais 50 int et pas 8 !
5  a = realloc(a, 50*sizeof(int));
6  // les 8 valeurs de a ont été recopiées automatiquement
7  assert(a[0] == 1); ...; assert(a[7] == 8);
8
9  // finalement je ne veux que 3 int !
10 a = realloc(a, 3*sizeof(int));
11 // les 3 premières valeurs de a ont été recopiées automatiquement
12 assert(a[0] == 1); ...; assert(a[2] == 3);
```

Dans notre cas, il faut déterminer quelle taille donner à la zone réallouée à chaque fois. Nous allons voir en cours qu'une méthode efficace est de **doubler** la taille allouée à chaque fois que l'on remplit la zone réservée.

Q4. Implémenter une fonction `int ma_getline(char** buffer, unsigned long int* n, FILE* f)` correspondant à la spécification donnée, en utilisant l'algorithme suggéré.

Q5. **(Optionnel)** Modifier la fonction pour qu'elle prenne en compte les entrées où `buffer` ne pointe pas vers un pointeur nul initialement.

Exercice 12: Optionnel mais fortement conseillé

Le but de cet exercice est d'écrire un programme qui écrit des messages dans le terminal en utilisant des grandes lettres.

Pour cela, on dispose de $26 + 26 + 10 = 62$ fichiers, un par lettre minuscule, un par majuscule et un par chiffre. Chaque fichier contient 8 entiers $x_0, \dots x_7$, qui encodent une image de 8×8 pixels de la manière suivante : chaque ligne de l'image a 8 pixels, et la ligne i a pour pixels allumés les pixels (i, j) tels que x_i a un 1 dans son écriture en binaire au j-ème bit à partir de la gauche. Par exemple, le fichier `2.txt` contient `60 66 4 24 32 126 0 0` : en traduisant en binaire (sur 8 bits) puis en mettant des `@` au niveau des 1 et en laissant les 0 vides on obtient :

```
60  ↦ 00111100 ↦   @@@@
66  ↦ 01000010 ↦  @    @
4   ↦ 00000100 ↦      @
24  ↦ 00011000 ↦    @@
32  ↦ 00100000 ↦   @
126 ↦ 01111110 ↦  @@@@@@
0   ↦ 00000000 ↦
0   ↦ 00000000 ↦
```

Certains systèmes de fichiers ne faisant pas la différence entre les lettres minuscules et majuscules, les lettres minuscules sont stockées dans les fichiers dont le nom est la lettre seule, pour les lettres majuscules la lettre est doublée. Par exemple, la lettre `A` sera stockée dans `AA.txt` mais la lettre `a` sera stockée dans `a.txt`. Tous ces fichiers sont présents dans le dossier `lettres/` de l'archive du TP.

Q1. A la main, représentez / dessinez la lettre '`y`' telle que décrite dans le fichier correspondant.

Le but de l'exercice est d'afficher une phrase en remplaçant chaque lettre par sa représentation en une grille de 8×8 caractères.

On souhaite pouvoir recréer le comportement suivant :

```
fredfrigo@ubuntu:~/TP4$ ./phrase @ 9
Rentrez la phrase à afficher: MP9 Tous des Oeufs

 @      @ @@@@      @@@@           @@@@@@@
 @@    @@ @    @    @    @             @                        @@@
 @ @ @ @ @@@@     @    @              @      @@@@     @    @    @
 @  @  @ @        @@@@@              @     @    @    @    @     @@
 @      @ @            @             @     @    @    @   @@       @
 @      @ @        @@@@              @      @@@@      @@ @    @@@


      @                              @@@
      @    @@@@      @@@            @   @    @@@@              @@       @@@
  @@@@@   @    @    @               @   @   @    @    @    @     @       @
 @    @   @@@@@@     @@              @   @   @@@@@@   @    @    @@@       @@
 @    @@   @            @            @   @   @        @   @@     @          @
  @@@ @    @@@@     @@@               @@@     @@@@      @@ @     @       @@@
```

Le programme va donc prendre en entrée un caractère p (comme pixel) à utiliser pour l'affichage et un nombre de lettres par ligne K, puis demander une phrase à l'utilisateur. En notant n le nombre de caractères de cette phrase, le programme va segmenter la phrase en $\lceil \frac{n}{K} \rceil$ blocs, chacun représenté par une grille de K lignes par 8 colonnes.

Avant de se lancer dans l'écriture de ce programme, nous allons réfléchir à l'organisation à adopter. Pour cela, on va chercher à lister les structures utilisées et les fonctions utiles, sans les implémenter immédiatement. Par exemple, on peut utiliser :

- Un tableau 2D de dimensions $8 \times 8n$ où n est la taille de la phrase, pour stocker la phrase mise sous format "image".
- Une fonction `char* img_filename(char c)` qui génère le nom du fichier à ouvrir pour trouver les données correspondant au caractère c. Si le caractère en entrée n'est pas un espace, une lettre ou un chiffre, la fonction renverra `NULL`.
- Une fonction `void print_bloc(char** G, int n, int m, int K)` qui affiche la grille G par blocs de taille $m \times K$. (En pratique m vaudra 8).

```c
/* Renvoie le nom du fichier contenant les données nécessaires
   pour dessiner le caractère c */
char* img_filename(char c);

/*
Affiche la matrice G de n lignes et m colonnes par blocs de K colonnes.
Par exemple, si G est la matrice suivante:
AAABBBCCC
AAABBBCCC
alors print_bloc(G, 2, 9, 5) affichera:
AAABB
AAABB
BCCC
BCCC
*/
void print_bloc(char** G, int n, int m, int K);
```

Notons que l'on n'a pas besoin de les implémenter pour le moment, on est en train de construire l'organisation du programme !

De manière générale, pour imaginer des fonctions et des structures utiles, on peut approcher le problème par le haut, c'est à dire considérer les grandes étapes à réaliser, ou bien l'approcher par le bas, c'est à dire considérer les petites briques de base dont on pourra avoir besoin. L'archive du TP contient un fichier `message.c` (**NE L'OUVREZ PAS TOUT DE SUITE !**) contenant une proposition d'organisation pour le programme, ainsi qu'une fonction main partiellement remplie. Le but de cet exercice est de réfléchir à une organisation du programme par vous-même, et d'utiliser ce fichier comme base si vous n'avez pas d'idées.

Q2. Décrivez l'organisation que vous envisagez pour votre programme, et listez les déclarations de fonctions que vous comptez utiliser. Soyez précis dans vos commentaires, et n'hésitez pas à y mettre des exemples illustrant le fonctionnement.

Q3. Implémentez le programme, soit en utilisant l'organisation que vous avez décrite à la question précédente, soit en utilisant l'organisation proposée dans le fichier `message.c`.