TP4: Allocation dynamique, fichiers

MP2I Lycée Pierre de Fermat

Consignes

Vous pouvez télécharger sur Cahier de Prépa une archive contenant des fichiers pour ce TP.

Vous devez rendre sur Cahier de Prépa une archive avec votre code et vos réponses avant mardi 14/11 à 22h00.

Supprimez bien tous les exécutables avant de compresser l'archive, car certaines boîtes mail refusent d'envoyer des archives contenant des exécutables (c'est le cas de Gmail). Certaines questions sont **optionnelles**, généralement elles viennent clore un exercice, et sont un peu plus dures. Il est conseillé de les sauter afin d'y revenir une fois que vous avez fini la partie principale du TP. Il est fortement conseillé de les faire si vous êtes à l'aise, et de vous y essayer peu importe votre niveau.

Vous devez commenter ***toutes*** vos fonctions, mettre des assertions lorsque c'est nécessaire pour garantir certaines préconditions, et surtout faire des tests. Une bonne pratique de programmation lorsqu'on code une fonction est d'écrire des tests **avant** d'implémenter la fonction. Cela permet de tester la fonction immédiatement après l'avoir codé, et de détecter les bugs assez rapidement, mais en plus ça nous force à réfléchir à des cas limites, et à des détails qui peuvent apparaître lors de l'implémentation.

Vous êtes encouragé/e/s à travailler à plusieurs, à vous expliquer les exercices entre vous, mais vous devez rendre uniquement du code que vous avez écrit vous-même !

Dans certains exercices de ce TP, vous aurez besoin de manipuler les chaînes de caractères. Nous avons déjà parlé de la librairie `string.h`, et vous avez même recodé certaines des fonctions de cette librairie dans le TP précédent. Dans ce TP, on pourra utiliser la librairie `<string.h>` à volonté. On rappelle les 4 fonctions principales de cette librairie :

- `strlen` (comme string length) calcule la longueur d'une chaîne de caractères
- `strcpy` (comme string copy) copie le contenu d'une chaîne de caractères vers un autre pointeur
- `strcat` (comme string concatenate) concatène une chaîne de caractères à la fin d'une autre.
- `strcmp` (comme string compare) compare deux chaînes de caractères, et renvoie 0 si elles sont égales, et ± 1 si elles ne sont pas égales, selon laquelle est la plus grande dans l'ordre lexicographique.

Vous pouvez regarder la spécification exacte de ces fonctions en regardant dans le ***Manuel***, avec la commande ***man*** du terminal. Par exemple pour savoir comment fonctionne strcat :

man strcat

Les pages du manuel sont généralement assez chargées, il faut donc s'entraîner à pouvoir en extraire les informations importantes : signature de type des fonctions, signification des paramètres et de la valeur de retour, préconditions, etc...

Fichiers

Toute cette partie peut être traitée sans utiliser `malloc`.

Un fichier est simplement 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.

Sur Linux, la commande `hexdump -C` permet d'afficher le contenu d'un fichier octet par octet, en hexadécimal.

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 sur 4 octets en entier signé. L'archive contient également le code `ecrire_40.c` du programme utilisé pour générer ce fichier, vous pouvez l'ouvrir et constater que ce programme recopie la zone mémoire d'une variable int contenant la valeur 40. Il utilise des fonctions que l'on n'a pas encore vu : fiez vous au commentaire.

Question 1. Dans un terminal, tapez la commande `hexdump -C quarante.txt`. Cette commande affiche :

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

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

Question 2. 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 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 que 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 des données 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.

— `int fclose(FILE* f)` permet de fermer le fichier `f`. Il faut imaginer que fopen et fclose sont des parenthèses : si l'on ouvre un fichier, on **doit** le fermer. Cette fonction renvoie un code d'erreur (0 si tout s'est bien passé), que l'on ignorera en TP car les erreurs de fermeture sont extrêmement rare.

— `fprintf` et `fscanf` fonctionnent exactement comme `printf` et `scanf` : on spécifie en premier argument le fichier dans lequel écrire/lire. `fscanf` renvoie le nombre d'objets lus, et renvoie une valeur spéciale nommée `EOF` (comme End Of File) lorsqu'elle atteint la fin du fichier.

Exemple 1. Le code suivant va demander deux noms de fichiers à l'utilisateur, aller lire des entiers dans le premier et écrire la somme dans le deuxième :

```c
#include <stdio.h>

int main(){
  char filename_in[20], filename_out[20];
  printf("Rentrez un nom de fichier à lire: ");
  scanf("%s", filename_in);
  printf("Rentrez un nom de fichier pour écrire la somme: ");
  scanf("%s", filename_out);

  // ouverture du fichier d'entrée: mode lecture
  FILE* f_in = fopen(filename_in, "r");
  assert(f_in != NULL);

  int sum = 0, n = 0;
  while (fscanf(f_in, "%d", &n) != EOF){ // on s'arrête à la fin du fichier
    sum += n;
  }
  fclose(f_in);

  // écrire la somme dans le fichier de sortie: mode écriture
  FILE* f_out = fopen(filename_out, "w");
  assert(f_out != NULL);
  fprintf(f_out, "%d\n", sum);
  fclose(f_out);

  return 0;
}
```

Le code du programme précédent est présent dans le fichier `filesum.c` dans l'archive du TP, vous pouvez le compiler et l'exécuter, puis le modifier afin d'expérimenter et de bien comprendre comment il fonctionne.

Dans tout le reste du TP, on manipulera uniquement des fichiers texte.

Exercice 2.

Dans cet exercice, vous allez écrire des fonctions manipulant des fichiers. Pour vos tests, vous devez donc créer des fichiers permettant de tester vos fonctions, en pensant bien à couvrir plusieurs cas, à tester les cas limites, etc...

Question 1. 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.

Question 2. Écrire une fonction `int nb_occurences(char* filename, char* motif)` qui prend en entrée un nom de fichier et une chaîne de caractères supposée sans espaces, et qui compte le nombre d'occurrences de la chaîne dans le fichier. On pourra supposer que le fichier ne contient que des mots de moins de 30 caractères, et donc que l'on peut stocker chaque mot lu dans un tableau `char mot[31];`.

Question 3. 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, si le fichier d'entrée contient :

```
MP2 Tous des Dieux
ruojnoB a setuot te suot
```

alors le fichier de sortie contiendra :

```
2PM suoT sed xueiD
Bonjour a toutes et tous
```

On pourra supposer qu'aucun mot ne fait plus que 50 caractères de long.

Question 4. (**Optionnel**) Renseignez-vous sur une des fonctions de tri suivantes : tri à bulle, tri par sélection, tri par insertion. Implémentez la, et écrivez une fonction qui prend en entrée un nom de fichier, supposé contenir uniquement des entiers, qui trie le contenu du fichier. On pourra supposer que le fichier n'aura jamais plus que 100000 entiers écrits.

Exercice 3.

Dans le terminal, la commande **cp** permet de copier un fichier. Elle s'utilise comme suit :

```
cp fichier_source fichier_arrivee
```

Ceci crée un fichier appelé `fichier_arrivee` identique au fichier source.

Nous allons écrire un programme C qui imite le comportement de **cp**. On remarque que lorsque l'on utilise **cp**, on écrit directement après la commande des arguments : le nom du fichier source, et celui du fichier sortie. De la même manière, la commande **gcc** prend des arguments (notamment le nom du fichier source à compiler).

On dit que ces commandes **prennent des argument**. Voyons comment écrire des programmes C qui peuvent accéder et manipuler les arguments qu'on leur donne.

Jusqu'ici, nous avons écrit la fonction `main` sans paramètres. En réalité, `main` se déclare ainsi :

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

Le premier argument, `argc`, donne le nombre d'arguments utilisés lorsque le programme a été exécuté. Par exemple, si l'on exécute la commande

```
./a.out bla 5 poisson
```

alors `argc` vaudra 4 et argv aura le contenu suivant :

argv[0] : "a.out"
argv[1] : "bla"
argv[2] : "5"
argv[3] : "poisson"

Ainsi, `argv` (comme **arg**ument **v**ector) est un tableau de longueur `argc` (comme **arg**ument **c**ount) qui contient les différents arguments utilisés. La case d'indice 0 contient toujours le nom de l'exécutable, et donc `argc` est toujours strictement positif.

Question 1. Écrire un programme C qui affiche son nombre d'arguments, puis affiche les arguments sur une ligne chacun.

Question 2. Renseignez-vous sur la fonction `atoi` de la librairie `stdio.h`. Écrire un programme C qui prend en argument un entier et affiche son carré. Si le programme est lancé sans argument, il doit afficher un message d'erreur et s'arrêter. On peut utiliser l'instruction `exit(1);` pour arrêter immédiatement un programme.

Question 3. Écrire un programme C qui imite le comportement de la commande **cp**.

Exercice 4.

Nous avons vu que `scanf("%s")` permet de lire des mots dans le terminal. Cependant, on ne peut pas directement l'utiliser pour lire des lignes entières car la fonction s'arrête de lire dès qu'elle voit un espace. Le but de cet exercice est donc d'implémenter la fonction suivante :

```c
/* Lit dans le fichier f une ligne entière, et la stocke dans resultat.
   Seul les n_max premiers caractères sont stockés, et si la ligne dépasse
   de cette limite, les caractères supplémentaires sont perdus.
   resultat doit pointer vers une zone d'au moins (n_max+1) octets pour
   y stocker la ligne lue ainsi que le caractère nul de termination.
*/
void lire_ligne(FILE* f, char* resultat, int n_max);
```

Le schéma pour cette fonction sera de lire dans `f` caractère par caractère, en utilisant le format `"%c"`, jusqu'à rencontrer soit un saut de ligne soit la fin du fichier, ou bien jusqu'à avoir lu le nombre maximal de caractères autorisé.

Question 1. Créez un fichier texte, écrivez-y quelques lignes d'au plus 30 caractères chacune, en mêlant des nombres, des mots, des espaces, etc...

Question 2. Avant d'implémenter la fonction `lire_ligne`, écrivez dans le main un jeu de test utilisant le fichier que vous venez de créer, qui permettra de vérifier le bon fonctionnement de la fonction une fois que vous l'aurez codé. On attend l'utilisation des fonctions `assert` et `strcmp`.

Question 3. Implémentez la fonction `lire_ligne`, et vérifiez qu'elle passe bien vos tests.

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. **N'oubliez pas de libérer la mémoire allouée** en utilisant la fonction `free`.

Vous êtes encouragés à utiliser **valgrind**, 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... Prenez le réflexe de lancer vos exécutables avec valgrind plutôt que de les lancer seuls. Pour cela, il faut compiler avec l'option -g, qui permettra à valgrind d'avoir le numéro des lignes où les erreurs se produisent :

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

Lorsque vous utilisez des options de valgrind pour avoir des rapports plus détaillés, il faut mettre les options **avant** le nom de l'exécutable :

```
grousseau@ubuntu:~/TP4$ valgrind --leak-check=full ./mon_prog
```

Exercice 5.

Tout le code de cet exercice ira dans un seul et même fichier. Pour chaque question, vous devez coder une fonction dont la spécification vous est donnée. Vous devez commenter et tester ces fonctions. Pensez bien à la pertinence de vos tests, pour cela, imaginez que vous écrivez des tests qui devront être passés par le code d'un ou d'une camarade : vous devez chercher les cas les plus embêtants qui respectent tout de même la spécification de la fonction.

Question 1. 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` dans la suite 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);
```

Question 2. Implémentez une fonction `int* zeros(int n)` qui renvoie un tableau de n cases, toutes nulles.

Question 3. 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.

Question 4. Implémentez une fonction `float* lire_flottants (FILE* f, int* n)` qui lit dans le fichier f jusqu'à $(*n)$ flottants. De plus, la fonction va stocker dans l'adresse pointée par n le nombre de flottants effectivement lus, si jamais il y en a strictement moins que prévu.

Exercice 6.

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

Question 1. 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 $[\![a, b]\!]$.

Question 2. 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.

Question 3. Écrire une fonction `int** somme_mat(int** g1, int** g2)` qui renvoie la matrice somme $g_1 + g_2$.

Question 4. Écrire une fonction `void min_moy(int**g, int n, int m)` qui cherche dans une grille les indices i, j tels que la moyenne des valeurs de $g[i][j]$ et de ses voisines (on comptera comme voisines les cases se touchant par un côté ou par un coin). Cette fonction affichera les indices (i, j) correspondant, ainsi que la moyenne obtenue. Il pourra être pratique d'écrire une fonction auxiliaire calculant la moyenne pour une position (i, j) donnée.

Exercice 7.

On reprend la fonction `lire_ligne` de l'exercice 4. Il existe dans la librairie standard, dans `stdio.h`, une fonction `int getline(char** buffer, unsigned long int* n, FILE* f)` qui permet de lire une ligne. Son comportement précis est le suivant :

- 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 la taille de la zone mémoire autorisée 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 :

```
1 char* ligne = NULL;
2 int n = 0;
3 int len = getline(&ligne, &n, stdin); // stdin est un fichier toujours ouvert
4                     // correspondant à l'entrée standard du
5                     // programme, i.e. on lit dans le terminal
```

Si l'on exécute ce code et que l'on écrit dans le terminal `Une phrase quelconque`, alors la valeur de `ligne` aura été modifiée, ça sera désormais l'adresse mémoire d'une zone d'au moins 23 octets contenant `Une phrase quelconque\n` (plus le caractère nul). `len` vaudra 22, et `n` vaudra au moins 23 (en pratique, `getline` réserve souvent un peu plus d'espace que nécessaire.

Question 1. Expliquer pourquoi le premier argument de getline est de type `char**` et pas simplement de type `char*`.

Question 2. 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 :

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

Nous allons maintenant implémenter une version un peu simplifiée de `getline`, dans laquelle on suppose que `buffer` pointe forcément vers un pointeur nul. 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

On pourra utiliser la fonction `realloc`, qui est de la même famille que malloc et permet de réallouer de la mémoire. Voici un exemple d'utilisation :

```c
int* a = malloc(8*sizeof(int));
a[0] = 1; a[1] = 2; ...; a[7] = 8;

// ah en fait je voudrais 50 int et pas 8 !
a = realloc(a, 50*sizeof(int));
// les 7 valeurs de a ont été recopiées automatiquement
assert(a[0] == 1); ...; assert(a[7] == 8);

// finalement je ne veux que 3 int !
a = realloc(a, 3*sizeof(int));
// les 3 premières valeurs de a ont été recopiées automatiquement
assert(a[0] == 1); ...; assert(a[2] == 3);
```

Dans notre cas, il faut déterminer quelle taille donner à la zone réallouée à chaque fois. Si on alloue à chaque fois 1 octet de plus, on terminera en n'ayant aucun octet réservé inutile, mais on aura appelé la fonction de réallocation à chaque caractère lu, ce qui est coûteux car cette fonction doit recopier toutes les données de l'ancienne zone mémoire vers la nouvelle.

Au contraire, si à chaque fois qu'on réalloue on double la taille, on va appeler la fonction de réallocation peu souvent mais on aura potentiellement la moitié de la mémoire réservée qui sera inutilisée à la fin du procédé.

En pratique, on utilisera tout de même cette deuxième version : on verra en cours qu'elle est bien plus rapide que la première alternative, et si on voulait vraiment être économe, il suffirait de réallouer à la toute fin du programme un tableau avec juste la taille nécessaire.

Question 3. Implémenter une fonction `int ma_getline(char** buffer, unsigned long int* n, FILE* f)` correspondant à la spécification donnée, en utilisant l'algorithme suggéré.

Question 4. **(Optionnel)** Modifier la fonction pour qu'elle prenne en compte les entrées où `buffer` ne pointe pas vers un pointeur nul, et où l'algorithme n'alloue pas d'espace initial.

Question 5. **(Optionnel)** Implémenter votre propre version de `realloc`.

Exercice 8.

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 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. 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.

Question 1. 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 :

```
grousseau@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, demander une phrase à l'utilisateur, et afficher des lignes de $8K$ caractères représentant la phrase.

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 proposer :

— D'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 $n \times K$

(On pourrait aussi stocker l'image de la phrase directement dans un tableau 2D de taille $8\frac{n}{K} \times 8K$, ce qui simplifiera la fonction d'affichage mais complexifiera la construction du tableau.)

Ainsi, dans notre code source, on peut directement **déclarer** les deux fonctions, sans pour autant les **définir**, et faire l'implémentation dans un 2eme temps, une fois que l'on aura bien cerné la structure de notre programme :

```
1  /* Renvoie le nom du fichier contenu les données nécessaires
2     pour dessiner le caractère c */
3  char* img_filename(char c);
4
5  /* Affiche la matrice G de n lignes et m colonnes par blocs de K colonnes.
6  Par exemple, si G est la matrice suivante:
7  AAABBBCCC
8  AAABBBCCC
9  alors print_bloc(G, 2, 9, 5) affichera:
10 AAABB
11 AAABB
12 BCCC
13 BCCC
14 */
15 void print_bloc(char** G, int n, int m, int K);
```

De manière générale, pour imaginer des fonctions 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 !** Ce fichier contient une proposition d'organisation pour le programme, ainsi qu'une fonction main partielle. Le but de cet exercice est de réfléchir à une organisation du programme par vous-même, et ce fichier est là pour que vous puissiez finir l'exercice si jamais vous ne voyez pas du tout comment procéder.

Question 2. Décrivez l'organisation que vous envisagez pour votre programme, et listez les déclarations de fonctions que vous comptez utiliser dans un fichier. Soyez précis dans vos commentaires, et si vos fonctions ont un effet graphique, n'hésitez pas à mettre dans le commentaire un exemple pour illustrer le fonctionnement.

Question 3. 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`.