Lire et écrire dans un fichier en mode texte

Exercice 1. Dans cet exercice, les fichiers de description de matrice sont organisés de la manière suivante :

- La première ligne est un entier indiquant le nombre de lignes de la matrice.
- La seconde ligne est un entier indiquant le nombre de colonnes de la matrice.
- Les lignes suivantes comportent autant d'entiers que le nombre de colonnes.
- Il y a en tout 2 lignes de plus que le nombre de lignes de la matrice.

Par exemple, le fichier **mat1.txt** contient :

```
2
3
50 12 10
-3 41 15

```

Il désigne la matrice

$$\begin{pmatrix} 50 & 12 & 10 \\ -3 & 41 & 15 \end{pmatrix}$$

Remarque. Connaissant le nombre de lignes et de colonnes on aurait pu se contenter de lister les coefficients sans passage à la ligne.

1. On souhaite vérifier de visu si un fichier est bien une description de matrice. Écrire une fonction `void cat(char * filename)` qui affiche à l'écran le contenu d'un fichier texte.
2. Ecrire une procédure `int ** create_matrix(char* nom, int * nbl, int * nbc)` qui prend en paramètres un nom de fichier de description de matrice et deux pointeurs d'entiers.
   Cette fonction met à jour les pointeurs d'entiers selon les informations lues dans le fichier : le premier pointe sur le nombre de lignes, le second sur le nombre de colonnes.
   La fonction crée ensuite une matrice, met à jour ses coefficients selon les informations lues dans le fichier puis la retourne.
3. Ecrire une procédure `void transpose(char * nom, int ** tab, int nbl, int nbc)` qui prend en paramètres un nom de fichier de sortie, un pointeur de pointeur d'entiers désignant une matrice et les dimensions d'icelle. La procédure écrit dans le fichier de sortie la transposée de la matrice passée en paramètre.

Solution. Voici pour `cat` :

```c
void cat(char * nom){
  FILE * flot = fopen(nom,"r");
  assert(flot!=NULL);
  char c = fgetc(flot);
  while(EOF!=c){
    printf("%c",c);
    c = fgetc(flot);
  }
  fclose(flot);
}
```

```
14
15 int main(int argc, char* argv[]) {
16   assert (argc>1);
17   for (int i=1; i<argc; i++)
18     {
19       printf("%s:\n",argv[i]);
20       cat(argv[i]);
21     }
22 }
```

Et pour les matrices :

```
26 int ** create_matrix(char* nom, int * nbl, int * nbc){
27   FILE* fichier  = fopen(nom, "r");
28   assert ( fichier !=NULL);
29   fscanf( fichier ,"%d", nbl); fscanf( fichier ,"%d", nbc);
30   // initialisation de la matrice
31   int ** mat = malloc ((*nbl) * sizeof(int *));
32   for (int i = 0; i< (*nbl); i++)
33     mat[i] = malloc( *nbc * sizeof(int));
34   //remplissage de la matrice
35   for (int i = 0; i< *nbl; i++){
36         for (int j = 0; j < *nbc; j++)
37    fscanf( fichier ,"%d",&mat[i][j]);// à tester
38   }//for i
39    fclose ( fichier );return mat;
40 }
41
42 void affiche(int ** mat, int nbl, int nbc){// afficher une matrice
43   for (int i = 0; i< nbl; i++){
44         for (int j = 0; j < nbc; j++)
45     printf("%d;",mat[i][j]);
46         printf("\n");
47   }
48 }
49
50 void test_create_matrix(){
51   int nbl, nbc;
52   char * nom = "mat1.txt";
53   int ** mat = create_matrix(nom,&nbl,&nbc);
54    printf("nblignes %d , nbcols %d\n",nbl,nbc);
55    affiche (mat,nbl,nbc);
56   for (int i = 0; i< nbl; i++) free(mat[i]);
57   free(mat);
58 }
59
60 void transpose(char * nom, int ** mat, int nbl, int nbc){
61   FILE* flot  = fopen(nom, "w");
62   if ( flot != NULL){
63         fprintf( flot ,"%d\n",nbc);
64         fprintf( flot ,"%d\n",nbl);
65        for (int j = 0; j < nbc; j++){
66          for (int i = 0; i< nbl; i++){
67     if (i!=nbl-1)
68       fprintf( flot ,"%d ", mat[i][j]);
69     else
70       fprintf( flot ,"%d\n", mat[i][j]);// pas d'espace apès le nombre
71         }
72        }
```

```
73      fclose ( flot );
74   }
75   else  printf("PB\n");
76 }
```

□

Ligne de commandes

Rappels

Le prototype de la fonction `main` est le suivant :

```
1 int main(int argc, char **argv);
```

argc est le nombre d'arguments passés au programme sur la ligne de commandes. Ce décompte inclus le nom du programme tel qu'il est invoqué par l'utilisateur.

argv est un tableau de chaînes de carcatères qui contient tous les arguments passés au programme en ligne de commande. Le premier d'entre-eux est le nom du programme lui même.

Par exemple, avec l'appel :

```
1 ./program hello world
```

- `argc` vaut 3 ;
- `argv[0]` est **"./program"**
- `argv[1]` est **"hello"** ;
- `argv[1]` est **"world"** ;

Il devient alors possible à la fonction `main` d'envoyer les arguments saisis à des fonctions du projet en vue d'un traitement particulier.

Exercices

Exercice 2.

1. Écrire une fonction `void traiter (int argc, char** argv)` qui prend en paramètres un entier `argc` et un tableau de chaînes de caractères (autant que `argc-1`).
   Si le tableau contient plus de deux chaînes, le programme s'arrête sur une erreur d'assertion. Sinon, c'est que le tableau contient exactement 2 chaînes. Ce sont deux noms de fichiers. La fonction copie le contenu du premier dans le second avec ajout à la fin
2. Modifier le `main` de votre projet **copier.c** de sorte que le programme obtenu affiche son propre nom puis lise les noms des deux fichiers en ligne de commande.

```
1 $ cat toto1.txt
2  Toto lé motivé
3 Gogo lé fatigué
4 La Dodo, salé bon mém!
5 $ ./copier toto1.txt toto_copy.txt
6 Nb arguments : 3, fichier source : toto1.txt, cible toto_copy.txt
7 $ cat toto_copy.txt
```

```
Toto lé motivé
Gogo lé fatigué
La Dodo, salé bon mém!

```

Solution. Voici

```
void copier(char * file1, char * file2){
  FILE * f1 = fopen(file1, "r");
  FILE * f2 = fopen(file2, "w");

  char c;
  while(fscanf(f1,"%c",&c) != EOF){
     fprintf(f2,"%c",c);
  }

  fclose(f1); fclose(f2);
}


void traiter (int argc, char** argv);

int main(int argc, char** argv){
  traiter(argc, argv);
  return EXIT_SUCCESS;
}

void traiter (int argc, char** argv){
  if (argc !=3){
       printf("mauvaise syntaxe\n");
       exit(EXIT_FAILURE);
    }

  printf("Nb d'arguments : %d, fichier source : %s, cible %s\n",
    argc, argv[1], argv[2]);

  copier(argv[1], argv[2]);

}
```

□

On rappelle le fonctionnement de la commande `echo` du **shell bash** : elle se contente d'afficher les arguments qu'on lui donne.

```
$ echo coucou
coucou
$ echo $PATH
/home/ivan/anaconda3/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin
```

Remarque. Dans la seconde commande on a demandé à `echo` d'afficher la variable d'environnement `$PATH`. Les répertoires qu'elle indique sont ceux dans lesquel l'OS cherche les exécutables passés en ligne de commande du terminal. Si un exécutable ne se trouve pas dans un répertoire de cette liste, l'OS n'y accède pas sans aide. On indique alors le chemin vers cet exécutable.

C'est la raison pour laquelle `a.out`, pour un programme qui vient d'être compilé dans le répertoire courant, ne produit aucun résultat : en général, le répertoire courant n'est pas listé dans

`$PATH`. Aussi, on aide l'OS en écrivant `./a.out`. Il comprend que l'exécutable est à chercher à la racine `/` du répertoire courant `.` ; en revanche, si l'éxécutable est dans le sous-répertoire **Toto** du répertoire courant, on écrit `./Toto/a.out`.

Exercice 3. Cet exercice est tiré d'un travail similaire de Quentin Fortier.

1. On veut simuler le comportement de `echo` par un programme C écrit dans `echo.c`.
2. La commande `cat` (pour *concatenate*) affiche à l'écran le contenu des fichiers dont les noms sont passés en argument. Lorsqu'il y a plusieurs fichiers, leurs contenus sont listés les uns à la suite des autres.
   Par exemple, si **toto** est un fichier qui contient **"toto 1"**, et si **gogo** contient **"gogo 2"**, alors `cat toto gogo` affiche

```
toto 1
gogo 2

```

   Écrire un programme **cat.c** de sorte à obtenir :

```
$ ./cat toto gogo
toto:
toto 1
gogo:
gogo 2
```

3. On veut émuler le fonctionnement de la commande `grep`. Dans son utilisation la plus simple, celle-ci indique les positions où on peut trouver un mot dans un fichier donné par son nom. Les positions sont ici indiquées par le nombre de caractères depuis le début du fichier parcouru.