Programmation C
TP n° 9 : Manipulation de fichiers

Exercice 1 : Modularisation

Un principe de base de la modularisation des programmes consiste à concevoir des structures de données dans lesquelles on distingue une représentation *externe* visible à l'utilisateur et une représentation *interne* qui devrait être invisible à l'utilisateur. En C, pour pallier à l'absence d'un mécanisme de définition d'une interface on effectue un découpage (assez pénible) du programme en fichiers et on décore certaines fonctions et variables avec le mot `static` ce qui implique qu'elles sont visibles seulement par les fonctions du fichier où elles sont déclarées.

1. On souhaite construire un module `compteur`. Du point de vue de l'utilisateur, ce module comporte deux fonctions :

```
1  void init_compteur();   //initialise le compteur à 0
2  int compteur();         //incrémente le compteur et retourne sa valeur
```

   Créez les fichiers `compteur.c`, `compteur.h`. Créez aussi un fichier `user.c` qui contient le `main` et un fichier `Makefile` pour compiler les 3 fichiers. Assurez-vous qu'un appel à `counter` qui n'est pas précédé par un appel à `init_counter` produit une terminaison du programme avec un message d'erreur et qu'un changement de la mise-en-oeuvre du compteur dans le fichier `counter.c` est transparente pour les fonctions dans le fichier `user.c`.

Exercice 2 : Manipulation de fichiers

1. Pour la manipulation de fichiers, on va se servir des fonctions `fopen`, `fclose`, `fputs` et `fgets` de la librairie `stdio.h`. Écrivez un programme `fill.c` qui attend en argument un nom de fichier et un chiffre entier `n`, crée le fichier correspondant et le remplit de `n` lignes avec un message choisi par vous même sur chaque ligne. (Allez voir la page de manuel de `atoi`). Que se passe-t-il si vous appelez deux fois votre programme avec le même nom de fichier ?
2. Comment modifier votre programme pour que si le fichier donné en argument existe déjà, les lignes soient ajoutées à la fin ?
3. Dans un ficher `fillib.c`, écrivez une fonction `int copy(FILE *fsrc, FILE *fdst)` qui prend deux flots et copie le contenu du premier dans le deuxième. Utilisez les fonctions `fputc` et `fgetc`. On suppose que le premier flot contient du texte. (N'oubliez pas de créer le fichier `fillib.h` correspondant.)
4. Créez un programme `mycp.c` qui prend en argument deux noms de fichiers et copie le premier dans le deuxième (il faut bien entendu utiliser la fonction précédente). Vérifiez que `./mycp fichierin fichierout` se comporte comme `cp`. Tester votre programme sur des fichiers texte et des fichiers binaires, e.g., un executable. Comparer la taille du fichier original avec la taille de la copie.
5. Créez un programme `mycat.c` qui prend en argument un nombre arbitraire de noms de fichier et affiche le contenu des fichiers correspondants sur la sortie standard (l'un après l'autre, sans séparateur). Si aucun argument n'est donné, le programme affiche le contenu de `stdin`. (Pensez à utiliser votre fonction `copy`.) Vérifiez que `./mycat` se comporte comme `cat` (sans options).

Exercice 3 : Vers un éditeur de texte simplissime

Le but de cet exercice est de coder les fonctionalités les plus basiques d'un éditeur de texte : splitting de lignes, fusion de lignes et bufferisation.

1. Ecrivez la fonction `document *decouper_en_lignes(char *name)` qui doit lire le fichier `name` et retourner un `document` tel que :
   - `len` est la longueur du fichier,
   - `txt` est un pointeur vers un tampon qui contient `len+1` caractères, les caractères du fichier suivis du caractère `null`,
   - `nbl` indique le nombre d'éléments dans le vecteur pointé par `lignes`,
   - `lignes` est vecteur qui contient les pointeurs vers le début de chaque ligne de `txt`.

```
typedef struct{
  unsigned len;
  char *txt;
  unsigned nbl;
  char **lignes;
} document;

```

2. Implémentez la fonction `int couper(document *doc, unsigned i, size_t k)` qui sépare la ligne `i` en deux lignes à la position `k`. La fonction retourne `-1` si la ligne n'existe pas ou si la ligne contient moins de `k` caractères, sinon la fonction retourne l'index de la nouvelle ligne créée.
3. Implémentez la fonction `int fusionner(document *doc, int i)`, qui supprime `\n` à la fin de la ligne `i`, ce qui fait que la ligne `i` colle avec la ligne suivante. Pensez à mettre à jour les pointeurs dans le vecteur `lignes`. La fonction retourne le nouveau nombre de lignes, `-1` si la ligne demandée n'existe pas.
4. Implémentez une fonction `void sauvegarder(document *doc, char *name)` permettant de sauvegarder le document dans le fichier.
5. Le site Projet Gutenberg [1] propose une vaste collection d'eBooks librement accessibles. Récupérez un fichier de taille env. 200kB au format Plain Text UTF-8 pour tester votre programme. Par exemple, les anciens terminaux étaient souvent limités à 80 caractères par ligne, vous pouvez proposer un programme qui convertit l'eBook de votre choix en un eBook où chaque ligne contient au plus 80 caractères. Fusionnez les lignes contigües tant que possible sans dépasser la limite de 80 caractères, ceci permettant un affichage plus compact. En vous aidant d'une fonction `afficher(document *doc, unsigned l, unsigned k)` qui affiche les lignes d'indice compris entre l et k, testez vos fonctions en affichant les lignes autour des lignes modifiées et comparer (visuellement) les résultas.

1. `https://www.gutenberg.org`