

TP : Entrées-sorties en C et Ocaml

Avant toute chose, récupérer les fichiers à votre nom dans les ressources pédagogiques (répertoire TP_IO).

Cours : rappels succincts de syntaxe

Pour plus amples détails, consulter la documentation des fonctions ci-après.

En C

- Ouverture d'un fichier : `fopen(<chemin fichier>, <mode de lecture>)`. Le premier argument est une chaîne de caractère correspondant au chemin du fichier et le deuxième une chaîne de caractères valant "w" pour ouvrir le fichier en écriture et "r" pour ouvrir le fichier en lecture. Si on ouvre un fichier qui n'existe pas en écriture, il est créé ; s'il existe, son contenu sera écrasé. La valeur de retour de `fopen` est de type `FILE*`.
- Fermeture d'un flux `f` (de type `FILE*`) : `fclose(f)`.
- Ecriture dans un fichier : `fprintf(<flux sur lequel écrire>, <chaîne formatée>, ...)` S'utilise comme `printf` sauf qu'on précise en premier argument où écrire.
- Lecture dans un fichier : `fscanf(<flux sur lequel lire>, <chaîne formatée>, <adresse1>, ..., <adressen>)`. Par exemple, si on sait qu'une ligne contient deux entiers séparés par un espace et que `a` et `b` sont de type `int`, `fscanf(f, "%d %d\n", &a, &b)` permet de les récupérer dans `a` et `b`. La valeur de retour de `fscanf` est un entier correspondant au nombre de paramètres correctement extraits (dans l'exemple si tout se passe bien cette valeur serait 2). Si aucune donnée ne peut être extraite, EOF est renvoyé.

En Ocaml

- Ouverture d'un fichier en lecture : `let f = open_in <chemin fichier>`. Si le fichier n'existe pas, lève l'exception `Sys_error "No such file or directory"`.
- Fermeture de ce flux : `close_in f`.
- Ouverture d'un fichier en écriture : `let f = open_out <chemin_fichier>`. Si le fichier n'existe pas, il est créé. Si le fichier existe déjà, le contenu sera écrasé.
- Fermeture de ce flux : `close_out f`.
- Lecture d'une ligne sur le canal `f` : `input_line f`. La ligne est "consommée" : le prochain appel à `input_line` donnera la chaîne correspondant à la ligne suivante. Si toutes les lignes ont été consommées, lève l'exception `End_of_file`.
- Ecriture de `s` sur un canal de sortie `f` : `output_string s f`. On peut aussi utiliser `Printf.fprintf f "%s" s`.

Partie 1 : déchiffrer les consignes

Cette partie est à traiter en C. Vous avez à votre disposition un fichier `Consignes_chiffre.txt`, pour le moment illisible. Ce document renferme les consignes permettant d'aborder les parties suivantes. Il ne contient que des lettres minuscules non accentuées et des symboles de ponctuation et blancs.

Chaque lettre minuscule de ce document a été chiffrée à l'aide d'un chiffrement de César : on s'est donné un entier d fixe, appelé *décalage* et pour chaque lettre ℓ dans le texte d'origine (dit : texte clair) le traitement suivant a été appliqué :

- Calculer la position $i \in \llbracket 0, 25 \rrbracket$ de ℓ dans l'alphabet.
- Calculer $i' = (i + d) \bmod 26$.
- Déterminer la lettre ℓ' dont la position est i' dans l'alphabet.
- Remplacer la lettre ℓ dans le document d'origine par ℓ' .

Les caractères autres que les lettres minuscules ne sont pas chiffrés. Le décalage de ce chiffrement est donné par le nombre d'espaces dans `Consignes_chiffre.txt`.

1. Ecrire une fonction `int compte_espaces(char* nom_fichier)` prenant en entrée le nom d'un fichier et déterminant le nombre de caractères espace qu'il contient.
2. Ecrire une fonction `char decale(char c, int d)`. Si c est une lettre minuscule, elle renvoie le caractère clair correspondant après déchiffrement de César de décalage d . Par exemple, si c vaut 'b' et d vaut 5, `decale(c,d)` doit valoir 'w'. Si c n'est pas une lettre minuscule, cette fonction le renvoie tel quel.

Indication : Lorsque i est négatif, $i\%26$ est négatif...

3. Ecrire une fonction `void dechiffre_cesar(char* nom_chiffre, char* nom_clair, int d)` qui lit les caractères du fichier `nom_chiffre` (supposés chiffrés à l'aide d'un chiffrement de César de décalage d selon les mêmes modalités que `Consignes_chiffre.txt`) un à un et écrit le caractère clair correspondant dans le fichier `nom_clair`.
4. Utiliser les fonctions précédentes pour stocker dans un fichier `Consignes_clair.txt` les consignes une fois déchiffrées à partir de `Consignes_chiffre.txt`.

Partie 2 : trouver le bon fichier

Cette partie est à traiter en C.

5. Ecrire une fonction `void trouve_message(char* nom_fichier, char d)` qui parcourt le fichier dont le nom est `nom_fichier` et affiche un à un tous les caractères de ce fichier situés entre deux occurrences du caractère délimiteur d .
6. Utiliser cette fonction et les consignes pour progresser dans votre quête.

Partie 3 : trouver la clé et conclure

Cette partie est à traiter en Ocaml.

7. Ecrire une fonction `fichier_to_string : string -> string` prenant en entrée le nom d'un fichier et renvoyant la chaîne de tous les caractères qu'il contient.

Attention : La fonction `input_line` renvoie le contenu de la ligne lue sauf le caractère '`\n`' final. Par ailleurs, attention au cas où le fichier commence par une ligne ne contenant que '`\n`'.

8. Ecrire une fonction `somme : string -> char -> int` telle que `somme s c` renvoie la somme de tous les codes ASCII des caractères suivant un caractère `c` dans la chaîne `s`.
9. Ecrire une fonction `contient_cle : string -> (char*char) option` tel que `contient_cle nom_fichier` renvoie `None` si le fichier ne contient pas de morceau de clé selon les spécifications données par les consignes et `Some (clair, chiffre)` sinon.

Les consignes nous indiquent que tôt ou tard on devra opérer à un déchiffrement monoalphabétique. On propose pour faciliter cette opération de stocker la clé de déchiffrement dans une table de hachage. Une clé dans cette table sera un caractère chiffré et la valeur correspondante le caractère clair correspondant. Par exemple, si notre table contient une association entre '`t`' et '`y`', cela signifie que pour déchiffrer il suffira de remplacer toutes les occurrences de '`t`' par '`y`'.

10. Ecrire une fonction `construire_cle : string -> (char,char) Hashtbl.t` prenant en entrée un nom de répertoire et construisant une table de hachage correspondant à la clé de déchiffrement recherchée selon les modalités données par les consignes.

Indication : C'est ici que `Sys.readdir` va servir ! Attention à utiliser les bons chemins.

11. Ecrire une fonction `dechiffrer_chaine : string -> (char,char) Hashtbl.t -> string` qui à partir d'une chaîne chiffrée et d'une clé de déchiffrement produit la chaîne déchiffrée.
12. Ecrire une fonction `dechiffrer_fichier : string -> string -> string -> unit` prenant en entrée le nom d'un répertoire contenant les fichiers donnant les morceaux de clé, le nom d'un fichier chiffré, le nom d'un fichier (destiné à être le clair correspondant au chiffré) et qui écrit dans ce dernier le texte clair correspondant au texte chiffré en appliquant les consignes.
13. Lire le message clair enfin obtenu. Profit !