1 Introduction

Ce TP a pour but de vous faire manipuler des chaînes de caractères et de vous introduire les fichiers (en bonus). Pour cela, je vous propose de faire un petit peu de cryptologie. La cryptologie, ou *science du secret*, englobe la cryptographie et la cryptanalyse.

La cryptographie est l'art de protéger des messages à l'aide de *secrets* ou de *clés*. La cryptographie ne doit pas être confondue avec la stéganographie, qui consiste à dissimuler un message plutôt qu'à le rendre incompréhensible à ceux ne connaissant pas le *secret*.

D'après Wikipedia, en cryptologie, un *chiffre* est une manière secrète d'écrire un message à transmettre, au moyen de caractères et de signes disposés selon une convention convenue au préalable. Plus précisément, le chiffre est l'ensemble des conventions et des symboles (lettres, nombres, signes, etc.) employés pour remplacer chaque lettre du message à rendre secret. Avec un chiffre, on transforme un *message en clair* en *message en chiffres*, ou *message chiffré*, ou encore *cryptogramme*.

La cryptanalyse consiste à essayer de comprendre un message chiffré sans en posséder la clé de déchiffrement. On appelle *attaque* le fait de tenter de comprendre le contenu d'un message chiffré.

2 Balbutiements

Créez tout d'abord un fichier **crypto.c** qui comporte une fonction **main** et qui compile correctement.

2.1 Affichage et saisie de chaînes de caractères

Au cours de ce TP, il est probable que vous souhaitiez saisir des chaînes de caractère afin de tester vos fonctions. Le code d'exemple ci-dessous vous montre comment créer une chaîne de caractères, comment la saisir grâce à la fonction **fgets** et comment l'afficher grâce à la fonction **printf**. Le fonctionnement de la fonction **fgets** est détaillé en section 3.5 (vous pouvez également vous fier au manuel en tapant la commande **man fgets** dans votre terminal).

```
const int taille = 128;
char chaine[taille];
printf("Veuillez entrer une chaîne : ");
fgets(chaine, taille, stdin);
printf("Vous avez saisi \"%s\"", chaine);
```

2.2 Les fonctions est_majuscule et est_minuscule

```
bool est_majuscule(char c);
bool est_minuscule(char c);
```

Implémentez les fonctions **est_majuscule** et **est_minuscule**. La première renvoie vrai si et seulement si le caractère **c** qu'elle reçoit comme paramètre est une lettre majuscule. La seconde renvoie vrai si et seulement si le caractère **c** qu'elle reçoit comme paramètre est une lettre minuscule. Exemple d'appels de ces fonctions :

```
est_majuscule('a');  // Renvoie false
est_minuscule('a');  // Renvoie true
est_majuscule('X');  // Renvoie true
est_minuscule('X');  // Renvoie false
est_majuscule(' ');  // Renvoie false
est_minuscule(' ');  // Renvoie false
```

2.3 La fonction taille_chaine

```
int taille_chaine(const char * str);
```

Implémentez la fonction **taille_chaine**, qui renvoie la taille de la chaîne valide **str**. Rappel : une chaîne valide est terminée par un caractère nul ('\0'). Rappel : la taille d'une chaîne de caractères est le nombre de caractères précédant le '\0'. Voici des résultats attendus par la fonction :

```
taille_chaine(""); // Renvoie 0
taille_chaine("Bouh !"); // Renvoie 6
```

2.4 La fonction en_minuscules

```
void en_minuscules(char * str);
```

Implémentez la fonction **en_minuscules**, qui modifie la chaîne valide de caractères **str** de telle sorte que les lettres initialement en majuscules de **str** soient en minuscules après l'appel de la fonction. Attention, il ne faut pas énumérer tous les cas mais se servir des spécificités de la table ASCII afin de réaliser cette conversion. Attention : les caractères qui ne sont pas des lettres ne doivent pas être modifiés par cette fonction. Voici un exemple d'appel de cette fonction :

```
char s[] = "Crieur : JE CRIE FORT !!!!!";
en_minuscules(s);
// Maintenant, s vaut "crieur : je crie fort !!!!!"
```

2.5 La fonction copie_chaine

```
void copie_chaine(char * dest, const char * src);
```

Implémentez la fonction **copie_chaine**, qui copie le contenu de la chaîne valide **src** dans **dest**. On suppose qu'il y a assez de place pour copier **src** dans **dest**. Attention : **dest** doit être une chaîne **valide** après l'appel de cette fonction (autrement dit, **dest** doit finir par un '\0'). Exemple d'appel de la fonction :

```
const char * s1 = "Okonomiyaki";
char s2[32];
copie_chaine(s2, s1);
// Maintenant, s2 vaut "Okonomiyaki". Ce qui implique
// s2[0] vaut 'O', s2[1] vaut 'k', s2[2] vaut 'o'
// ... s2[9] vaut 'k', s2[10] vaut 'i' s2[11] vaut '\0'
```

3 Le chiffrement par décalage

Le chiffrement par décalage est une des plus simples méthodes de chiffrement. Pour obtenir un *message chiffré* à partir d'un *message en clair*, il suffit de remplacer chaque caractère du *message en clair* par un autre caractère, qui est à distance fixe du premier caractère. Par exemple, si on souhaite effectuer un décalage de trois caractères vers la droite, 'a' devient 'd', 'f' devient 'i', 'z' devient 'c'... Ce décalage est représenté sur la figure 1 (page 3). La position des caractères les uns par rapport aux autres n'est pas changée par cette méthode de chiffrement.

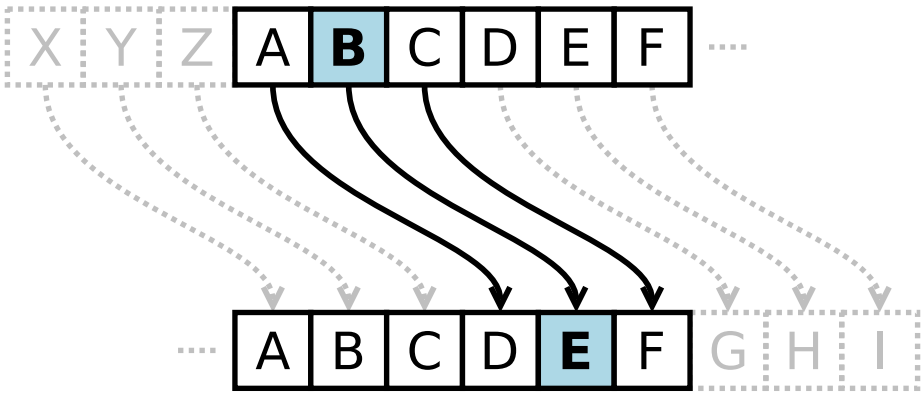

FIGURE 1 – Représentation du chiffrement par décalage. Ici, le décalage est de trois caractères vers la droite

3.1 La fonction shift_char

```
char shift_char(char c, int offset);
```

Implémentez la fonction **shift_char**, qui renvoie le résultat d'un décalage du caractère **c** d'une longueur **offset**. Le paramètre **offset** est un entier dans $[-25, 25]$. Si **c** n'est pas une lettre minuscule, la fonction doit renvoyer **c**. Attention : vous devez vous assurer que les lettres transformées restent des lettres. Deux cas spéciaux sont à traiter pour cela :

— lorsque offset > 0 et qu'on *dépasse* 'z', il faut continuer par 'a' et non par '{',
— lorsque offset < 0 et qu'on *dépasse* 'a', il faut continuer par 'z' et non par '`'.

Voici un exemple du résultat attendu de cette fonction :

```
shift_char('a',  3); // Renvoie 'd'
shift_char('y',  3); // Renvoie 'b'
shift_char(' ', -3); // Renvoie ' '
shift_char('b',  2); // Renvoie 'd'
shift_char('3',  3); // Renvoie '3'
shift_char('b', -3); // Renvoie 'y'
shift_char('.', 25); // Renvoie '.'
```

3.2 La fonction shift_cipher

```
void shift_cipher(const char * clear, char * ciphered, int offset);
```

Implémentez la fonction **shift_cipher**, qui reçoit en entrée un *message en clair* **clear** et une longueur de décalage **offset**. Le paramètre **offset** est un entier dans $[-25, 25]$. Cette fonction modifie son paramètre **ciphered** de telle sorte qu'après l'appel de la fonction, **ciphered** contienne le *message chiffré* correspondant au chiffrément par décalage de longueur **offset** du *message en clair* **clear**. Pour cela, effectuez les opérations suivantes :

— Copiez **clear** dans **ciphered**,
— Mettez **ciphered** en minuscules,
— Réalisez le chiffrement en lui-même sur **ciphered**.

Voici un exemple d'appel de cette fonction :

```
const char * s = "Juste du texte, qui sert d'exemple...";
char s2[64];
shift_cipher(s, s2, 0);
// s2 vaut "juste du texte, qui sert d'exemple..."
shift_cipher(s, s2, 3);
// s2 vaut "mxvwh gx whawh, txl vhuw g'hahpsoh..."
shift_cipher(s, s2, -10);
// s2 vaut "zkiju tk junju, gky iuhj t'unucfbu..."
```

3.3 La fonction shift_decipher

```
void shift_decipher(const char * ciphered, char * clear, int offset);
```

Implémentez la fonction **shift_decipher**, qui reçoit en entrée un *message chiffré* **ciphered** et une longueur de décalage **offset**. Le paramètre **offset** est un entier dans $[-25, 25]$. Cette fonction modifie son paramètre **clear** de telle sorte qu'après l'appel de la fonction, **clear** contienne le *message en clair* correspondant au déchiffrément par décalage de longueur **offset** du *message chiffré* **ciphered**. Note : vous pouvez directement vous servir de la fonction **shift_cipher** pour réaliser cette opération. Exemple d'utilisation de la fonction :

```
char s[64];
shift_decipher("Juste du texte, qui sert d'exemple...", s, 0);
// s vaut "Juste du texte, qui sert d'exemple..."
shift_decipher("mxvwh gx whawh, txl vhuw g'hahpsoh...", s, 3);
// s vaut "Juste du texte, qui sert d'exemple..."
shift_decipher("zkiju tk junju, gky iuhj t'unucfbu...", s, -10);
// s vaut "Juste du texte, qui sert d'exemple..."
```

3.4 La fonction decrypt_shift_bf

La longueur du décalage, dans le chiffrement par décalage, peut être vue comme la *clé de chiffrement*. Ce chiffre n'offre qu'un nombre très restreint de clés possibles : les nombres entiers entre 0 et 25. En effet, le décalage étant circulaire, faire un décalage de 273456 vers la droite revient à faire un décalage de 14 vers la droite car $273456 \equiv 14 \ (mod\ 26)$. De plus, faire un décalage de 4 vers la gauche revient à faire un décalage de 22 vers la droite car $-4 \equiv 22 \ (mod\ 26)$.

Puisque le nombre de clés est très petit, il est facile de tester toutes les clés possibles et d'afficher le texte décrypté correspondant à chaque fois. Il suffit ensuite de lire (manuellement) les messages et de trouver (manuellement) le message en clair au milieu de tous les autres. Le fait de tester toutes les possibilités est communément appelé une attaque par force brute (ou *brute-force*).

```
void decrypt_shift_bf(const char * ciphered);
```

Implémentez la fonction **decrypt_shift_bf**, qui, pour chaque clé entière dans $[0, 25]$, affiche le message décrypté correspondant. Voici un exemple d'appel de cette fonction :

```
decrypt_shift_bf("redzekh");
/* Affiche :
    clé= 0, texte="redzekh"
    clé= 1, texte="qdcydjg"
    clé= 2, texte="pcbxcif"
    ...
    clé=16, texte="bonjour"
    ...
    clé=24, texte="tgfbgmj"
    clé=25, texte="sfeafli" */
```

3.5 La fonction decalage

```
void decalage(void);
```

La fonction **decalage** va appeler les différentes fonctions que vous avez créées pour réaliser un chiffrement par décalage. La fonction **decalage** doit être appelée depuis votre fonction **main**. Les affichages de votre fonction doivent être identiques à ceux de l'exemple d'exécution de la figure 2 (page 7). Le fonctionnement de cette fonction est le suivant :

1. Demande à l'utilisateur de la saisie d'un *message en clair* (une chaîne de caractères). Pour cela, servez-vous de la fonction **fgets** (décrite ci-dessous).
2. Demande à l'utilisateur de la saisie d'une longueur de décalage. Effectuez la saisie grâce à la fonction **scanf**. Vous devez vous assurer que cette longueur est valide (*i.e.* qu'elle est dans $[-25, 25]$).
3. Chiffrement de la chaîne saisie par l'utilisateur grâce à un chiffrement par décalage de la longueur indiquée par l'utilisateur. Affichez le *message chiffré*.
4. Déchiffrement du *message chiffré* avec la longueur indiquée par l'utilisateur. Affichez le *message en clair* ainsi obtenu. Vérifiez bien que ce message est identique (à la casse près) à celui entré initialement par l'utilisateur.
5. Création d'une chaîne de caractères *mystere* qui vaut "Zp cvbz hyypclg h spyl jl aleal, cvayl mvujapvu zopma_kljpwoly zltisl thyjoly. Cvbz klclg jlwlukhua cvbz hzzbyly xbl jl aleal lza spzpisl bupxbltlua svyzxbl cvbz hwwlslg zopma_kljpwoly hclj bu vmmzla lnhs h zlwa (vb tvpuz kpeulbm)".
6. Appel de votre fonction **decrypt_shift_bf** sur cette chaîne. Observez le résultat. Une des lignes doit être en français et contient la suite des instructions pour cette fonction. Si aucune ligne n'est intelligible, votre code ne marche pas :(.

```
#include <stdio.h>
char *fgets(char *s, int size, FILE *stream);
```

La fonction **fgets** permet de lire une chaîne de caractères depuis un fichier **stream**. Ici, nous voulons lire depuis l'entrée standard, il faut donc donner **stdin** comme troisième paramètre à cette fonction. Le paramètre **s** de cette fonction est un tableau de caractères que vous devez créer vous-même avant d'appeler cette fonction. C'est dans **s** que la saisie sera stockée. Le paramètre **size** limite la taille de la chaîne de caractères à lire. Dans ce cas, vous devez utiliser la taille du tableau de caractères que vous avez créé. La fonction **fgets** renvoie le texte saisi par l'utilisateur, s'arrêtant au premier '\n' rencontré ou au $size - 1$-ème caractère si aucun '\n' n'a été rencontré. La chaîne renvoyée est une chaîne valide (terminée par un '\0'). La fonction **fgets** renvoie *NULL* si

et seulement si une erreur de saisie s'est produite. Note : cette fonction stocke le '\n' à la fin de la chaîne de caractères si elle en a rencontré un. Si vous souhaitez enlever ce saut de ligne, il vous faudra modifier le dernier caractère de la chaîne et le remplacer par un '\0'. Un exemple d'utilisation de fgets se trouve ci-dessous :

```
char input[256];
char * ret = fgets(input, 256, stdin);
if (ret != NULL)
{
    // La saisie a fonctionné
}
```

```
Entrez un message en clair : Mais qu'est-ce donc ?
Entrez une clé : 42
Entrée invalide, 42 n'est pas dans [-25, 25].
Entrez une clé : 5

Message chiffré : "rfnx vz'jxy-hj itsh ?"
Message déchiffré : "mais qu'est-ce donc ?"

clé= 0, texte="zp cvbz hyypclg h spyl jl aleal [...] kpe-ulbm)"
clé= 1, texte="yo buay gxxobkf g roxk ik zkdzk [...] jod-tkal)"
...
clé=25, texte="aq dwca izzqdmh i tqzm km bmfbm [...] lqf-vmcn)"
```

FIGURE 2 – Exemple d'exécution de la fonction **decalage**. Votre affichage doit être le plus proche possible de celui-ci. Les caractères en gras sont ceux que l'utilisateur a saisi au clavier.

4 Déchiffrement par analyse fréquentielle

Dans cette partie, nous allons jouer le rôle d'un cryptanalyste. Vous pouvez trouver ici le fichier **whatami.txt**. Ce fichier est chiffré par une méthode de substitution monoalphabétique. Cette méthode consiste à remplacer chaque lettre *claire* par une lettre *chiffrée*, de telle sorte qu'une même lettre *claire* soit toujours remplacée par la même lettre *chiffrée*. L'ordre des lettres reste inchangé. Le chiffrement par décalage que nous venons de voir est un cas particulier de chiffrement par substitution monoalphabétique.

La clé d'un chiffrement par substitution monoalphabétique est une chaîne de caractères de taille 26. On peut par exemple prendre la clé "azertyuiopqsdfghjklmwxcvbn", ce qui veut dire que la lettre 'a' est chiffrée en 'a', la lettre

'b' en 'z', la lettre 'c' en 'e', ..., la lettre 'z' en 'n'. Ainsi, avec une telle clé, le message "bouh" est chiffré en "zgwi". Cette substitution est representée en figure 3.

```
lettre claire :   abcdefghijklmnopqrstuvwxyz
lettre chiffrée : azertyuiopqsdfghjklmwxcvbn
```

FIGURE 3 – Une substitution monoalphabétique. Chaque lettre claire est toujours chiffrée dans la même lettre chiffrée. Par exemple, 'k' est toujours chiffrée en 'q'. La clé utilisée ici est "azertyuiopqsdfghjklmwxcvbn".

Une méthode pour casser ce type de chiffrement est l'analyse fréquentielle.

4.1 Analyse de fréquence

D'après Wikipedia, l'analyse de fréquence (ou *analyse fréquentielle*) est une méthode de cryptanalyse qui consiste à examiner la fréquence des lettres employées dans un message chiffré. On peut ensuite comparer ces fréquences à celles utilisées dans la langue que nous pensons être utilisée dans le *message en clair*. Cette méthode est souvent employée pour décrypter des messages chiffrés par substitution.

```
void freq_analysis(const char * str, double * freq);
```

La fonction **freq_analysis** reçoit en entrée une chaîne de caractères valide **str**. On suppose que toutes les lettres de **str** sont en minuscule. Cette fonction calcule la fréquence d'apparition de chaque lettre et stocke le résultat dans le tableau **freq** (que l'on suppose de taille au moins 26). La fréquence d'apparition d'une lettre est un nombre flottant situé entre 0 (jamais apparu) et 1 (présent partout). On souhaite qu'après l'appel de la fonction, freq[0] contienne la fréquence d'apparition de la lettre 'a', freq[1] celle de la lettre 'b', ..., freq[25] celle de la lettre 'z'. Pour implémenter la fonction **freq_analysis**, je vous propose d'utiliser l'algorithme suivant :

- Créez un tableau d'entiers **nb_occ** de taille 26. Initialisez toutes les cases du tableau à 0.
- Créez une variable **nb_lettres** initialisée à 0.
- Parcourez la chaîne **str** en
  - incrémentant **nb_lettres** dès que vous rencontrez une lettre,
  - incrémentant une case de **nb_occ** dès que vous rencontrez une lettre. Incrémentez la première case si vous rencontrez un 'a', la troisième si vous rencontrez un 'c'...
- Une fois la chaîne **str** parcourue, vous pouvez calculer la fréquence d'apparition de chaque caractère (**freq**) en divisant le nombre d'occurrences de chaque caractère (**nb_occ**) par le nombre de lettres lues (**nb_lettres**).

4.2 Lecture d'un fichier

Dans cette partie, on s'intéresse à la lecture d'un fichier. Le contenu d'un fichier peut être vu comme une longue séquence de bits. Les bits étant peu lisibles par les humains, on préfère les agréger en octets afin de les manipuler. On peut dire qu'il existe deux grands types de fichiers :

— les fichiers textuels dont le contenu est du texte. Ils peuvent être affichés sans outil particulier (*cat* ou *less* sous Linux par exemple). Ils peuvent être édités par un éditeur de texte (*nano*, *vim*, *emacs*, *Sublime Text*...). Exemple : les fichiers texte brut, les fichiers de code source (en C ou n'importe quel autre langage), le fichier LaTeX qui sert à faire le présent document...

— Les fichiers binaires, dont le contenu n'est pas lisible en tant que texte directement. Exemple : une image au format PNG, un fichier exécutable, une archive zip... Certains de ces fichiers peuvent être édités par des outils qui leur sont propres (un logiciel de retouche d'image pour une image PNG, un logiciel de traitement vidéo pour un fichier de vidéo...). On peut manipuler ces fichiers à l'octet près grâce à un éditeur hexadécimal.

Dans cet exemple, nous allons uniquement manipuler un fichier textuel.

4.2.1 La fonction filename_to_string

```
bool filename_to_string(const char * filename,
                        char * content,
                        int buf_size);
```

La fonction **filename_to_string** ouvre le fichier **filename**, lit son contenu et le stocke dans la chaîne **content** puis ferme le fichier. On suppose que la zone mémoire dans laquelle **content** est situé peut stocker **buf_size** caractères au maximum. Cette fonction renvoie vrai si et seulement si le fichier a pu être ouvert, lu, copié en entier dans **content** et fermé. Attention : **content** doit dans tous les cas être une chaîne valide après l'appel de votre fonction. Afin d'implémenter cette fonction, voici les fonctions de la bibliothèque standard dont vous devez vous servir :

```
#include <stdio.h>
FILE *fopen(const char *path, const char *mode);
int fgetc(FILE *stream);
int fclose(FILE *stream);
```

La fonction **fopen** permet d'ouvrir le fichier dont le nom est **path**. On peut spécifier plusieurs modes d'ouverture à cette fonction mais seul le mode "r" nous intéresse ici puisqu'on veut uniquement lire le fichier et non modifier son contenu. Cette fonction renvoie un pointeur nul si le fichier n'a pas pu être ouvert.

La fonction **fgetc** permet de lire un caractère depuis un fichier. Cette fonction renvoie le caractère lu en temps normal. Elle peut également renvoyer EOF si on a atteint la fin du fichier ou qu'une erreur de lecture s'est produite.

La fonction **fclose** permet de fermer un fichier qui a été ouvert avec **fopen**. Elle renvoie 0 si le fichier a bien pu être fermé et EOF sinon.

Voici un exemple de manipulation de fichier :

```
const char * nom_fichier = "./fichier_texte.txt";
FILE * fichier = fopen(nom_fichier, "r");
if (fichier != NULL)
{
    printf("Le fichier '%s' a été ouvert avec succès.\n", nom_fichier);
    printf("Contenu du fichier :\n");
    printf("####################\n");
    char c = 'a';
    while (c != EOF)
    {
        c = fgetc(fichier);
        if (c != EOF)
            printf("%c", c);
    }
    printf("####################\n");
    printf("Fin du contenu du fichier '%s'.\n", nom_fichier);
    if (fclose(fichier) != 0)
        printf("Impossible de fermer le fichier '%s'\n", nom_fichier);
}
else
    printf("Impossible d'ouvrir le fichier '%s' en lecture.\n", nom_fichier);
```

Implémentez la fonction **filename_to_string**.

4.2.2 L'appel de filename_to_string depuis le main

Puisqu'un fichier peut être assez volumineux, il vaut mieux éviter de le stocker dans la pile. Nous verrons plus tard dans le cours comment allouer des données dans le tas, mais pour que vous puissiez tout de même faire ce TP je vous propose un bout de code qui permet d'appeler la fonction **filename_to_string**.

```
#include <stdlib.h>
// ...
const int buf_size = 1024 * 192;
char dkey[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
char * file_content = malloc(sizeof(char) * buf_size);
char * deciphered = malloc(sizeof(char) * buf_size);
if (filename_to_string("./whatami.txt", file_content, buf_size))
{
    en_minuscules(file_content);
    // ... : Manipulation de file_content
}
```

```
free(file_content);
free(deciphered);
// ...
```

Cet extrait de code permet de créer des chaînes de caractères dans le tas grâce à la fonction **malloc**, appelle notre fonction puis libère les chaînes de caractères du tas grâce à la fonction **free**. Les tableaux ainsi créés sont de taille 192 kio, ce qui devrait être suffisant pour lire ce fichier. Attention : ce code lit un fichier "./whatami.txt". Il faut donc que votre fichier **whatami.txt** soit dans le même répertoire que celui depuis lequel vous exécutez votre programme pour que la lecture soit réalisée avec succès.

4.3 Déchiffrement monoalphabétique

Dans cette partie, on s'intéresse à la réalisation d'un déchiffrement par substitution monoalphabétique.

4.3.1 La fonction monoalphabetic_decipher

```
void monoalphabetic_decipher(const char * ciphered,
                             const char * dkey,
                             char * clear);
```

Dans cette partie, il vous est demandé d'implémenter la fonction **monoalphabetic_decipher**, qui déchiffre la chaîne chiffrée **ciphered** en utilisant la clé de déchiffrement **dkey** et stocke le résultat dans **clear**. On suppose que chaque lettre de **ciphered** est en minuscules. On suppose que **dkey** est une chaîne de caractères valide de taille 26 et est une permutation des lettres minuscules de l'alphabet. La chaîne **clear** est supposée pouvoir stocker au moins le même nombre de caractères que ceux de **ciphered**. Pour implémenter cette fonction, je vous propose de faire les actions suivantes :

- Parcourez la chaîne ciphered caractère par caractère.
  - Si le caractère n'est pas une lettre, copiez-le à l'identique dans **clear**.
  - Si le caractère est une lettre, appliquez la clé **dkey** sur le caractère. Par exemple, si la clé de déchiffrement est "aywmcnophqrstzijkdlegxuvfb", un 'a' chiffré devient un 'a' clair, un 'b' devient un 'y' clair, ..., un 'y' chiffré devient un 'f' clair, un 'z' chiffré devient un 'b' clair. Placez alors la lettre claire dans **clear**.

4.4 Trouver la clé utilisée pour chiffrer whatami.txt

Dans cette partie, il faut trouver la clé de déchiffrement du fichier **whatami.txt**. Cette partie est plus exploratoire que les précédentes. Pour vous aider, je vous propose de partir du code de la figure 4. Ce code initialise votre clé de déchiffrement à "ABCDEFGHIJKLMNOPQRSTUVWXYZ", appelle la fonction **monoalphabetic_decipher** et affiche le début du message déchiffré par **dkey**. Si votre clé est la bonne, le message déchiffré sera égal au message

original (et donc compréhensible). Je vous propose d'écrire, dans **dkey**, les caractères inconnus (ceux dont vous ne connaissez pas l'équivalent clair) par des majuscules et ceux connus par des minuscules.

```
char dkey[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
monoalphabetic_decipher(file_content, dkey, deciphered);
printf("%.*s\n", 1000, deciphered);
```

FIGURE 4 – Une base de code pour rechercher la clé de déchiffrement du fichier **whatami.c**.

Je vous propose maintenant de faire une analyse de fréquence des lettres présentes dans le fichier **whatami.txt**. Appelez la fonction **freq_analysis** sur la chaîne lue depuis le fichier **whatami.txt** et affichez son résultat. Cette analyse fréquentielle devrait vous permettre de voir quelles sont les lettres les plus fréquentes et les moins fréquentes du fichier. Sachant que ce fichier est un livre plutôt connu écrit en anglais, vous pouvez comparer ces résultats à la fréquence d'apparition des lettres en anglais. Cela devrait vous permettre de trouver les lettres les plus fréquentes et les moins fréquentes. Vous pouvez modifier la clé en conséquence. Par exemple, s'il s'avère que 'Z' est la lettre la plus fréquente dans le texte, il est possible que 'Z' soit l'image d'un 'e' du texte original. Vous pouvez ainsi, dans **dkey**, remplacer 'Z' par 'e'. Vous pouvez ensuite relancer votre programme pour voir si cette transformation semble cohérente ou non. Si c'est le cas, vous pouvez continuer à rechercher des lettres. Sinon, annulez votre changement dans la clé.

Pour trouver la clé, vous aurez besoin de certaines astuces... Vous pouvez par exemple essayer de trouver des mots dans le message déchiffré, ce qui peut rapidement vous faire trouver la clé. Par exemple si vous voyez souvent une suite de trois lettres "XYZ" dans le message chiffré et que X et Z sont des caractères très fréquents, il est probable que "XYZ" représente "the" et donc que 'X' soit l'image d'un 't', 'Y' soit l'image d'un 'h' et que 'Z' soit l'image d'un 'e'. Cette liste recense les mots les plus courants en anglais et devrait pouvoir vous aider.

De plus, le début du fichier indique des informations sur le livre comme son titre ou son auteur, ce qui devrait vous permettre de trouver rapidement la clé de déchiffrement. Les derniers caractères seront plus difficiles à trouver puisqu'ils sont peu fréquents. Le plus simple pour les trouver est probablement d'afficher toute la chaîne **deciphered** dans votre programme et d'appeler différemment votre programme. Si vous recherchez un 'Z' vous pouvez par exemple exécuter :

```
gcc -std=c99 -Wall -Wextra -Wfatal-errors crypto.c -o crypto
./crypto | grep --color=auto 'Z'
```

Bonne recherche !

5 Bonus culturel

Si la cryptologie vous intéresse, je vous conseille de lire le livre [1] qui montre le rôle important que la cryptologie a joué dans l'histoire et explique de manière simple et pédagogique certaines méthodes de chiffrement.

Si les aspects plus pratiques de la sécurité informatique vous intéressent, je ne peux que vous conseiller le livre [2]. De plus, si vous cherchez des défis en sécurité, ce site peut vous intéresser.

Références

[1] S. Singh and C. Coqueret. *Histoire des codes secrets : de l'Égypte des Pharaons à l'ordinateur quantique*. Le Livre de poche. Librairie générale française, 2001.

[2] ACISSI. and M. Agé. *Sécurité informatique - Ethical Hacking : Apprendre l'attaque pour mieux se défendre*. Epsilon (Saint-Herblain). Editions ENI, 2009.