# NF05 - Introduction au langage C

## Rapport de projet : Cryptographie

**Aurélien DUREUX et Aya ABDALA**

Automne 2020

utt
UNIVERSITÉ DE TECHNOLOGIE
TROYES

# Table des matières

Au cours de notre troisième semestre à l'Université de Technologie de Troyes, nous avons suivi l'Unité d'Enseignement (UE) NF05, « Introduction au langage C ». Dans le cadre de cette UE, nous avons eu l'opportunité de réaliser un projet dans ce langage. Parmi les sujets proposés, nous avons opté pour le sujet portant sur la cryptographie.

La cryptographie est le procédé permettant de chiffrer un message, c'est-à-dire le rendre « secret » afin que seuls les destinataires puissent le lire. Pour pouvoir crypter et décrypter le message, il faut avoir une « clef de chiffrement ». Notre programme applique une méthode de cryptage dite symétrique, cela signifie que la clef permet à la fois de crypter et de décrypter.

La méthode de cryptage que nous avons codé est une méthode itérative. Lors de l'exécution, il y aura, en effet, une répétition des étapes de cryptage N fois, (N étant choisi par l'utilisateur souhaitant crypter). Il y a en tout 5 étapes de cryptage correspondant chacune à une procédure dans notre programme.

Nous allons premièrement détailler le fonctionnement des différentes procédures et de l'algorithme principal de notre programme.

Ensuite, vous retrouverez une partie évoquant les problèmes que nous avons rencontrés lors de l'élaboration de ce programme, ainsi que les solutions que nous avons adoptées en conséquence.

Suite à cela, vous trouverez un mode d'emploi utilisateur afin de mieux comprendre comment vous servir de ce programme.

Enfin, dans la conclusion de ce projet, vous trouverez des perspectives d'évolutions et d'améliorations au programme.

L'intégralité du programme commenté pourra être retrouvé en annexe.

Au fil du document, ***si vous trouvez du texte avec cette police***, c'est que c'est une lien hypertexte vous renvoyant vers la partie mentionnée par ce texte. Cela signifie que vous pouvez cliquer dessus.

# 1 Fonctionnement général

Cette partie vous explique le fonctionnement de notre programme, notamment en détaillant les divers choix pour lesquels nous avons optés face au sujet permettant une certaine liberté.

Comme expliqué en introduction, notre programme est constitué d'un algorithme principal appelant les différentes étapes de cryptage ou décryptage de façon itérative. Il y a en tout 5 étapes, vous trouverez la description du fonctionnement de chacune d'entre elle pour la partie cryptage et celle de décryptage.

Pour l'étape 4, il y aura une fonction supplémentaire permettant de générer la sous-clé nécessaire aux calculs de cette étape. L'étape 2, bénéficie également d'une particularité puisque c'est une seule et unique fonction qui permet à la fois de crypter et décrypter (vous retrouverez l'explication de ce choix dans le sous-chapitre dédié à cette étape : ***1.3***Fonction étape 2)

En plus de cela, il y a deux fonctions afin de modifier l'état des octets : l'une permettant de passer de la valeur décimale à la valeur binaire et l'autre permettant l'inverse.

Enfin, une dernière fonction permet de faire une saisie dynamique des différentes chaines de caractères nécessaires pour faire fonctionner le programme (notamment la clé, le nom complet du fichier en entrée et éventuellement celui en sortie si l'utilisateur décide de le saisir de lui-même).

Pour résumer, dans ce programme il y a :

- La fonction principale : le main (***1.1***)
- 2 procédures (cryptage et décryptage) pour chaque étape (de ***1.2*** à ***1.6***) - sauf la 2
- 1 procédure pour l'étape 2 (***1.3***)
- 1 procédure pour passer du binaire au décimal (***1.7.1***)
- 1 procédure pour passer du décimal au binaire (***1.7.2***)
- Une procédure générant des sous-clé pour l'étape 4 (***1.5.1***)
- Une procédure pour la saisie dynamique de chaine de caractère (***1.8***)

## 1.1 Fonction principale - main

Nous avons établi que c'est dans la partie principale du programme, le main, que toutes les informations importantes au cryptage et décryptage devraient être saisies. Ainsi, nous y retrouvons donc la saisie :

- Du choix de l'opération à effectuer (cryptage ou décryptage)
- De la clé de chiffrement
- Du nombre d'itération
- Du nom complet du fichier en entrée (chemin d'accès, nom et extension)
- Eventuellement du nom complet du fichier en sortie selon le choix de l'utilisateur

En effet, si l'utilisateur souhaite saisir le nom du fichier en sortie, il peut directement le faire. Si ce n'est pas le cas, le fichier en sortie sera nommé à partir de celui en entrée tout en rajoutant juste *« _crypted »* ou *« _decrypted »* avant l'extension en fonction de l'opération effectuée.

**Exemple :** Un fichier en entrée nommé *« doc\test.txt »* sera nommé *« doc\test_crypted.txt »* en sortie de cryptage et *« doc\test_decrypted.txt »* en sortie de décryptage.

Les différentes saisies de chaines de caractères se font toutes grâce à la fonction explicitée dans le chapitre suivant : ***Saisie dynamique chaine de caractères.***

Après cette saisie, avant de commencer les opérations, nous ouvrons les fichiers. S'il y a un problème, le programme renverra une erreur et il faudra relancer le programme en vérifiant soigneusement les noms des fichiers et chemin d'accès.

Ensuite c'est le début soit du cryptage, soit du décryptage en fonction de l'opération choisie. Dans les deux cas, les étapes de cryptage ou décryptage seront effectuées par paquet de 4 octets et de façon itérative (c'est-à-dire en se répétant).

### 1.1.1 Cryptage principal

Le principe général est de lire 4 octets du fichier en clair, les crypter en effectuant la répétition des étapes autant de fois que spécifié par l'utilisateur puis de les recopier dans le fichier crypté en sortie. Ce procédé se répète jusqu'à le fin du fichier.

Les étapes de cryptage s'enchainent de l'étape 1 à l'étape 5 avec deux manipulation des formats des octets. Un premier passage en binaire entre les étapes 2 et 3 et un second passage en décimal entre les étapes 4 et 5. Pour en savoir plus sur ces passages, il faut consulter la partie : ***Manipulation format octets***.

Lors du cryptage, nous ne connaissons à l'avance ni le nombre d'octet du fichier, ni s'il possède un multiple fini de 4 octets. Pour cela, nous vérifions donc au fur et à mesure que l'octet lu ne corresponde pas un 'EOF' : soit la fin du fichier. Si c'est le cas, la boucle de cryptage s'arrêtera après le tour en question. Sinon, la boucle continuera normalement le cryptage.

Ensuite, nous vérifions la position de l'octet dans le paquet ; si c'est le premier alors nous pouvons ignorer le paquet et fermer les fichiers. Si cette condition n'est pas vérifiée, alors nous avons recours à des octets de bourrage afin d'avoir un multiple fini de 4 octets. Le nombre de ces octets de bourrage sera copié à la fin du fichier crypté pour qu'ils soient indiqué lors du décryptage.

### 1.1.2 Décryptage principal

Lors du décryptage, grâce au cryptage, nous sommes sûrs d'avoir un multiple fini de 4 octets avec en plus 2 octets correspondant au nombre d'octet de bourrage et au 'EOF' (fin de fichier). C'est pourquoi, cette fois le principe sera simplement de lire les octets du fichier en entrée (étant crypté), de les décrypter et les recopier en dans le fichier en sortie.

Comme nous sommes dans le cas d'un cryptage symétrique, les étapes de décryptage s'enchaînent de l'étape 5 à 1 avec encore deux manipulations de formats des octets. Un premier passage en binaire entre les étapes 5 et 4 et un second passage en décimal entre les étapes 3 et 2.

Il faudra simplement à la fin du décryptage de chaque paquet d'octets vérifier si le deuxième octet du paquet suivant correspond à la fin de fichier.

Si c'est le cas, la boucle de décryptage s'arrêtera après ce tour et nous collecterons le premier octet du paquet suivant correspondant au nombre d'octet de bourrage. Il ne restera plus qu'à copier les octets dans le fichier en sortie sans copier ceux correspondant au bourrage.

Si ce n'est pas le cas, le curseur reculera de 2 crans pour continuer le décryptage normalement.

## 1.2 Fonctions étape 1

Dans cette étape, nous utilisons la valeur décimale des octets.

Le sujet imposait la modification de chaque octet selon une table de permutation liée à la clé de chiffrement. Pour cela, nous avons choisi d'utiliser une méthode similaire à la méthode de cryptage de Vigenère. Un écart sera calculé à partir des valeurs ASCII des caractères de la clé. Ainsi cela permet une bijection comme spécifié dans le sujet : chaque élément de l'ensemble de départ aura une solution dans l'ensemble d'arrivée.

### 1.2.1 Cryptage (1)

Durant le cryptage, nous incrémentons de l'écart. Comme expliqué ci-dessus, nous sommons la valeur ASCII des termes de la clé et nous l'ajoutons à la valeur initiale de chaque octet du paquet pour pouvoir obtenir leur valeur crypté.

Attention, comme nous parlons d'octet, la valeur doit rester entre 0 et 255 donc il faut appliquer un modulo au résultat final.

### 1.2.2 Décryptage (1)

Durant le décryptage, nous décrémentons de l'écart. Comme pour le cryptage, nous sommons la valeur ASCII des termes de la clé et la soustrayons de la valeur initiale de chaque octet du paquet pour pouvoir obtenir leur valeur décrypté.

Comme pour le cryptage, il faudra que le résultat soit compris entre 0 et 255 et vérifier que c'est bien un modulo positif et non négatif (auquel cas il suffit d'ajouter 256 jusqu'à avoir la première valeur positive).

## 1.3 Fonction étape 2

Dans cette étape, nous utilisons la valeur décimale des octets.

L'étape 2 permet de transformer chaque octet en un autre selon une table de permutation rédigée dans un tableau. La permutation que nous avons choisie est une permutation simple : 0 devient 255, 1 devient 254, 2 devient 253 etc… jusqu'à 255 devient 0. Cette permutation peut être complexifiée pour un meilleur cryptage (voir partie : **Conclusion et améliorations possibles**).

Attention, pour que cette étape puisse fonctionner correctement, il faut que le fichier contenant les permutant soit dans le même dossier que le fichier contenant le programme. S'il y a un problème avec ce fichier, il y aura un affichage le stipulant clairement.

### 1.3.1 Cryptage (2)

Pour crypter, il faut d'abord recopier la table de permutation dans un tableau multidimensionnel de 256 lignes à 2 colonnes. Une fois l'entièreté du tableau recopié, il faut rechercher dans la première colonne, la « cellule » contenant la valeur identique à notre octet. Lorsqu'elle est trouvée, il suffit de remplacer l'octet par la valeur associée dans la seconde colonne.

Il suffit de réitérer cette recherche pour les 4 octets afin qu'ils soient tous cryptés.

### 1.3.2 Décryptage (2)

Le décryptage est identique au cryptage car lorsque x devient y, y devient x. Si la permutation est notée dans un sens, elle peut s'appliquer dans l'autre. Par soucis de « praticité », nous avons décidé de rédiger la table de permutation pour les 256 octets, ainsi il ne faut chercher la valeur de l'octet que dans la première colonne. Ainsi, il nous est donc possible de ne réaliser qu'une seule procédure permettant à la fois de crypter et de décrypter.

## 1.4 Fonctions étape 3

Cette étape utilisant la valeur binaire des octets (d'où le passage en binaire dans l'algorithme principal juste avant cette étape).

Cette étape est purement calculatoire, nous y appliquons les calculs matriciels avec les constantes imposées dans le sujet.

### 1.4.1 Cryptage (3)

Pour le cryptage, nous appliquons le calcul matriciel avec les constantes imposées dans le sujet. Le calcul est $HV + c$ où V représente l'octet et H et c les constantes suivantes :

$$\mathbf{H} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \quad \text{et} \quad \mathbf{c} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

Nous effectuons d'abord le produit matriciel de H par V. Pour cela, nous appliquons la méthode générale de ce calcul (cf. **MATH03 - Algèbre linéaire**) qui est illustrée dans l'image ci-dessous.

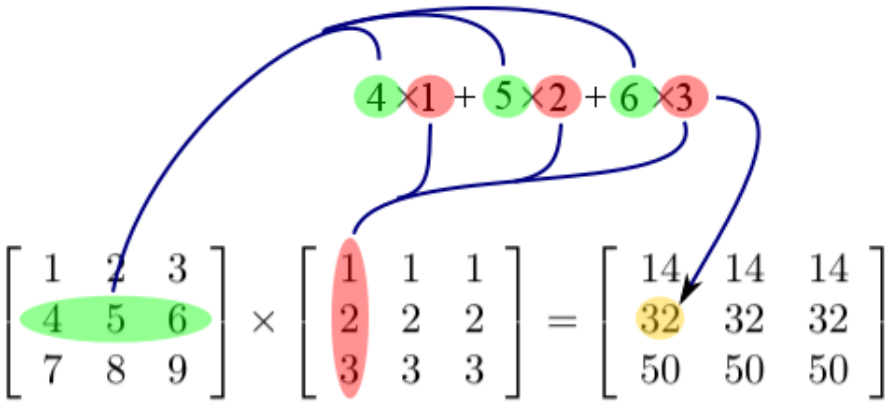



**Figure 1**
**Illustration produit matriciel**

Une fois ce produit matriciel effectué, nous obtenons une matrice de 8*1 (soit 8 lignes, 1 colonnes). Il suffit juste de sommer les termes des lignes correspondantes entre cette matrice et la constante c.

Tous les calculs sont effectués avec un modulo 2 comme un bit est en base 2 donc ne peut prendre comme valeur que 0 ou 1.

### 1.4.2 Décryptage (3)

Comme pour le cryptage, nous appliquons le calcul matriciel avec les constantes imposées dans le sujet. Le calcul est cette fois-ci $H'V + c'$ où V représente l'octet et H' et c' les constantes suivantes :

$$\mathbf{H'} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \quad \text{et} \quad \mathbf{c'} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Pour le décryptage également, nous commençons par le produit matriciel illustré dans la partie cryptage (**Figure 1**).

Comme pour le cryptage, nous effectuons les calculs avec un modulo 2 car nous n'utilisons que des bits ayant comme valeur 0 ou 1.

## 1.5 Fonctions étape 4

Cette étape s'effectue comme l'étape 3 en binaire.

Que ce soit en cryptage ou en décryptage, la fonction effectuera d'abord la génération d'une sous-clé qui permettra d'effectuer les calculs. Cette sous-clé est générée à partir de la clé d'itération.

### 1.5.1 Génération sous-clés

Pour générer la sous-clés, la fonction a besoin de l'étape de l'itération et du nombre d'itération. La sous-clé correspond à la somme des valeurs ASCII de la clé de chiffrement divisée par l'étape d'itération.

Si nous cryptons, la division se fait par une variable allant de 0 au nombre total d'itération. Si nous décryptons, cela sera l'inverse, donc du nombre total d'itération à 0.

Ce dénominateur en question peut être calculé durant le cryptage comme le décryptage grâce au nombre total d'itérations et à l'étape de l'itération.

Le résultat obtenu est un résultat décimal que nous transformons en binaire et dont nous ne gardons que les 8 premiers bits.

### 1.5.2 Cryptage (4)

Pour le cryptage, nous effectuons (comme imposé dans le sujet) un calcul XOR bit par bit entre la valeur initiale de l'octet et la sous-clé. Le calcul XOR, opérateur combinatoire, correspond à un « OU exclusif ». Cela qui signifie que si les 2 bits (de la sous-clé et de l'octet) sont identiques le bit résultat est 0 ; si au contraire ils sont différents alors le bit résultat est 1.

### 1.5.3 Décryptage (4)

Pour le décryptage, il faut effectuer la fonction inverse de XOR : c'est elle-même ! Nous effectuons donc comme lors du cryptage un XOR bit par bit entre la valeur de l'octet et la sous-clé.

## 1.6 Fonctions étape 5

Cette étape utilise les valeurs décimales des octets.

Il s'agit ici de calculer les valeurs des octets en fonction des valeurs précédentes selon un système imposé dans le sujet.

### 1.6.1 Cryptage (5)

Lors du cryptage, nous appliquons le système d'équation imposé dans le sujet. Ce système d'équation utilise les valeurs trouvées dans l'étape 4 pour donner les valeurs finales des octets qui correspondront donc aux octets cryptés finaux.

Comme nous utilisons les anciennes valeurs des octets, nous en effectuons une copie pour effectuer les calculs sans perdre les valeurs au fur et à mesure de l'exécution du système.

$$\begin{cases} Z_0 = Y_0 + Y_1 \\ Z_1 = Y_0 + Y_1 + Y_2 \\ Z_2 = Y_1 + Y_2 + Y_3 \\ Z_3 = Y_2 + Y_3 \end{cases}$$

Où $Z_0, Z_1, Z_2, Z_3$ sont les nouvelles valeurs des octets (octets cryptés) et $Y_0, Y_1, Y_2, Y_3$ sont les anciennes valeurs dont nous avons effectué une copie.

### 1.6.2 Décryptage (5)

Pour le décryptage, nous appliquons donc l'inversion du système précédent pour retrouver les valeurs originelles. De même que pour le cryptage, nous effectuerons une copie des valeurs pour pouvoir effectuer les calculs sans les perdre.

Nous avons fait l'inversion et nous retrouvons donc le système équivalent suivant :

$$\begin{cases} Y_0 = Z_0 - Z_2 + Z_3 \\ Y_1 = Z_2 - Z_3 \\ Y_2 = Z_1 - Z_0 \\ Y_3 = Z_3 - Z_1 + Z_0 \end{cases}$$

Où $Y_0, Y_1, Y_2, Y_3$ sont les nouvelles valeurs des octets (octets décryptés) et $Z_0, Z_1, Z_2, Z_3$ sont les anciennes valeurs dont nous avons effectué une copie

## 1.7 Manipulation format octets

### 1.7.1 Binaire vers décimal

Cette procédure permet de passer de la valeur binaire à la valeur décimale des octets, c'est-à-dire passer d'une valeur en base 2 à une valeur en base 10.

Pour cela, nous initialisons les valeurs de tous les octets à 0, puis nous sommons les valeurs de chaque bit multiplié par leur poids (puissance de 2).

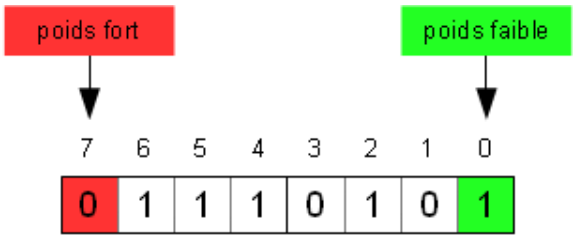



**Figure 2 :**
**Illustration poids d'un bit**

Par exemple pour l'image ci-dessus :

$$octet = 0 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 + 1 \times 2^4 + 1 \times 2^5 + 1 \times 2^6 + 0 \times 2^7$$

### 1.7.2 Décimal vers binaire

Cette procédure permet l'inverse de la précédente, elle permet de passer de la valeur décimale à la valeur binaire des octets, c'est-à-dire passer d'une valeur en base 2 à une valeur en base 10.

Pour cela, nous effectuons une division euclidienne par 2, le reste de chaque division corresponds à chaque bit allant du poids le plus fort vers le plus faible (autrement dit écrits de gauche à droite). La méthode est celle illustrée dans l'image ci-contre :

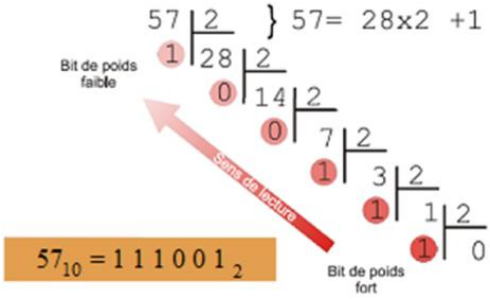



**Figure 3 :**
**Illustration passage décimal à binaire d'un bit**

## 1.8 Saisie dynamique chaine de caractères

Cette fonction permet de retourner une chaine de caractères allouée dynamiquement.

En effet, elle commence par allouer seulement 2 caractères à la chaine (le premier caractère lui-même et le caractère '\0' signifiant la fin d'une chaine de caractères). Pour enregistrer le premier caractère, la fonction ignore tous les sauts de lignes liées au saisies précédentes.

Ensuite, elle lit la chaine petit à petit. Tant que celle-ci n'est pas arrivée à sa fin, - soit un saut de ligne - la taille de la chaine est réalloué en étant incrémentée pour enregistrer le nouveau caractère lu. Lorsque la chaine en est à sa fin, la fonction remplace le saut de ligne par le bon caractère de fin d'une chaine de caractères.

# 2 Problème rencontrés et solutions

Durant le processus de codage du programme, nous avons parfois rencontrés des problèmes qui perturbaient le bon fonctionnement du cryptage ou du décryptage. Nous avons également eu des problèmes liés à la gestion du projet. Cette partie va vous présenter ces problèmes et les solutions apportées en conséquence.

Un des problèmes rencontrés a été la volonté de prendre en main le projet trop tôt. En effet, nous avons très vite fait le choix du sujet que nous allions traiter étant donné que la cryptographie nous intéressait tout deux fortement. Après une première analyse, nous avons commencé à essayer de développer les différentes étapes. Cependant le sujet donnant une trop grande liberté de choix, nous nous sommes parfois sentis perdus sans savoir vers quoi s'orienter. Parfois, nous avons cherché de suite à complexifier certaines étapes (par exemple : pour l'étape 1 effectuer une première approche avec une permutation par transposition rectangulaire des paquets d'octets, pour tester les fonction l'essai d'un programme convertissant un message en plusieurs paquets d'octets). Ces complexifications ont parfois pris beaucoup de temps, ont parfois faits apparaître des problèmes (par exemple pour l'étape 1 évoquée précédemment, des soucis de bijection). Ces contre-temps ont surtout poussé à une démotivation de notre part, d'autant plus face à la modification du sujet dont nous n'avions pas eu notification. Ce problème a surtout été résolu par le fait de travailler en groupe, nous avons dû nous soutenir mutuellement et redéfinir les tâches de chacun pour pouvoir mener le projet à bien. Une seconde solution a été d'être moins ambitieux et d'avoir comme objectif principal d'avoir une première version qui marche même simple puis ensuite, nous verrons afin de l'améliorer.

Un autre problème rencontré est lié au confinement ; travailler sur un projet commun est davantage complexe lorsque l'on est à distance. Il est donc plus compliqué de s'aligner sur la façon de faire et de comprendre les attentes de l'autre binôme. Pour pallier cela, nous avons essayé de régulièrement organiser des réunions téléphoniques avec des éventuels partages d'écrans. A la fin de chacune d'entre elles, nous avons rédigé un compte-rendu permettant de résumer les points abordés.

Ce problème de distanciel a donc également porté des problèmes d'alignement. Le principal étant la déclaration des octets, que ce soit en valeur décimale ou en valeur binaire. Lors de la première version du projet, ceux-ci ont été placés en variable globale comme ce sont des variables amenées à être manipulées par toutes les fonctions. Cependant ce choix a porté des divergences au sein même de notre binôme, mais également avec l'avis des professeurs et des règles de l'état de l'art du code. Nous avons donc finalement décider de garder les variables globales jusqu'à avoir finalisé le projet pour ensuite voir si nous avions le temps de les passer en variables locales dans le main et sous forme de pointeurs. Ce dernier passage sous forme de pointeurs a été encouragé par la dernière consignes (Bonus du sujet) indiquant de passer en argument des pointeurs à chaque fonctions.

Un autre problème technique est que nous avons directement décidé d'adapter nos fonctions pour un fonctionnement sous plusieurs itérations (bonus du sujet). Lors de la conception de l'étape 4, il a donc fallu réfléchir à un principe permettant de générer une sous-clé d'itération différente pour chaque itération et pouvant être inversé au décryptage. Cette sous-clé devant être générée pseudo-

aléatoirement à partir de la clé, nous avons donc choisi de simplement sommer les valeurs ASCII de ses caractères et les diviser par le numéro de l'itération (pour en savoir plus : ***Génération sous-clés***).

Une partie assez complexe a été également la lecture du fichier par groupe de 4 octets. Il fallait lire le fichier jusqu' à ce qu'un paquet d'octets contienne la valeur de fin de fichier. Après plusieurs tentatives infructueuses, le choix s'est porté pour l'utilisation d'une condition d'arrêt détectant l'apparition d'une fin de fichier. Il a fallu isoler le cas particulier où le nouveau paquet d'octet serait vide donc inutile (i.e. le premier octet du paquet serait la fin de fichier). Pour ce faire l'utilisation d'une seconde variable booléenne permet donc d'activer ou non l'exclusion du paquet (qui n'arrivera que si la condition d'arrêt de lecture est vraie donc le cas particulier n'arrive bien qu'une seule fois dans la lecture du fichier).

Enfin, le dernier problème également lié à la lecture du fichier est que nous ne savons pas à l'avance s'il y a un multiple fini de 4 octets. Ainsi, pour pallier le problème, nous ajoutons des octets dits de bourrage initialisés à 0. Pour que le décryptage s'effectue sans soucis, il faut donc savoir combien il y a d'octets de bourrage. Pour cela, nous décidons de noter ce nombre à la toute fin du fichier crypté. Lors du décryptage, nous lirons toujours les 2 octets suivant du paquet d'octet décrypté afin de savoir si le 2$^{ème}$ correspond à la fin du fichier. Si c'est le cas, alors nous saurons le nombre d'octets de bourrage et ils ne seront pas recopiés. Si ce n'est pas le cas, le curseur revient en arrière de 2 crans et le décryptage continu. Pour plus d'informations concernant la lecture des fichiers en cryptage et décryptage vous pouvez lire la partie dédiée à cela : ***Fonction principale - main***.

# 3 Guide d'utilisation

Cette partie vous présente un guide pour bien utiliser le programme de cryptage et de décryptage, étape par étape. L'utilisation de ce programme est assez simple et intuitive mais dans le doute, vous pourrez-vous référé à ce guide.

Premièrement, voici une capture d'écran de toutes les étapes, vous pouvez-vous y référer au fur et à mesure de la lecture du guide pour mieux comprendre :

```
Quelle operation souhaitez vous effectuer :
Crypter [0] ou Decrypter [1]
Choix operation :
  >> 0


Veuillez saisir le nombre d'iteration, attention, il ne peut pas etre nul :
  >> 5


Veuillez saisir votre cle, elle ne doit pas contenir d'espaces :
  >> KEYNF05


Veuillez saisir le chemin d'acces complet du fichier \x\x\x.ext :
  >> C:\Users\Aya\Desktop\UTT\TC03\NF05\projet\test.txt


Souhaitez-vous choisir le chemin d'acces complet du fichier en sortie
 Non [0] ou Oui [1]
  >>0
Fichier converti : 0 %
Fichier converti : 20 %
Fichier converti : 40 %
Fichier converti : 60 %
Fichier converti : 80 %
Fichier converti : 100 %


Temps d'execution de 0 heures, 0 minutes, 1 secondes

L'operation s'est parfaitement bien passee
```

1- La première chose que va vous demander le programme, c'est de saisir l'opération souhaitée. Il faudra taper 0 si vous souhaitez crypter votre fichier et 1 si vous souhaitez le décrypter.
Dans l'exemple, le choix est « 0 » donc c'est bien un cryptage qui sera effectué

2- Ensuite, il vous faudra entrer le nombre d'itérations souhaité. Il doit être positif différent de 0 :

- Si vous cryptez, c'est à vous de le choisir. Plus il sera grand, plus il sera difficile de déchiffrer le fichier sans le programme (votre fichier sera mieux protégé). N'oubliez pas de bien communiquer cette donnée à la personne qui décryptera le fichier.
- Si vous décryptez, il faudra saisir le même nombre d'itération que la personne ayant crypté le fichier. Si elle ne vous l'a pas communiqué, vous ne pourrez pas décrypter le fichier.

Dans l'exemple, le nombre est « 5 » donc les étapes de cryptage seront répétées 5 fois.

3- Après cela, il vous faudra entrer la clé de chiffrement.

- Si vous cryptez, c'est à vous de la choisir. Une fois encore, n'oubliez pas de la communiquer à la personne qui va décrypter le fichier ensuite.

- Si vous décryptez, il faudra saisir la même que la personne ayant crypté le fichier. Si elle ne vous l'a pas communiqué, encore une fois il vous sera impossible de décrypter le fichier.

Dans l'exemple, la clé est « KEYNF05 », grâce à elle certaine étape du cryptage seront effectuées.

4- Il faudra ensuite entrez le nom complet du fichier en entrée (chemin d'accès\nom.extension). Vous pouvez crypter n'importe quel type de fichier avec ce programme (docx, pdf, rtf, txt, etc…).
Pour connaître le chemin d'accès, vous pouvez le copier-coller depuis :

- L'explorateur de fichier de votre ordinateur
- Les propriétés du fichier que vous voulez crypter ou décrypter.

Vous pouvez également indiquer le chemin d'accès à partir du dossier contenant le programme en utilisant '..\' pour reculer de dossier. S'il est dans le même dossier nom.extension suffit.
Attention, cette étape est très importante pour ne pas avoir de problème lors de l'ouverture des fichiers. Il ne faut pas se tromper
Dans l'exemple, le nom complet du fichier est « C:\Users\Aya\Desktop\UTT\TC03\NF05\test.txt ».

5- Le programme vous demande après si vous souhaitez choisir le nom du fichier en sortie. Tapez 1 si vous souhaitez le nommer et 0 si vous ne le souhaitez pas.

- Si vous avez choisi de le nommer vous-même, il faudra entrez le nom complet (chemin d'accès\nom.extension) que vous souhaitez pour ce fichier. L'extension doit être la même que le fichier pour pouvoir le rouvrir après le décryptage.
- Si vous ne voulez pas renommer le fichier, c'est le programme qui se chargera de le nommer à partir du fichier en entrée.

Dans l'exemple, le choix est « 0 » donc le programme renommera le fichier en sortie « C:\Users\Aya\Desktop\UTT\TC03\NF05\test_crypted.txt ».

Le choix aurait pu être « 1 » en nommant le fichier « C:\Users\Aya\Desktop\cryptage.txt ».

6- Voilà, vous n'avez plus qu'à attendre et votre fichier sera prêt !

- Que ce soit en cryptage ou en décryptage, un affichage vous tiendra informer du bon déroulement de l'opération.
  Vous aurez également l'évolution de cryptage de 20% en 20%.
  A la fin, un affichage vous permettra également de voir en combien de temps ont eu lieu les opérations.
- Si vous n'avez pas cet affichage, c'est qu'il y a eu une erreur. En général l'erreur sera affichée, relancez donc votre programme en faisant attention à celle-ci. Elle est le plus souvent liée à :
  - Une erreur dans le nom du fichier en entrée
  - Une mauvaise position du fichier permut_etap_2.txt (ce fichier doit vous être fourni par la personne ayant crypté et il doit être rangé dans le même dossier que votre programme).

Attention : les informations ne doivent jamais être communiquées à la personne à qui vous souhaitez les envoyer dans le même message ou email. Sinon il serait trop facile pour une tierce personne d'intercepter toutes les données dont elle a besoin pour décrypter votre fichier sans votre accord.

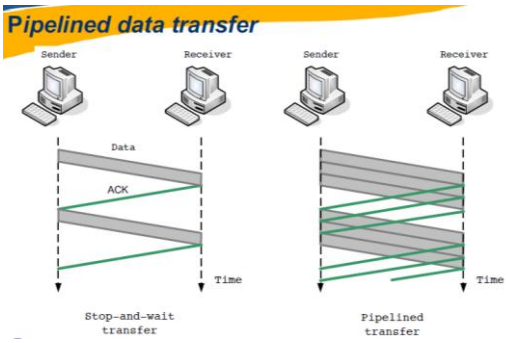
# 4 Conclusion et améliorations possibles

Pour conclure, ce projet a été très enrichissant et intéressant. Nous avons fortement apprécié améliorer nos compétences en langage C avec un programme aussi concret et utile. Cela a été un réel plaisir pour nous de le rendre complet comme s'il était destiné à une réelle utilisation par des clients.

Bien que complet, nous sommes conscients que ce projet peut être amélioré pour une bien meilleure expérience utilisateur et un meilleur cryptage à l'ère où la sécurité de nos fichiers et données sensibles deviens un réel enjeu. C'est pourquoi, nous exposons dans la suite de cette conclusion toutes nos idées d'améliorations pour ce projet.

1. Premièrement, l'amélioration notable à laquelle nous avons pensé suite aux différents essais est une interface utilisateur plus ergonomique. En effet, au lieu d'avoir à taper 0 ou 1 pour effectuer ses choix, l'utilisateur n'aurait juste à cliquer sur un bouton « OUI » ou « NON ».
2. Également que l'utilisateur puisse directement uploader ces fichiers comme nous le faisons par exemple pour une pièce-jointe dans un email, serait en faveur d'une meilleure expérience pour l'utilisateur. Cela lui permettrait d'éviter les erreurs dans la copie du chemin d'accès.
3. Une autre amélioration pour améliorer l'ergonomie serait que pour les utilisateurs souhaitant simplement crypter un message puisse directement le faire dans la console avec en sortie un fichier texte (.txt).
4. Une dernière amélioration pour l'ergonomie à laquelle nous avons pensé est la génération d'un fichier annexe avec le fichier en sortie contenant la clé et le nombre d'itération pour que la personne ayant crypté ne l'oublie pas.
5. Une amélioration technique à laquelle nous avons pensé concerne l'étape 2. Générer directement le fichier de permutation lors du cryptage plutôt que d'avoir un fichier « écris à la main ». Cela permettrait de diversifier les combinaisons possibles et multiplier la diversités des cryptages, qui seraient alors plus difficile à déchiffrer pour une personne tierce. Bien évidemment, il faudra faire attention à bien avoir une bijection de [0 ; 255] à [0 ; 255].
6. Lors des différents tests de notre programme, nous avons remarqué certaines limites des performances. En effet, un fichier de taille assez conséquente prendra beaucoup de temps à être crypté ou décrypté (par exemple : 1 fichier de 2Mo a été crypté en 50 min). Pour contrer cela, nous avons pensé à une éventuelle amélioration en adoptant l'approche du « *pipeline data transfer* » (cf. ***NF02A***). Cela permettrait d'envoyer plusieurs paquet d'octets pour qu'ils soient traités en même temps comme ils ne

dépendent pas les uns des autres. Cela permet de diviser le temps d'exécution et donc de crypter ou décrypter plus rapidement. L'image ci-contre (issue du diaporama de NF02A permet d'illustrer ce procédé appliqué dans l'architecture des réseaux).




7. Enfin dans ce dernier point, nous souhaitions simplement évoquer les améliorations que nous avons effectuées durant le projet qui elles aussi apportent un plus à ce projet :
    - Nous avons décidé premièrement d'afficher l'évolution de l'opération de cryptage ou décryptage. Cela permet en effet de savoir où en est l'algorithme et ne pas attendre dans le vide lorsque le fichier en entrée est assez
    - Pour accompagner cette donnée, nous avons décidé d'afficher le temps pris par l'algorithme pour crypter ou décrypter afin de donner une idée d'ordre.

# 5 Annexe

## 1.1 Annexe 1 : Documentation Doxygen

La documentation de ce projet a été effectuée grâce à Doxygen. Celui-ci fournit un fichier HTML et un pdf. Ces documents sont dans le dossier Documentation.

Le fichier HTML Doxygen est présent dans le sous-dossier html. Vous pouvez l'ouvrir en cliquant sur ce lien : documentation/html/index.html. Si cela ne marche pas, vous pouvez l'ouvrir en allant directement dans le dossier et en ouvrant le fichier index.html.

Le fichier pdf se nomme Documentation_Doxygen_Cryptographie.pdf et se trouve directement dans le dossier. Vous pouvez l'afficher en cliquant sur ce lien : documentation/Documentation_Doxygen_Cryptographie.pdf

## 1.2 Annexe 2 : Code complet commenté

Voici le code entièrement commenté :

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <time.h>

//Definition d'operation
#define CRYPT     0
#define DECRYPT   1
#define FALSE     0
#define TRUE      1

//Nombre d'octet et de bits par octet (peut être facilement modifiable)
#define NB_OCTET  4
#define NB_BIT    8


/**
  Passage de valeurs decimales a valeurs binaires
*/
void pass_binary(int *oct, int **bits_oct)
{
  // Initialisation de tous les bits des octets a 0
  for (int i1 = 0; i1 < NB_OCTET; i1++) {
    for (int i2 = 0; i2 < NB_BIT; i2++)
        bits_oct[i1][i2] = 0;
  }

  // Calcul valeur binaire bits par bits
  for (int i1 = 0; i1 < NB_OCTET; i1++) {
    int octet = oct[i1]; // pour ne pas modifier la valeur originale de l'octet
    for (int i2 = NB_BIT - 1; octet > 0 || i2 >= 0; i2--) {
      bits_oct[i1][i2] = octet % 2;
      octet = octet / 2;
    }
  }
}
/**
 Passage de valeurs binaires a valeurs decimales
*/
```

```
41. void pass_decimal(int **bits_oct, int *oct)
42. {
43.   // Initialisation de tous les octets a 0
44.   for (int i1 = 0; i1 < NB_OCTET; i1++)
45.     oct[i1] = 0;
46. 
47.   // Calcul valeur decimale
48.   for (int i1 = 0; i1 < NB_OCTET; i1++){
49.     for (int i2 = 0; i2 < NB_BIT; i2++){
50.       oct[i1] += bits_oct[i1][7 - i2] * (int)pow(2, i2);
51.       oct[i1] = oct[i1] % 256;
52.       while (oct[i1] < 0)
53.         oct[i1] = oct[i1] + 256;
54.     }
55.   }
56. }
57. 
58. 
59. 
60. /**
61.  Etape de cryptage 1 - Associe à chaque valeur une autre par ajout valeurs ASCII clé
62. */
63. void etape_1_crypt(int *oct, const char *key)
64. {
65.   int sum = 0, indx = 0;
66. 
67.   // Calcul écart en sommant valeurs ASCII clé de chiffrement
68.   while (key[indx] != '\0'){
69.     sum += (int)key[indx];
70.     indx++;
71.   }
72. 
73.   // Incremente les valeurs de chaque octet de l'écart calculé
74.   for (int i = 0; i < NB_OCTET; i++)
75.     oct[i] = (oct[i] + sum) % 256;
76. }
77. /**
78. Etape de décryptage 1 Associe à chaque valeur une autre par retrait valeurs ASCII clé
79. */
80. void etape_1_decrypt(int *oct, const char *key)
81. {
82.   int sum = 0, indx = 0;
83. 
84.   // Calcul écart en sommant valeurs ASCII clé de chiffrement
85.   while (key[indx] != '\0'){
86.     sum += (int)key[indx];
87.     indx++;
88.   }
89. 
90.   // Décremente les valeurs de chaque octet de l'écart calculé
91.   for (int i = 0; i < NB_OCTET; i++) {
92.     oct[i] = (oct[i] - sum) % 256;
93. 
94.     // pour toujours avoir un resultat positif
95.     while (oct[i] < 0)
96.       oct[i] = oct[i] + 256;
97.   }
98. }
99. 
100. 
101. 
102.       /**
103.        Etape de cryptage et décryptage 2 - Associe à chaque x un y grâce au fichier permu
    t_etape_2.txt
104.       */
105.       void etape_2_crypt_decrypt(int *oct)
```

```
106.      {
107.        // Declaration tableau de permutation et pointeur fichier
108.        int tab_permut[256][2];
109.        FILE *file_permut = NULL;
110.
111.       // Ouverture fichier avec table permutation doit ê dans même fichier que le code
112.        file_permut = fopen("permut_etap_2.txt", "r");
113.        // Vérification bonne ouverture du fichier et affichage en cas d'erreur
114.        if (file_permut == NULL) {
115.          printf("Erreur ouverture fichier permutation\n");
116.          exit(EXIT_FAILURE);
117.        }
118.        else {
119.          // Boucle de copie de la table de permutation
120.          for(int i1 = 0; i1 < 256; i1++)
121.            for (int i2 = 0; i2 < 2; i2++)
122.              fscanf(file_permut, "%d", &tab_permut[i1][i2]);
123.
124.          //Recherche valeur dans table permutation puis permutation
125.          for(int i = 0; i < NB_OCTET; i++)
126.            for (int indx = 0; indx < 256; indx++)
127.              if(oct[i] == tab_permut[indx][0]) {
128.                oct[i] = tab_permut[indx][1];
129.                break;
130.              }
131.
132.          //Fermeture du fichier de permutation
133.          fclose(file_permut);
134.        }
135.      }
136.
137.
138.
139.      /**
140.       Etape de cryptage 3 - Calcul matriciel selon constantes imposées dans le sujet
141.      */
142.      void etape_3_crypt (int **bits_oct)
143.      {
144.        //Initialisation bits intermediaires de calcul
145.        int bits_inter[NB_OCTET][NB_BIT];
146.        for (int i = 0; i < NB_OCTET; i++)
147.          for (int j = 0; j < NB_BIT; j++)
148.            bits_inter[i][j] = bits_oct[i][j];
149.
150.
151.        //Initialisation des constantes
152.        int H[8][8] = {{1, 0, 0, 0, 1, 1, 1, 1},
153.                       {1, 1, 0, 0, 0, 1, 1, 1},
154.                       {1, 1, 1, 0, 0, 0, 1, 1},
155.                       {1, 1, 1, 1, 0, 0, 0, 1},
156.                       {1, 1, 1, 1, 1, 0, 0, 0},
157.                       {0, 1, 1, 1, 1, 1, 0, 0},
158.                       {0, 0, 1, 1, 1, 1, 1, 0},
159.                       {0, 0, 0, 1, 1, 1, 1, 1}};
160.        int c[8] = {1, 1, 0, 0, 0, 1 , 1, 0};
161.
162.        //[H*V] Boucle pour le produit matriciel avec modulo 2 (pas obligatoire, peut se
    faire seulement etape suivante)
163.        for (int i = 0; i < NB_OCTET; i++)
164.          for (int j = 0; j < NB_BIT; j++) {
165.            int res = 0;
166.            for (int k = 0; k < NB_BIT; k++)
167.                res += bits_inter[i][k] * H[j][k];
168.            bits_oct[i][j] = res % 2;
169.          }
170.
```

```
171.          //[H*V+c] Addition du résultat précédent avec c + application modulo 2
172.          for (int i = 0; i < NB_OCTET; i++)
173.            for (int j = 0; j < NB_BIT; j++)
174.              bits_oct[i][j] = (bits_oct[i][j] + c[j])%2;
175.
176.        }
177.
178.        /**
179.        Etape de décryptage 3 - Calcul matriciel selon constantes imposées dans le sujet
180.        */
181.        void etape_3_decrypt(int **bits_oct)
182.        {
183.          //Initialisation bits intermediaires de calcul
184.          int bits_inter[4][8];
185.          for (int i = 0; i < NB_OCTET; i++)
186.            for (int j = 0; j < NB_BIT; j++)
187.              bits_inter[i][j] = bits_oct[i][j];
188.
189.          //Initialisation des constantes
190.          int H[8][8] = {{0, 0, 1, 0, 0, 1, 0, 1},
191.                         {1, 0, 0, 1, 0, 0, 1, 0},
192.                         {0, 1, 0, 0, 1, 0, 0, 1},
193.                         {1, 0, 1, 0, 0, 1, 0, 0},
194.                         {0, 1, 0, 1, 0, 0, 1, 0},
195.                         {0, 0, 1, 0, 1, 0, 0, 1},
196.                         {1, 0, 0, 1, 0, 1, 0, 0},
197.                         {0, 1, 0, 0, 1, 0, 1, 0}};
198.          int c[8] = { 1, 0, 1, 0, 0, 0, 0, 0};
199.
200.          //[H'*V] Boucle pour le produit matriciel avec application modulo 2 (peut etre ef
    fectue seulement à l'étape suivante)
201.          for (int i = 0; i < NB_OCTET; i++)
202.            for (int j = 0; j < NB_BIT; j++) {
203.              int res = 0;
204.              for (int k = 0; k < NB_BIT; k++)
205.                  res += bits_inter[i][k] * H[j][k];
206.              bits_oct[i][j] = res %2;
207.            }
208.
209.          //[H'*V+c'] Addition du résultat précédent avec c' + application du modulo 2
210.          for (int i = 0; i < NB_OCTET; i++)
211.            for (int j = 0; j < NB_BIT; j++)
212.              bits_oct[i][j] = (bits_oct[i][j] + c[j])%2;
213.
214.        }
215.
216.
217.
218.        /**
219.        Generation sous_cles via clé de cryptage et etape d'itération pour étape 4 cryptage
220.        */
221.        void sous_cle(int mode, int nb_it_tot, int etape_it, char *key, int sub_key[])
222.        {
223.          int sum = 0, div;
224.
225.          // DETERMINATION SOUS CLE : somme valeurs caracteres cle chiffrement divisé en fo
    ntion etape d'iteration
226.          for (int i = 0; i < strlen(key); i++)
227.            sum += (int)key[i];
228.
229.          // DISTINCTION ENTRE CRYPTAGE ET DECRYPTAGE : crypt = 1 a nb d'iteration / decryt
    = nb d'iteration à 1
230.          if (mode == CRYPT) div = etape_it + 1;
231.          else div = nb_it_tot - etape_it;
232.
```

```
233.          // CALCUL FINAL VALEUR : somme de la clé / div : dépendant du mode de l'opération
234.          sum /= div;
235.
236.          // PASSAGE BINAIRE DE LA SOUS CLE : seulement les 8 premiers bits sont conservés
237.          int i = 0;
238.          while (sum > 0 && i < NB_BIT){
239.              sub_key[(NB_BIT - 1) - i] = sum % 2;
240.              sum = sum / 2;
241.              i++;
242.          }
243.        }
244.        /**
245.         Etape de cryptage 4 - Genere sous_clé via clé et étape pour faire XOR bit par bit
246.        */
247.        void etape_4_crypt(int **bits_oct, int nb_iteration, int etape_iteration, char *key
    )
248.        {
249.          int sub_key[8] = {0};
250.
251.          //Generation sous_cle
252.          sous_cle(CRYPT,  nb_iteration, etape_iteration, key, sub_key);
253.
254.          // Application calcul XOR bit par bit
255.          for (int i = 0; i < NB_OCTET; i++)
256.            for (int j = 0; j < NB_BIT; j++)
257.              bits_oct[i][j] ^= sub_key[j];
258.
259.        }
260.        /**
261.        Etape de décryptage 4 - Génère sous_clé via clé et étape pour faire XOR bit par bit
262.        */
263.        void etape_4_decrypt(int **bits_oct, int nb_iteration, int etape_iteration, char *k
    ey)
264.        {
265.          int sub_key[8] = {0};
266.
267.          // Generation sous_cle
268.          sous_cle(DECRYPT, nb_iteration, etape_iteration, key, sub_key);
269.
270.          // Application calcul XOR bit par bit
271.          for (int i = 0; i < NB_OCTET; i++)
272.            for(int j = 0; j < NB_BIT; j++)
273.              bits_oct[i][j] ^= sub_key[j];
274.
275.        }
276.
277.
278.
279.        /**
280.         Etape de cryptage 5 - Calcul système imposé par le sujet
281.        */
282.        void etape_5_crypt(int *oct)
283.        {
284.          int inter[NB_OCTET];
285.
286.          // Boucle copie anciennes valeurs
287.          for(int i = 0; i < NB_OCTET; i++)
288.              inter[i] = oct[i];
289.
290.          //Application système imposé dans le sujet
291.          oct[0] = (inter[0] + inter[1]) % 256;
292.          oct[1] = (inter[0] + inter[1] + inter[2]) % 256;
293.          oct[2] = (inter[1] + inter[2]+ inter[3]) % 256;
294.          oct[3] = (inter[2] + inter[3]) % 256;
```

```
295.
296.          // Boule verification valeures entre 0 et 255 car octet = 8 bit
297.          for(int i = 0; i < NB_OCTET; i++)
298.              while (oct[i] < 0)
299.                  oct[i] = oct[i] + 256;
300.        }
301.        /**
302.         Etape de décryptage 5 - Calcul inverse système etape 5 crypt imposée par le sujet
303.        */
304.        void etape_5_decrypt(int *oct)
305.        {
306.          int inter[NB_OCTET];
307.
308.          // Boucle copie anciennes valeurs
309.          for(int i = 0; i < NB_OCTET; i++)
310.              inter[i] = oct[i];
311.
312.          //Application système imposé dans le sujet
313.          oct[0] = (inter[0] - inter[2] + inter[3]) % 256;
314.          oct[1] = (inter[2] - inter[3]) % 256;
315.          oct[2] = (inter[1] - inter[0]) % 256;
316.          oct[3] = (inter[3] - inter[1] + inter[0]) % 256;
317.
318.          // Boule verification valeures entre 0 et 255 car octet = 8 bit
319.          for(int i = 0; i < NB_OCTET; i++)
320.              while (oct[i] < 0)
321.                  oct[i] = oct[i] + 256;
322.
323.        }
324.
325.
326.
327.        /**
328.        Saisie une chaîne dynamiquement sans connaître sa taille à l'avance
329.        */
330.        char *saisie_chaine_dynam()
331.        {
332.          char *chaine = NULL ;
333.          // Un caractere et un '\0'
334.          int n = 2;
335.
336.          //Allocation et lecture premier caractère
337.          chaine = malloc(n*sizeof(char));
338.          scanf("%c", &chaine[n-2]);
339.
340.          // Eviter les retours de lignes des autres saisies
341.          while (chaine[n-2] == '\n')
342.            scanf("%c", &chaine[n-2]);
343.
344.          //Boucle de reallocation dynamique tq pas fin de saisie
345.          while (chaine[n-2] != '\n') {
346.              n++; //on augmente la taille
347.              chaine = realloc(chaine, n * sizeof(char));
348.              scanf("%c", &chaine[n-2]);
349.          }
350.
351.          // Caractère de fin de chaine
352.          chaine[n-2] = '\0';
353.
354.          return chaine ;
355.        }
356.
357.
358.        /**
359.        Saisie informations necessaires et effecture lecture et écriture dans les fichiers
360.        */
```

```
361.       int main()
362.       {
363.         // Variables concernant les octets (forme décimale et forme binaire)
364.         int *octet, **bits_octet;
365.
366.         // Variables principales des paramètres cryptage/decryptage
367.         int choix_ops, nb_iteration, choix_name = FALSE;
368.         char *key;
369.
370.         // Variables noms des fichiers
371.         char *extension = NULL, *name_in = NULL, *name_out = NULL;
372.         int lenght_ext, j;
373.         char add_crypted[9] = {"_crypted"}, add_decrypted[11] = {"_decrypted"};
374.         // Variables fichier et leur contrôle
375.         FILE *file_in = NULL, *file_out = NULL;
376.         int arret = FALSE, ignore_oct = FALSE;
377.
378.         // Variables octets bourrage
379.         int nb_bourrage = 0, rech_bourrage[2];
380.
381.         // Variables pour affichage progression opération cryptage ou décryptage
382.         int tot_size, cur_size, percent = 20;
383.
384.         // Allocation dynamique des variables concernant les octets
385.         octet = (int*) malloc ( NB_OCTET * sizeof(int));
386.         bits_octet = (int **) malloc( NB_OCTET * sizeof(int *) );
387.
388.         for (int i = 0; i < NB_OCTET; i++)
389.           bits_octet[i] = (int *) malloc(NB_BIT * sizeof(int));
390.
391.
392.         // Saisie de l'operation souhaitee avec un controle bon choix
393.         do{
394.           printf ("Quelle operation souhaitez vous effectuer : \nCrypter [0] ou Decrypter
    [1]\nChoix operation :\n  >> ");
395.           scanf("%d", &choix_ops);
396.         } while (choix_ops != CRYPT && choix_ops != DECRYPT);
397.
398.         // Saisie nombre iteration avec contrôle bon choix (pas de résultat = 0)
399.         do {
400.           printf("\n\nVeuillez saisir le nombre d'iteration, attention, il ne peut pas et
   re nul :\n  >> ");
401.           scanf("%d", &nb_iteration);
402.         } while (nb_iteration == 0);
403.
404.         // Saisie clé de chiffrement (dynamique)
405.         printf("\n\nVeuillez saisir votre cle, elle ne doit pas contenir d\'espaces :\n
   >> ");
406.         key = saisie_chaine_dynam();
407.
408.         // Saisie nom chemin d'accès + nom + extension fichier dynamique
409.         do {
410.           printf("\n\nVeuillez saisir le chemin d'acces complet du fichier \\x\\x\\x.ext
   :\n  >> ");
411.           name_in = saisie_chaine_dynam ();
412.
413.           // Rercherche à partir d'où commence l'extension
414.           j = (int)strlen(name_in);
415.           while (j > 0 && name_in[j] != '.') j--;
416.           if (j == 0 || j == strlen(name_in))
417.             printf("Erreur de saisie!\n");
418.         } while(j == 0 || j == strlen(name_in));
419.
420.         // Copie de l'extension dans la variable qui lui est attitrée avec allocation dyn
   amique
421.         lenght_ext = (int)strlen(name_in) - j;
```

```
422.          extension = (char *) malloc ( lenght_ext * sizeof(char) );
423.          strncpy(extension, name_in + j, lenght_ext);
424.          extension[lenght_ext] = '\0';
425.
426.          // Nommage fichier de sortie
427.          printf("\n\nSouhaitez-
    vous choisir le chemin d\'acces complet du fichier en sortie\n Non [0] ou Oui [1]\n  >>");

428.          scanf("%d", &choix_name);
429.          if (choix_name == TRUE) {
430.            printf("Veuillez saisir le chemin et nom voulu pour le fichier en sortie :\n  >
    > ");
431.            name_out = saisie_chaine_dynam();
432.          }
433.          else {
434.            // Allocation dynamique nom fichier sortie avec taille chaine entrée + 12 car
435.            name_out = (char *) malloc ((strlen(name_in) + 12) * sizeof(char));
436.
437.            // Copie nom fichier en entrée dans nom fichier en sortie mais sans l'extension

438.            strncpy(name_out, name_in, strlen(name_in) - (lenght_ext));
439.
440.            // Ajout chaine de caractère en fonction de l'opération effectuée pour différen
    cier name_in de name_out
441.            if (choix_ops == CRYPT)
442.                strcat(name_out, add_crypted);
443.            else strcat(name_out, add_decrypted);
444.
445.            // Ajout extension
446.            strcat(name_out, extension);
447.          }
448.
449.            //Début calcul temps
450.            clock_t debut = clock();
451.
452.          // Ouverture des fichiers en binaire
453.          file_in = fopen(name_in, "rb");
454.          file_out = fopen(name_out, "wb");
455.
456.
457.          // Verification ouverture des fichiers
458.          if (file_in == NULL){
459.              printf("Erreur d\'ouverture du fichier entrant\n");
460.              exit(EXIT_FAILURE);}
461.          else if (file_out == NULL){
462.              printf("Erreur d\'ouverture du fichier sortant\n");
463.              exit(EXIT_FAILURE);
464.          }
465.
466.          fseek(file_in, 0L, SEEK_END);
467.          tot_size = ftell(file_in);
468.          fseek(file_in, 0L, SEEK_SET);
469.          printf("Fichier converti : 0 %%\n");
470.
471.          // CRYPTAGE : lecture et écriture fichier jusqu'à sa fin (4 par 4 octets)
472.          if (choix_ops == CRYPT) {
473.            while (arret == FALSE) {
474.
475.              //Affichage progression cryptage en pourcentage
476.              cur_size = ftell(file_out);
477.              if ((cur_size * 100 / tot_size) / percent == 1 ){
478.                printf("Fichier converti : %d %%\n", cur_size * 100 / tot_size);
479.                percent += 20;
480.              }
481.
482.              for (int i = 0; i < NB_OCTET; i++) {
```

```
483.               //Lecture d'un octet
484.               octet[i] = fgetc(file_in);
485.
486.               // Condition pour savoir si fin fichier
487.               if (octet[i] == EOF) {
488.                 arret = TRUE;
489.                 //Condition pour savoir si on est au premier octet d'un paquet
490.                 if (i == 0) {
491.                   // Condition vérifiée :ignore paquet octet
492.                   ignore_oct = TRUE;
493.                   break;
494.                 }
495.                 // Condition non vérifiée : octets de bourrage
496.                 octet[i] = 0;
497.                 // Variable pour enregistrer le nombre d'octet de bourrage
498.                 if (nb_bourrage == 0 )
499.                   nb_bourrage = NB_OCTET - i;
500.               }
501.             }
502.
503.             // Condition pour vérifier si le paquet est ignoré
504.             if (ignore_oct != TRUE) {
505.
506.               //CRYPTAGE ITERATIF avec contrôle des passages binaire/decimal
507.               for (int etape = 0; etape < nb_iteration; etape++){
508.                 etape_1_crypt(octet, key);
509.                 etape_2_crypt_decrypt(octet);
510.                 pass_binary(octet, bits_octet);
511.                 etape_3_crypt(bits_octet);
512.                 etape_4_crypt(bits_octet, nb_iteration, etape, key);
513.                 pass_decimal(bits_octet, octet);
514.                 etape_5_crypt(octet);
515.               }
516.
517.
518.               //Ecriture des octets dans file_out
519.               for (int i = 0; i < NB_OCTET; i++)
520.                 fprintf(file_out, "%c", octet[i]);
521.
522.             }
523.           }
524.         }
525.
526.         // DECRYPTAGE : lecture et écriture fichier jusqu'à sa fin (4 par 4 octets)
527.         else {
528.           while (arret == FALSE) {
529.
530.             //Affichage progression décryptage
531.             cur_size = ftell(file_out);
532.             if ((cur_size * 100 / tot_size) / percent == 1 ){
533.               printf("Fichier converti : %d %%\n", cur_size * 100 / tot_size);
534.               percent += 20;
535.             }
536.
537.             // Lecture des octets 4 par 4
538.             for (int i = 0; i < NB_OCTET; i++)
539.               octet[i] = fgetc(file_in);
540.
541.             //Lecture des 2 octets du prochain paquet pour savoir si c'est la fin
542.             for(int i = 0; i < 2; i++)
543.               rech_bourrage[i] = fgetc(file_in);
544.
545.             // condition fin vérifiée : note nombre d'octets pour ne pas les recopier
546.             if (rech_bourrage[1] == EOF) {
547.               //pour ne pas relire 4 octet
548.               arret = TRUE;
```

```
549.               nb_bourrage = rech_bourrage[0];
550.             }
551.             //Retour du curseur 2 crans en arrière pour pouvoir continuer tranquillement

552.             else fseek (file_in, -2, SEEK_CUR);
553.
554.             //DECRYPTAGE ITERATIF avec contrôle des passages binaire/decimal
555.             for (int etape = 0; etape < nb_iteration; etape++){
556.               etape_5_decrypt(octet);
557.               pass_binary(octet, bits_octet);
558.               etape_4_decrypt(bits_octet, nb_iteration, etape, key);
559.               etape_3_decrypt(bits_octet);
560.               pass_decimal(bits_octet, octet);
561.               etape_2_crypt_decrypt(octet);
562.               etape_1_decrypt(octet, key);
563.             }
564.
565.             //Ecriture octets dans file_out tout en enlevant octets bourrage s'il y en a
566.             for (int i = 0; i < NB_OCTET - nb_bourrage; i++)
567.               fprintf(file_out, "%c", octet[i]);
568.
569.           }
570.         }
571.
572.       //Affichage manuel 100% fichier converti (nb bourrage et EOF non lu)
573.         if (choix_ops == DECRYPT)
574.           printf("Fichier converti : 100 %%\n");
575.         // Noter le nombre d'octet de bourrage en fin de fichier durant le cryptage
576.         if (choix_ops == CRYPT)
577.           fprintf (file_out, "%c", nb_bourrage);
578.
579.         //Fermeture des fichiers
580.         fclose(file_in);
581.         fclose(file_out);
582.
583.         // Libération espaces alloués dynamiquement
584.         free(octet);
585.         for(int i = 0; i < NB_OCTET; i++)
586.           free(bits_octet[i]);
587.         free(bits_octet);
588.         free(key);
589.         free(name_in);
590.         free(name_out);
591.         free(extension);
592.
593.         // Affichage temps d'éxécution de l'opération
594.         clock_t fin = clock();
595.         int temps = (fin - debut) / CLOCKS_PER_SEC;
596.         int heures = (int) temps / 3600;
597.         int minutes = ((int) temps - 3600 * heures)/ 60;
598.         int secondes = (int) temps - 3600 * heures - minutes * 60;
599.         printf("\n\nTemps d'execution de %d heures, %d minutes, %d secondes\n\n", heures,
    minutes,secondes);
600.
601.         printf("L'operation s'est parfaitement bien passee\n");
602.
603.         return EXIT_SUCCESS;
604.       }
```