**Conversion de l’encrypteur de C# à C**

Le C est un langage de très bas niveau, il est près de la machine. Il a servi de base au C++ qui a, à partir duquel le C# a été créé. La syntaxe en C est très similaire à celle utilisé en C# mais il y a beaucoup moins de fonctions incluses.

Voici les fonctionnalités qui devaient être intégrées:

* Lire séquentiellement dans un fichier
* Écrire séquentiellement et par bloque dans un fichier
* Ajouter et retirer des caractères (les masques)
* Substituer des caractères lors de l’écriture
* Modifier aléatoirement l’ordre des caractères dans les clés
* Inversion des clés 1 et 3 en lecture et écriture

La méthode d’encryption sera sensiblement la même; on trouve l’index du caractère sur la clé et on le substitue pour celui à ce même index mais sur la clé de base. Si le caractère n’est pas dans la clé on le réécrit sans changement. On ajoute 1 caractère aléatoire et 4 chiffres (nombre aléatoire entre 0 et 9999).

L’écriture et la lecture de fichier est assez simple, il suffit de l’ouvrir avec fopen et effectuer opération désiré (fread, fwrite ou fprintf).

* Fread : lie les données du fichier pour un nombre de bits défini et les écrits dans une variable.
* Fwrite : Inverse de fread, écrit les données contenues dans une variable dans le fichier.
* Fprintf : Écrit une string dans le fichier en block, inclus les caractères de contrôle si il y en a donc il faut bien formater la string.

Trois programmes distincts ont été écrits pour les différentes manipulations : yates.c pour la création des clés, encrypt.c pour l’encryption de fichier et decryptk.c pour la décryption.

La création de clé est plus complexe en C puisqu’il est un peu plus difficile de générer aléatoirement tous les nombres dans un certain intervalle (ie 0-69). Il n’y a pas de méthode permettant de limiter les valeurs maximales du générateur, les nombres sont de 0 à MAX\_RAND. On doit utiliser l’opérateur modulo pour limiter les résultats. La méthode que j’avais utilisé en C# (générer des nombres de 0 à 69 jusqu’à ce que tous les index soient obtenus) n’est pas fonctionnelle en C. Le temps pour trouver les 6 – 7 derniers index est très grand, approchant l’infini théorique. En effet cette méthode à un meilleur cas de O(n) mais un pire cas de O(∞).

J’ai donc dû trouver une meilleure solution. Je suis tombé sur l’algorithme de Yates et Fisher (Yates & Fisher Shuffle) qui est utilisé pour modifier l’ordre d’une suite aléatoirement. Au lieu de chercher tous les index, il suffit de générer n – 1 nombre aléatoire, l’intervalle de ‘recherche’ est de n – i, il diminue donc à chaque itération. On obtient donc un rendement de O(n-1) qui est BEAUCOUP plus efficace que ma méthode initiale. Voici un tableau représentant la méthode de Yates.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Visual representation of a Yates Shuffle | | | | | | | | | | | |
| Index | | | | | | | | | RANGE | ROLL | i |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| A | B | C | D | E | F | G | H | I | %9 | 5 | 9 |
| A | B | C | D | E | I | G | H | F | %8 | 3 | 8 |
| A | B | C | H | E | I | G | D |  | %7 | 4 | 7 |
| A | B | C | H | G | I | E |  |  | %6 | 1 | 6 |
| A | I | C | H | G | B |  |  |  | %5 | 2 | 5 |
| A | I | C | G | H |  |  |  |  | %4 | 3 | 4 |
| G | I | C | A |  |  |  |  |  | %3 | 0 | 3 |
| G | C | I |  |  |  |  |  |  | %2 | 1 | 2 |
| G | C |  |  |  |  |  |  |  | %1 | 0 | 1 |
| G | C | I | A | H | B | E | D | F | OUTPUT | | |
| A | B | C | D | E | F | G | H | I | INPUT | | |
| © Dominic Letourneau 2016 | | | | | | | | | | | |

La méthode de Yates est assez simple, on génère un chiffre aléatoire entre 0 et l’index maximal de l’array; on substitue l’élément situé à cet index au dernier élément de l’array. On génère ensuite un second index entre 0 et l’index maximal moins 1, donc on ne touche pas au dernier et on refait la substitution et ainsi de suite jusqu’à ce qu’un nombre aléatoire ai été généré pour chaque index de l’array.

Il est théoriquement possible que l’array produit soit exactement l’inverse ou une copie de l’original mais la probabilité est très base. Cette méthode est aussi dépendante du générateur de nombres aléatoires. Une mauvaise génération aura tendance à générer des séries similaires. Mon implémentation de cette méthode semble suffisamment aléatoire pour les besoins de ce projet.

Le programme d’encryption agit en deux phases. La première consiste à lire les clés dans le fichier envoyé en paramètre, les chargées en mémoires dans un array 2D et de les écrire dans le fichier de sortie. Lors de la deuxième est un peu plus complexe; on substitue chaque caractère selon la méthode d’encryption et on ajoute les éléments de camouflage. Pour respecter le formatage initial du texte les caractères de contrôle sont reproduits.

Le processus inverse requière aussi 2 opérations distinctes. Les 284 premiers caractères sont lus (4 clés de 70 caractère et les terminateurs), et mis en mémoire dans un array 4 x 70. Par la suite la décryption commence, puisque seulement le premier caractère de chaque bloc de 6 est valide on ne traite que celui-là. On fait la substitution inverse et on écrit le caractère décrypté dans le fichier de sortie. Toujours pour respecter le formatage original, on reproduit les caractères de contrôle sans modifications.

Un shell script a été créé pour automatiser l’encryption, il génère un groupe de clé et l’utilise pour encrypter le fichier texte. L’utilisateur doit entrer le nom des fichiers de source et de sortie. Le fichier de clé et de source sont détruit. Un autre script a été écrit pour la décryption, encore une fois l’utilisateur entre les noms de fichiers et la source est détruite.

Tous les fichiers sont disponibles sur GitHub : <https://github.com/DLetourneau/Crypto>.