Le programme de l'algorithme AES est divisé en 3 fichiers. Les fonctions sont définies et explicitées dans les 2 fichiers fonctions.h et fonctions.cpp. Le fichier main.cpp est le fichier principal au sein duquel sont appelées les différentes fonctions pour faire tourner le programme.

Au sein du fichier main.cpp sont définies toutes les variables nécessaires telles que la sbox (pour le chiffrement), l'inverse de la sbox (pour le déchiffrement), la matrice rcon pour le key expansion, la matrice du message initial ainsi que la clef de chiffrement.

Dans la théorie, l'algorithme AES fonctionne sur 10 tours. Dans la pratique, j'ai essayé d'implémenter la fonction de key expansion mais j'ai une erreur de mémoire qui se produit lorsque je veux exécuter la fonction SubBytes par la suite. Donc seule la première itération de l'algorithme fonctionne, c'est-à-dire toutes les étapes de chiffrement + déchiffrement avant d'entrer dans la boucle utilisant la key expansion.

Dans mon programme, j'ai fait le choix de représenter le message codé sous la forme d'une matrice 4x4 et de même pour la clef de déchiffrement. Les sbox, invsbox et rconbox sont des listes de dimension 256 car il est plus simple dans ce cas de parcourir une liste plutôt qu'une matrice 2 dimensions.

Le premier round est l'enchainement des fonctions suivantes :

```
//chiffrement
SubBytes(matrice , sbox);
ShiftRows(matrice);
MixColumns(matrice[0]);
MixColumns(matrice[1]);
MixColumns(matrice[2]);
MixColumns(matrice[3]);
AddRoundKey(matrice, clef);
//dechiffrement
InvAddRoundKey(matrice, clef);
InvMixColumns(matrice[0]);
InvMixColumns(matrice[1]);
InvMixColumns(matrice[2]);
InvMixColumns(matrice[3]);
InvShiftRows(matrice);
InvSubBytes(matrice , invsbox);
```

Détail de chacune des fonctions :

- **SubBytes** : cette fonction a pour but de remplacer chaque valeur du message initial, donc de la matrice comme expliqué précédemment, par la correspondance dans la sbox. Pour cela, via une double itération sur les lignes et colonnes de la matrice (2 boucles for()), je viens remplacer la valeur de la i-ème ligne et j-ième colonne de la matrice par son équivalent dans la sbox via la ligne suivante :

  matrice[i][j] = sbox[matrice[i][j]];

  En effet, le fait d'avoir mis la sbox sous forme de liste et non d'une matrice permet de directement récupérer la bonne valeur par l'indice retourné par la matrice.

- **ShiftRows** : cette fonction ne prend que la matrice du message en paramètre. Je viens faire une translation des valeurs des lignes 2, 3 et 4 de la matrice de manière que les valeurs de la diagonale se retrouvent dans la première colonne. Pour cela, j'utilise pour chaque ligne une variable temporaire à laquelle j'assigne la valeur pivot, puis je remplace chaque valeur par sa nouvelle valeur, en finissant par assigner la dernière par la valeur pivot.
  Exemple :
  ```
  temp = matrice[1][1];
  matrice[1][1] = matrice[1][2];
  matrice[1][2] = matrice[1][3];
  matrice[1][3] = matrice[1][0];
  matrice[1][0] = temp;
  ```

- **MixColumns** : cette fonction prend un unique paramètre qui est une seule des 4 colonnes de la matrice. Je récupère les 4 colonnes dans le main() à l'aide de boucles for(). Ensuite, j'appelle 4 fois la fonction MixColumns, une fois par colonne de la matrice du message. Au sein de cette fonction je viens copier la matrice dans une variable a et également dans une variable b, variable b qui est multipliée par 2 via un décalage de 1 bit sur la gauche. Enfin, je viens appliquer la transformation polynomiale suivante : (les indices 0 à 3 étant les positions des valeurs dans les colonnes)
  - 1ère valeur : $2 * a0 + a3 + a2 + 3 * a1$
  - 2nde valeur : $2 * a1 + a0 + a3 + 3 * a2$
  - 3ème valeur : $2 * a2 + a1 + a0 + 3 * a3$
  - 4ème valeur : $2 * a3 + a2 + a1 + 3 * a0$

  Pour ce faire, j'ai réalisé des additions bit à bit en décomposant de la sorte : si la valeur a est un multiple de 2 alors j'utilise x*b, si c'est un nombre impair : x*b+a :
  - matrice[0] = b[0] ^ a[3] ^ a[2] ^ b[1] ^ a[1];
  - matrice[1] = b[1] ^ a[0] ^ a[3] ^ b[2] ^ a[2];
  - matrice[2] = b[2] ^ a[1] ^ a[0] ^ b[3] ^ a[3];
  - matrice[3] = b[3] ^ a[2] ^ a[1] ^ b[0] ^ a[0];

- **AddRoundKey** : cette fonction prend la clef et la matrice du message en paramètres. Elle sert à faire une simple addition bit à bit des valeurs aux mêmes positions dans les 2 matrices. J'ai codé cela en faisant une double itération sur les lignes colonnes des 2 matrices :
  ```
  for(i = 0; i < 4; i++){
      for(j = 0; j < 4; j++){
          matrice[i][j] = matrice[i][j] ^ clef[i][j];
      }
  }
  ```

- **InvAddRoundKey** : cette fonction est exactement la même que celle précitée, la seule différence est que la matrice du message d'entrée et celle qui résulte de la fonction AddRoundKey, donc différente de celle en paramètre de la fonction précédente.

- **InvMixColumns** : Cette fonction fonctionne comme la fonction MixColumns, la différence réside dans les polynomes utilisées pour obtenir le résultat. Dans ce cas les polynômes sont les suivants :

- 1ère valeur : `14 * a[0] + 9 * a[3] + 13 * a[2] + 11 * a[1]`
- 2nde valeur : `14 * a[1] + 9 * a[0] + 13 * a[3] + 11 * a[2]`
- 3ème valeur : `14 * a[2] + 9 * a[1] + 13 * a[0] + 11 * a[3]`
- 4ème valeur : `14 * a[3] + 9 * a[2] + 13 * a[1] + 11 * a[0]`

De la même manière, si la valeur a est un multiple de 2 alors j'utilise x*b, si c'est un nombre impair : x*b+a, ce qui donne :

- matrice[0] = b[0] ^ b[0] ^ b[0] ^ b[0] ^ b[0] ^ b[0] ^ b[0] ^ b[3] ^ b[3] ^ b[3] ^ b[3] ^ a[3] + b[2] ^ b[2] ^ b[2] ^ b[2] ^ b[2] ^ b[2] ^ a[2] ^ b[1] ^ b[1] ^ b[1] ^ b[1] ^ b[1] ^ a[1];
- matrice[1] = b[1] ^ b[1] ^ b[1] ^ b[1] ^ b[1] ^ b[1] ^ b[1] ^ b[0] ^ b[0] ^ b[0] ^ b[0] ^ a[0] + b[3] ^ b[3] ^ b[3] ^ b[3] ^ b[3] ^ b[3] ^ a[3] ^ b[2] ^ b[2] ^ b[2] ^ b[2] ^ b[2] ^ a[2];
- matrice[2] = b[2] ^ b[2] ^ b[2] ^ b[2] ^ b[2] ^ b[2] ^ b[2] ^ b[1] ^ b[1] ^ b[1] ^ b[1] ^ a[1] + b[0] ^ b[0] ^ b[0] ^ b[0] ^ b[0] ^ b[0] ^ a[0] ^ b[3] ^ b[3] ^ b[3] ^ b[3] ^ b[3] ^ a[3];
- matrice[3] = b[3] ^ b[3] ^ b[3] ^ b[3] ^ b[3] ^ b[3] ^ b[3] ^ b[2] ^ b[2] ^ b[2] ^ b[2] ^ a[2] + b[1] ^ b[1] ^ b[1] ^ b[1] ^ b[1] ^ b[1] ^ a[1] ^ b[0] ^ b[0] ^ b[0] ^ b[0] ^ b[0] ^ a[0];

- **InvShiftRows** : tout comme la fonction ShiftRows, celle-ci prend la matrice du message codé en paramètre. Au lieu de prendre les valeurs de la diagonale et de les mettre dans la première colonne en décalant les autres valeurs sur les lignes, cette fonction agit à l'inverse. Chaque valeur positionnée sur la première colonne est mise dans la diagonale et les autres valeurs de chaque ligne sont décalées en conséquence.
  Exemple :
  ```
  temp = matrice[1][0];
  matrice[1][0] = matrice[1][3];
  matrice[1][2] = matrice[1][1];
  matrice[1][3] = matrice[1][2];
  matrice[1][1] = temp;
  ```

- **InvSubBytes** : cette fonction prend en paramètres la matrice du message codé ainsi que l'inverse de la matrice sbox. Chaque valeur de la matrice du message codé est remplacée par sa correspondance dans la matrice inverse sbox. Cette étape identique à la fonction SubBytes à la différence près que le paramètres est invsbox et non sbox cette fois.

Au niveau de l'implémentation de la fonction de Key Expansion, j'ai procédé comme suit :
J'ai créé la fonction RotWord. Celle-ci prend en paramètres une colonne de 4 valeurs et effectue une rotation, comme la fonction ShiftRows.
Ensuite, j'ai codé la fonction SubWord. Elle fonctionne comme la fonction SubBytes, chaque valeur est remplacée par celle de son indice dans la liste sbox.

Ces fonctions servent seront ensuite appelées dans la fonction KeyExpansion :
Cette fonction prend en paramètre le round (entre 2 et 10 car le premier est hors de la boucle), la matrice du message à coder, une expended key qui est la clef de codage cumulée à chaque étape (4 lignes, 40 colonnes), la sbox et la rconbox).

Dans un premier temps je calcule en fonction du round, l'index des colonnes à utiliser dans la expended key :

```
int firstCol = round * 4 - 4;
int secondCol = round * 4 - 3;
int thirdCol = round * 4 - 2;
int lastCol = round * 4 - 1;

int previousFirstCol = round * 4 - 8;
int previousSecondCol = round * 4 - 7;
int previousThirdCol = round * 4 - 6;
int previousLastCol = round * 4 - 5;
```

Ensuite, en iterant je viens récupérer la nouvelle clef de codage que je stocke dans la variable key.
J'applique la fonction RotWord précédemment créé, en passant la clef qui vient d'être mise à jour en paramètre.
Ensuite, j'applique la fonction SubWord, qui prend la clef et la sbox en paramètres.
L'étape suivante est une addition bit à bit entre le résultat de la fonction précédente et l'expended key (première colonne de la clef précédente).
Ensuite, j'assigne dans une variable temporaire une liste contenant en première position la valeur de la rconbox à l'index du round en question (exemple : si round 4, rconbox[4]), suivi de 3 valeurs à 0 : int temp4[4] = {rconbox, 0x00, 0x00, 0x00};

L'étape suivante est une addition bit à bit entre le résultat de l'addition bit à bit précédente et la variable créé contenant la valeur de la rconbox.
Ce résultat est la première colonne de la nouvelle clef.

Pour les 3 autres colonnes de la nouvelle clef, il s'agit de faire une addition bit à bit entre la colonne précédente de la nouvelle clef avec la colonne du même index de la précédente clef.
Exemple : colonne2-round3 = colonne1-round3 + colonne2-round2
colonne4-round4 = colonne3-round4 + colonne4-round3

je n'ai cependant pas réussi à faire fonctionner cette fonction de key expansion.