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

// --- Définitions ---

#define MAX_ALPHABET_SIZE 100
#define MAX_MESSAGE_LENGTH 512
#define MAX_MATRIX_DIMENSION 10

// Structure pour représenter une matrice
typedef struct {
    int rows;
    int cols;
    int data[MAX_MATRIX_DIMENSION][MAX_MATRIX_DIMENSION];
} Matrix;

// Variables globales
char alphabet[MAX_ALPHABET_SIZE] =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'ùéèàç- ê .";
int alphabet_len = sizeof(alphabet) - 1;

// --- Utilitaires ---

// Fonction pour générer un nombre aléatoire dans une plage donnée
int random_range(int min, int max) {
    return min + rand() % (max - min + 1);
}

// Fonction pour mélanger un tableau (algorithme de Fisher-Yates)
void shuffle_array(int arr[], int n) {
    for (int i = n - 1; i > 0; i--) {
        int j = random_range(0, i);
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
}

// Fonction pour créer un alphabet unique (légère altération)
void create_unique_alphabet() {
    int indices[MAX_ALPHABET_SIZE];
    for (int i = 0; i < alphabet_len; i++) {
        indices[i] = i;
    }
    shuffle_array(indices, alphabet_len - 5); // Mélanger une partie de
l'alphabet
    char temp_alphabet[MAX_ALPHABET_SIZE];
    strcpy(temp_alphabet, alphabet);
    for (int i = 0; i < alphabet_len - 5; i++) {
        alphabet[i] = temp_alphabet[indices[i]];
    }
    // Ajouter une petite modification déterministe
    if (alphabet_len > 0) {
```

```
        alphabet[alphabet_len - 1] = (alphabet[alphabet_len - 1] + 1) %
128;
    }
}

// Fonction pour trouver l'index d'un caractère dans l'alphabet
int get_char_index(char c) {
    for (int i = 0; i < alphabet_len; i++) {
        if (alphabet[i] == c) {
            return i;
        }
    }
    return -1;
}

// --- Test de la matrice ---

// Fonction pour calculer le déterminant d'une matrice (méthode récursive
pour plus de généralité)
int determinant(Matrix m, int n) {
    int det = 0;
    Matrix submatrix;
    if (n == 1) {
        return m.data[0][0];
    }
    if (n == 2) {
        return (m.data[0][0] * m.data[1][1]) - (m.data[0][1] *
m.data[1][0]);
    }
    for (int x = 0; x < n; x++) {
        int subi = 0;
        for (int i = 1; i < n; i++) {
            int subj = 0;
            for (int j = 0; j < n; j++) {
                if (j == x) {
                    continue;
                }
                submatrix.data[subi][subj] = m.data[i][j];
                subj++;
            }
            subi++;
        }
        det += (x % 2 == 0 ? 1 : -1) * m.data[0][x] *
determinant(submatrix, n - 1);
    }
    return det;
}

// Fonction pour trouver l'inverse modulaire
int modular_inverse(int a, int m) {
    a %= m;
    for (int x = 1; x < m; x++) {
        if ((a * x) % m == 1) {
            return x;
        }
    }
    return -1;
}
```

```
// Fonction pour tester si une matrice est inversible modulo l'alphabet
bool is_invertible_mod_alphabet(Matrix m, int dim, int* inverse_det) {
    int det = determinant(m, dim);
    if (det == 0) {
        return false;
    }
    *inverse_det = modular_inverse(det % alphabet_len, alphabet_len);
    return *inverse_det != -1;
}

// --- Saisie ---

// Fonction pour saisir une matrice
Matrix saisir_matrice() {
    Matrix m;
    int dim;
    printf("Entrez la dimension de la matrice (max %d) : ",
MAX_MATRIX_DIMENSION);
    if (scanf("%d", &dim) != 1 || dim < 1 || dim > MAX_MATRIX_DIMENSION)
{
        printf("Dimension invalide.\n");
        exit(EXIT_FAILURE);
    }
    m.rows = dim;
    m.cols = dim;
    printf("Les coefficients de la matrice vont être entrés ligne par
ligne, séparés par des espaces.\n");
    for (int i = 0; i < dim; i++) {
        printf("Ligne %d : ", i + 1);
        for (int j = 0; j < dim; j++) {
            if (scanf("%d", &m.data[i][j]) != 1) {
                printf("Erreur de saisie des coefficients.\n");
                exit(EXIT_FAILURE);
            }
        }
    }

    int inverse_det;
    if (!is_invertible_mod_alphabet(m, dim, &inverse_det)) {
        printf("\nMalheureusement, cette matrice n'est pas inversible
modulo %d.\nVeuillez en saisir une autre.\n", alphabet_len);
        // Gérer la récursion avec une limite pour éviter le débordement
de pile
        static int recursion_depth = 0;
        if (recursion_depth < 5) {
            recursion_depth++;
            return saisir_matrice();
        } else {
            printf("Nombre maximal de tentatives de saisie de matrice
atteint.\n");
            exit(EXIT_FAILURE);
        }
    }
    return m;
}

// Fonction pour saisir un message
```

```
char* saisir_message() {
    char* mot = (char*)malloc(MAX_MESSAGE_LENGTH * sizeof(char));
    if (mot == NULL) {
        perror("Erreur d'allocation mémoire");
        exit(EXIT_FAILURE);
    }
    printf("\nEntrez le message (max %d caractères) : ",
MAX_MESSAGE_LENGTH - 1);
    if (fgets(mot, MAX_MESSAGE_LENGTH, stdin) == NULL) {
        printf("Erreur de lecture du message.\n");
        free(mot);
        exit(EXIT_FAILURE);
    }
    mot[strcspn(mot, "\n")] = 0; // Supprimer le caractère de nouvelle
ligne
    return mot;
}

// --- Chiffrement ---

char* chiffrement(Matrix m, const char* mot) {
    int dim = m.rows;
    size_t mot_len = strlen(mot);
    size_t padding = (dim - (mot_len % dim)) % dim;
    char* padded_mot = (char*)malloc((mot_len + padding + 1) *
sizeof(char));
    if (!padded_mot) {
        perror("Erreur d'allocation mémoire");
        exit(EXIT_FAILURE);
    }
    strcpy(padded_mot, mot);
    for (size_t i = 0; i < padding; i++) {
        padded_mot[mot_len + i] = ' ';
    }
    padded_mot[mot_len + padding] = '\0';
    size_t padded_len = strlen(padded_mot);
    char* chiffre = (char*)malloc((padded_len + 1) * sizeof(char));
    if (!chiffre) {
        perror("Erreur d'allocation mémoire");
        free(padded_mot);
        exit(EXIT_FAILURE);
    }
    chiffre[0] = '\0';

    for (size_t i = 0; i < padded_len; i += dim) {
        for (int ligne = 0; ligne < dim; ligne++) {
            int a = 0;
            for (int colonne = 0; colonne < dim; colonne++) {
                int index = get_char_index(padded_mot[i + colonne]);
                if (index == -1) {
                    printf("Caractère non supporté: %c\n", padded_mot[i +
colonne]);
                    free(padded_mot);
                    free(chiffre);
                    exit(EXIT_FAILURE);
                }
                a += m.data[ligne][colonne] * index;
            }
```

```
            a %= alphabet_len;
            strncat(chiffre, &alphabet[a], 1);
        }
    }
    free(padded_mot);
    return chiffre;
}

// --- Déchiffrement ---

// Fonction pour calculer la matrice adjointe (pour une matrice 2x2,
simplifiée)
void adjoint_matrix_2x2(Matrix in, Matrix* out) {
    out->rows = 2;
    out->cols = 2;
    out->data[0][0] = in.data[1][1];
    out->data[0][1] = -in.data[0][1];
    out->data[1][0] = -in.data[1][0];
    out->data[1][1] = in.data[0][0];
}

char* dechiffrement(Matrix m, const char* mot) {
    int dim = m.rows;
    size_t mot_len = strlen(mot);
    if (mot_len % dim != 0) {
        printf("La longueur du message chiffré n'est pas un multiple de
la dimension de la matrice.\n");
        return NULL;
    }

    int det = determinant(m, dim);
    int inv_det = modular_inverse(det % alphabet_len, alphabet_len);

    if (inv_det == -1) {
        printf("La matrice n'est pas inversible pour le
déchiffrement.\n");
        return NULL;
    }

    Matrix inv_matrix;
    inv_matrix.rows = dim;
    inv_matrix.cols = dim;

    if (dim == 2) {
        adjoint_matrix_2x2(m, &inv_matrix);
        for (int i = 0; i < dim; i++) {
            for (int j = 0; j < dim; j++) {
                inv_matrix.data[i][j] = (inv_matrix.data[i][j] * inv_det
% alphabet_len + alphabet_len) % alphabet_len;
            }
        }
    } else {
        printf("Le déchiffrement pour les matrices de dimension
supérieure à 2 n'est pas implémenté dans cette version simplifiée.\n");
        return NULL;
    }

    char* dechiffre = (char*)malloc((mot_len + 1) * sizeof(char));
```

```
    if (!dechiffre) {
        perror("Erreur d'allocation mémoire");
        exit(EXIT_FAILURE);
    }
    dechiffre[0] = '\0';

    for (size_t i = 0; i < mot_len; i += dim) {
        for (int ligne = 0; ligne < dim; ligne++) {
            int a = 0;
            for (int colonne = 0; colonne < dim; colonne++) {
                int index = get_char_index(mot[i + colonne]);
                if (index == -1) {
                    printf("Caractère non supporté dans le message
chiffré: %c\n", mot[i + colonne]);
                    free(dechiffre);
                    return NULL;
                }
                a += inv_matrix.data[ligne][colonne] * index;
            }
            a %= alphabet_len;
            strncat(dechiffre, &alphabet[a], 1);
        }
    }

    return dechiffre;
}

// --- Menu ---

void menu_principal(Matrix m, char* mot) {
    char choix[2];
    printf("\nSouhaitez-vous Chiffrer (C) ou Déchiffrer (D) le message
saisi ? ");
    if (scanf("%1s", choix) != 1) {
        printf("Erreur de saisie.\n");
        // Gérer l'erreur de saisie
        while (getchar() != '\n'); // Nettoyer le buffer d'entrée
        menu_principal(m, mot);
        return;
    }
    while (getchar() != '\n'); // Nettoyer le buffer d'entrée

    if (toupper(choix[0]) == 'C') {
        char* chiffre = chiffrement(m, mot);
        printf("\nLe message chiffré est : %s (%zu caractères)\n",
chiffre, strlen(chiffre));
        free(chiffre);
    } else if (toupper(choix[0]) == 'D') {
        char* dechiffre = dechiffrement(m, mot);
        if (dechiffre != NULL) {
            printf("\nLe message déchiffré est : %s (%zu caractères)\n",
dechiffre, strlen(dechiffre));
            free(dechiffre);
        }
    } else {
        printf("Choix invalide.\n");
        menu_principal(m, mot);
    }
```

```
}

int main() {
    srand(time(NULL));
    create_unique_alphabet();

    printf("\n-------------------\nChiffrement de Hill\n-----------------
--\n");
    printf("Pour chiffrer ou déchiffrer un message, il faut une matrice
carrée inversible.\n");

    Matrix matrice_hill = saisir_matrice();
    char* message = saisir_message();

    menu_principal(matrice_hill, message);

    free(message);

    printf("\nOpération terminée.\n");
    return 0;
}
```