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

#define ALPHABET_SIZE 37

int char_to_num(char c) {
    if (c >= 'A' && c <= 'Z') return c - 'A';
    if (c >= '0' && c <= '9') return c - '0' + 26;
    if (c == ' ') return 36;
    return -1;
}

char num_to_char(int n) {
    if (n < 26) return 'A' + n;
    if (n < 36) return '0' + n - 26;
    return ' ';
}

int** transpose_matrix(int** matrix, int dim) {
    int** transpose = (int**)malloc(dim * sizeof(int*));
    for (int i = 0; i < dim; i++) {
        transpose[i] = (int*)malloc(dim * sizeof(int));
        for (int j = 0; j < dim; j++) {
            transpose[i][j] = matrix[j][i];
        }
    }
    return transpose;
}

int determinant_matrix(int** matrix, int dim) {
    if (dim == 1) return matrix[0][0] % ALPHABET_SIZE;
    if (dim == 2) {
        int det = (matrix[0][0] * matrix[1][1] - matrix[0][1] *
matrix[1][0]) % ALPHABET_SIZE;
        return (det + ALPHABET_SIZE) % ALPHABET_SIZE;
    }

    int det = 0;
    for (int i = 0; i < dim; i++) {
        int** minor = (int**)malloc((dim - 1) * sizeof(int*));
        for (int j = 1; j < dim; j++) {
            minor[j - 1] = (int*)malloc((dim - 1) * sizeof(int));
            for (int k = 0, l = 0; k < dim; k++) {
                if (k != i) minor[j - 1][l++] = matrix[j][k];
            }
        }
        int sign = (i % 2 == 0) ? 1 : -1;
        det = (det + sign * matrix[0][i] * determinant_matrix(minor, dim
- 1)) % ALPHABET_SIZE;

        for (int j = 0; j < dim - 1; j++) free(minor[j]);
        free(minor);
    }
    return (det + ALPHABET_SIZE) % ALPHABET_SIZE;
}
```

```
int** cofactor_matrix(int** matrix, int dim) {
    int** cofactor = (int**)malloc(dim * sizeof(int*));
    for (int i = 0; i < dim; i++) {
        cofactor[i] = (int*)malloc(dim * sizeof(int));
        for (int j = 0; j < dim; j++) {
            int** minor = (int**)malloc((dim - 1) * sizeof(int*));
            for (int k = 0; k < dim - 1; k++) {
                minor[k] = (int*)malloc((dim - 1) * sizeof(int));
            }

            for (int k = 0, ki = 0; k < dim; k++) {
                if (k == i) continue;
                for (int l = 0, lj = 0; l < dim; l++) {
                    if (l == j) continue;
                    minor[ki][lj++] = matrix[k][l];
                }
                ki++;
            }

            cofactor[i][j] = ((i + j) % 2 == 0 ? 1 : -1) *
determinant_matrix(minor, dim - 1);

            for (int k = 0; k < dim - 1; k++) free(minor[k]);
            free(minor);
        }
    }
    return cofactor;
}

int modInverse(int a, int m) {
    a = a % m;
    for (int x = 1; x < m; x++) {
        if ((a * x) % m == 1) return x;
    }
    return -1;
}

int** inverse_matrix(int** matrix, int dim) {
    int det = determinant_matrix(matrix, dim);
    if (det == 0) {
        printf("La matrice n'est pas inversible (determinant nul)\n");
        return NULL;
    }

    int det_inv = modInverse(det, ALPHABET_SIZE);
    if (det_inv == -1) {
        printf("Pas d'inverse pour le determinant %d modulo %d\n", det,
ALPHABET_SIZE);
        return NULL;
    }

    int** cofactor = cofactor_matrix(matrix, dim);
    int** adjugate = transpose_matrix(cofactor, dim);
    int** inverse = (int**)malloc(dim * sizeof(int*));

    for (int i = 0; i < dim; i++) {
        inverse[i] = (int*)malloc(dim * sizeof(int));
        for (int j = 0; j < dim; j++) {
```

```
            inverse[i][j] = (adjugate[i][j] * det_inv) % ALPHABET_SIZE;
            if (inverse[i][j] < 0) inverse[i][j] += ALPHABET_SIZE;
        }
    }

    for (int i = 0; i < dim; i++) {
        free(cofactor[i]);
        free(adjugate[i]);
    }
    free(cofactor);
    free(adjugate);

    return inverse;
}

void hill_encrypt(char* message, int** K, int dim) {
    int len = strlen(message);

    int pad_len = (dim - (len % dim)) % dim;
    char* padded = (char*)malloc(len + pad_len + 1);
    strcpy(padded, message);
    for (int i = 0; i < pad_len; i++) padded[len + i] = ' ';
    padded[len + pad_len] = '\0';
    len += pad_len;

    for (int i = 0; i < len; i += dim) {
        int* block = (int*)malloc(dim * sizeof(int));
        for (int j = 0; j < dim; j++) {
            block[j] = char_to_num(padded[i + j]);
        }

        int* encrypted = (int*)malloc(dim * sizeof(int));
        for (int j = 0; j < dim; j++) {
            encrypted[j] = 0;
            for (int k = 0; k < dim; k++) {
                encrypted[j] += K[j][k] * block[k];
            }
            encrypted[j] = (encrypted[j] % ALPHABET_SIZE + ALPHABET_SIZE)
% ALPHABET_SIZE;
        }

        for (int j = 0; j < dim; j++) {
            padded[i + j] = num_to_char(encrypted[j]);
        }

        free(block);
        free(encrypted);
    }

    strcpy(message, padded);
    free(padded);
}

void hill_decrypt(char* message, int** K_inv, int dim) {
    int len = strlen(message);
    char* decrypted = (char*)malloc(len + 1);
    strcpy(decrypted, message);
```

```
    for (int i = 0; i < len; i += dim) {
        int* block = (int*)malloc(dim * sizeof(int));
        for (int j = 0; j < dim; j++) {
            if (i + j < len) {
                block[j] = char_to_num(decrypted[i + j]);
            } else {
                block[j] = char_to_num(' ');
            }
        }

        int* decrypted_block = (int*)malloc(dim * sizeof(int));
        for (int j = 0; j < dim; j++) {
            decrypted_block[j] = 0;
            for (int k = 0; k < dim; k++) {
                decrypted_block[j] += K_inv[j][k] * block[k];
            }
            decrypted_block[j] = (decrypted_block[j] % ALPHABET_SIZE +
ALPHABET_SIZE) % ALPHABET_SIZE;
        }

        for (int j = 0; j < dim && (i + j) < len; j++) {
            decrypted[i + j] = num_to_char(decrypted_block[j]);
        }

        free(block);
        free(decrypted_block);
    }

    int i = len - 1;
    while (i >= 0 && decrypted[i] == ' ') i--;
    decrypted[i + 1] = '\0';

    strcpy(message, decrypted);
    free(decrypted);
}

int is_invertible(int **matrix, int dim) {
    int det = determinant_matrix(matrix, dim);
    if (det == 0) return 0;
    return (modInverse(det, ALPHABET_SIZE) != -1);
}

int main() {
    char nom[100];
    printf("Entrez votre nom : ");
    scanf("%99s", nom);

    int dim;
    printf("Entrez la dimension de la matrice: ");
    scanf("%d", &dim);

    int** k = (int**)malloc(dim * sizeof(int*));
    for (int i = 0; i < dim; i++) {
        k[i] = (int*)malloc(dim * sizeof(int));
    }

    int is_valid;
    do {
```

```
        printf("\nEntrez les elements de la matrice de chiffrement
(dimension %dx%d):\n", dim, dim);
        for (int i = 0; i < dim; i++) {
            for (int j = 0; j < dim; j++) {
                printf("Element [%d][%d] : ", i + 1, j + 1);
                scanf("%d", &k[i][j]);
                k[i][j] = (k[i][j] % ALPHABET_SIZE + ALPHABET_SIZE) %
ALPHABET_SIZE;
            }
        }

        is_valid = is_invertible(k, dim);
        if (!is_valid) {
            printf("\nLa matrice n'est pas inversible modulo %d
(determinant = 0 ou non premier avec %d).\n", ALPHABET_SIZE,
ALPHABET_SIZE);
            printf("Veuillez entrer une nouvelle matrice.\n");
        }
    } while (!is_valid);

    printf("\nMatrice valide ! Calcul de l'inverse...\n");
    int** k_inv = inverse_matrix(k, dim);
    if (k_inv == NULL) {
        for (int i = 0; i < dim; i++) free(k[i]);
        free(k);
        return 1;
    }

    FILE* historique = fopen("historique.txt", "a");
    if (historique == NULL) {
        printf("IMPOSSIBLE D'OUVRIR LE FICHIER historique.txt\n");
        for (int i = 0; i < dim; i++) {
            free(k[i]);
            free(k_inv[i]);
        }
        free(k);
        free(k_inv);
        return 1;
    }

    while (1) {
        printf("\nBonjour %s, Que voulez-vous faire ?\n", nom);
        printf("1 - Chiffrer\n");
        printf("2 - Dechiffrer\n");
        printf("3 - Consulter l'historique\n");
        printf("4 - Sortir\n");
        printf("Votre choix: ");

        int choix;
        scanf("%d", &choix);
        getchar();

        switch (choix) {
            case 1: {
                char message[100];
                printf("Entrez le message a chiffrer : ");
                fgets(message, sizeof(message), stdin);
                message[strcspn(message, "\n")] = '\0';
```

```
                char message_chiffre[100];
                strcpy(message_chiffre, message);
                hill_encrypt(message_chiffre, k, dim);
                printf("Le message chiffre est: %s\n", message_chiffre);

                fprintf(historique, "%s/%s/%s\n", message,
message_chiffre, "");
                fflush(historique);
                break;
            }
            case 2: {
                char message[100];
                printf("Entrer le message a decrypter: ");
                fgets(message, sizeof(message), stdin);
                message[strcspn(message, "\n")] = '\0';

                char message_dechiffre[100];
                strcpy(message_dechiffre, message);
                hill_decrypt(message_dechiffre, k_inv, dim);

                printf("Le message dechiffre est: %s\n",
message_dechiffre);

                fprintf(historique, "%s/%s/%s\n", message, "",
message_dechiffre);
                fflush(historique);
                break;
            }
            case 3: {
                fclose(historique);
                historique = fopen("historique.txt", "r");
                if (historique == NULL) {
                    printf("IMPOSSIBLE D'OUVRIR LE FICHIER
historique.txt\n");
                    break;
                }

                printf("\nHistorique:\n");
                char ligne[300];
                while (fgets(ligne, sizeof(ligne), historique)) {
                    printf("%s", ligne);
                }
                printf("\n");

                fclose(historique);
                historique = fopen("historique.txt", "a");
                break;
            }
            case 4:
                fclose(historique);
                for (int i = 0; i < dim; i++) {
                    free(k[i]);
                    free(k_inv[i]);
                }
                free(k);
                free(k_inv);
                return 0;
```

```
            default:
                printf("Choix invalide\n");
        }
    }
    return 0;
}
```