Rapport du projet

DATE : 21/01/2024

Bouda Hamza
Encadré par : Mr. Naji

➢ Introduction

Le projet FreeCell Game est une implémentation du populaire jeu de cartes FreeCell en langage C. Le jeu FreeCell est un solitaire qui utilise un ensemble standard de 52 cartes et se joue sur un tableau de huit colonnes, quatre cellules libres (FreeCells) et quatre piles de fondation. L'objectif du jeu est de déplacer toutes les cartes vers les piles de fondation dans l'ordre croissant, en respectant les règles spécifiques au FreeCell.

Le programme est conçu pour être exécuté en ligne de commande, offrant une expérience interactive où l'utilisateur peut interagir avec le jeu en effectuant des déplacements de cartes entre les différentes zones du plateau. L'interface utilisateur est simple et conviviale, permettant aux joueurs de choisir la source et la destination de leurs déplacements.

Le projet est structuré de manière à utiliser des structures de données telles que des piles (Pile) pour représenter les différentes zones du jeu, des éléments (Element) pour stocker chaque carte individuellement, et une structure de tableau (Board) pour organiser l'ensemble du plateau de jeu.

Le code utilise également des bibliothèques standard telles que `<stdio.h>`, `<stdlib.h>`, `<string.h>`, `<time.h>`, et `<windows.h>` (pour les fonctionnalités spécifiques à Windows) afin de gérer l'entrée/sortie, la génération de nombres aléatoires, et la gestion des couleurs dans la console.

Le jeu propose une mécanique de déplacement de cartes robuste, basée sur des règles logiques, avec des fonctionnalités telles que le mélange des cartes au début du jeu, la validation des mouvements, la possibilité de redémarrer la partie, et une vérification de la victoire lorsque toutes les cartes sont correctement empilées sur les piles de fondation.

Dans ce rapport, nous explorerons en détail la structure du code, les fonctions clés du programme, les choix de conception, ainsi que les fonctionnalités implémentées .

➢ Structures de données

Le projet FreeCell Game utilise plusieurs structures de données pour représenter efficacement le plateau de jeu, les cartes, les piles, et les éléments individuels. Ces structures sont cruciales pour la manipulation des cartes, le suivi de l'état du jeu et la gestion des mouvements des joueurs.

- **Carte :**

- La structure représente une carte individuelle avec deux attributs : la valeur de la carte et le type de la carte .
- La valeur peut être un entier de 1 à 13, représentant les cartes de 1 (As) à 13 (Roi).

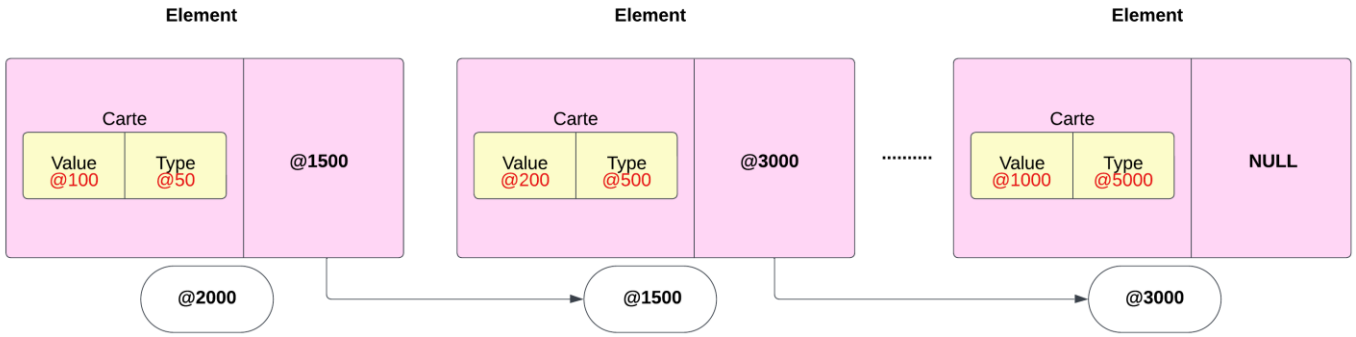
- Le type est une chaîne de caractères (`C`, `P`, `D`, `T`) représentant respectivement les cœurs, les piques, les carreaux et les trèfles.

```c
typedef struct {
    int value;
    char *type;
} Carte;
```

- **Element :**

- La structure est utilisée pour représenter chaque élément d'une pile. Chaque élément contient une carte et un pointeur vers l'élément suivant dans la pile.

```c
typedef struct element {
    Carte *card;
    struct element *next;
} Element;
```



- **Pile :**

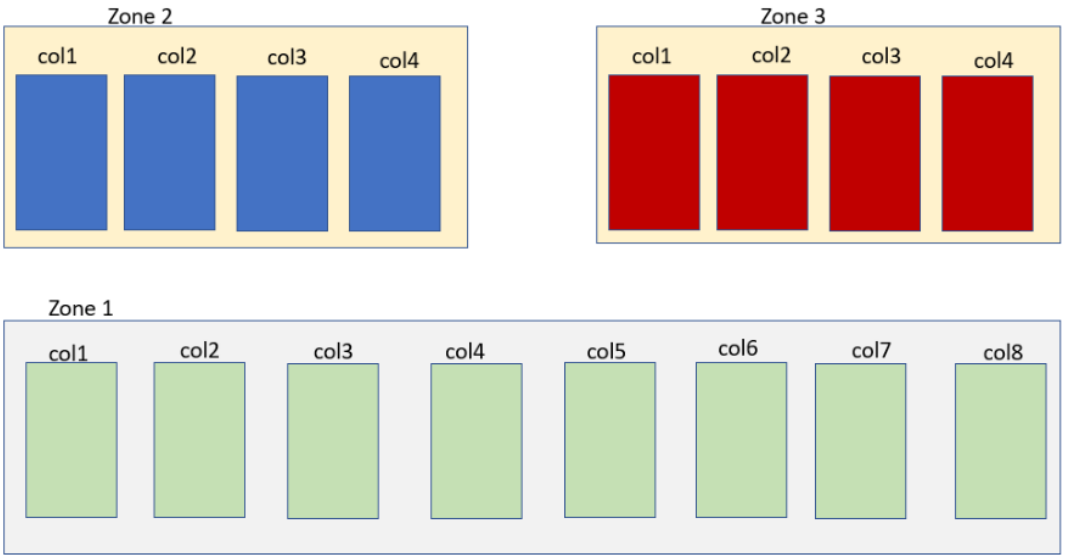
- La structure est utilisée pour représenter une pile de cartes(colonne). Elle contient un pointeur vers le sommet de la pile et la taille actuelle de la pile .

```c
typedef struct {
    Element *top;
    int size;
} Pile;
```

- **Board :**

- La structure représente l'ensemble du plateau de jeu et contient trois tableaux de piles (`zone1`, `zone2`, `zone3`), représentant respectivement les colonnes principales, les cellules libres (FreeCells), et les piles de fondation.

```c
typedef struct {
    Pile *zone1[8];
    Pile *zone2[4];
    Pile *zone3[4];
} Board;
```



Ces structures de données sont utilisées de manière efficace pour stocker et manipuler les cartes tout au long du jeu. Les piles permettent d'implémenter les règles spécifiques au FreeCell, tandis que la structure du plateau de jeu organise l'ensemble de l'environnement de jeu. Ces structures offrent une base solide pour la logique du jeu et facilitent la gestion des déplacements de cartes entre différentes zones du plateau.

➢ Fonctions pour Manipuler les Piles :

Le projet implémente un ensemble de fonctions dédiées à la manipulation des piles. Ces fonctions permettent d'effectuer des opérations telles que la création d'une pile, l'ajout et la suppression de cartes dans une pile, la vérification de l'état de la pile, et la consultation de la carte située au sommet de la pile.

- **Création d'une Pile `createPile` :**

- La fonction `createPile` alloue dynamiquement de la mémoire pour une nouvelle pile, initialise le sommet à NULL et la taille à 0, puis renvoie la pile nouvellement créée.

```c
Pile *createPile() {
    Pile *newPile = (Pile *)malloc(sizeof(Pile));
    newPile->top = NULL;
    newPile->size = 0;
    return newPile;
}
```

- **Empiler une Carte `pushCard` :**

- La fonction `pushCard` prend une pile et une carte en paramètre, crée un nouvel élément contenant la carte, place cet élément au sommet de la pile, puis ajuste la taille de la pile.

```
void pushCard(Pile *pile, Carte *card) {
    Element *newElement = (Element *)malloc(sizeof(Element));
    newElement->card = card;
    newElement->next = pile->top;
    pile->top = newElement;
    pile->size++;
}
```

- **Dépiler une Carte `popCard` :**

- La fonction `popCard` retire la carte située au sommet de la pile, libère la mémoire de l'élément correspondant, ajuste le sommet de la pile et la taille, puis renvoie la carte retirée.

```
Carte *popCard(Pile *pile) {
    if (pile->size == 0) {
        return NULL;  // Pile is empty
    }
    Element *topElement = pile->top;
    Carte *topCard = topElement->card;
    pile->top = topElement->next;
    free(topElement);
    pile->size--;
    return topCard;
}
```

- **Vérifier si une Pile est Vide `isEmpty` :**

- La fonction `isEmpty` prend une pile en paramètre et vérifie si la pile est vide en examinant si le sommet est NULL.

```
int isEmpty(Pile *pile) {
    return (pile == NULL || pile->top == NULL);
}
```

- **Consulter la Carte au Sommet d'une Pile `peekCard` :**

- La fonction `peekCard` retourne la carte située au sommet de la pile sans la dépiler. Elle gère également le cas où la pile est vide.

```
Carte *peekCard(Pile *pile) {
    if (pile->size == 0) {
        return NULL;  // Pile is empty
    }
    return pile->top->card;
}
```

- **Consulter la Carte à un Niveau Spécifique d'une Pile `peekCardAtLevel` :**

- La fonction `peekCardAtLevel` permet de consulter la carte à un niveau spécifique de la pile en utilisant un déplacement itératif à travers la pile.

```
Carte *peekCardAtLevel(Pile *pile, int level) {
    Element *current = pile->top;
    for (int i = 0; i < level; i++) {
        current = current->next;
    }
    return current->card;
}
```

Ces fonctions forment l'infrastructure nécessaire pour gérer les piles de cartes dans le jeu FreeCell, permettant ainsi le déplacement correct des cartes entre les différentes zones du plateau de jeu.

➢ Affichage

- **Fonction `effacerEcran` :**

Cette fonction est cruciale pour maintenir une interface utilisateur propre et lisible. Elle est utilisée pour effacer l'écran à chaque mise à jour du jeu, permettant ainsi d'éviter un encombrement d'informations et d'assurer une expérience de jeu claire. L'utilisation de la fonction `system("cls")` permet d'effacer l'écran du terminal, offrant une mise à jour visuelle nette à chaque étape du jeu.

```
void effacerEcran() {
    system("cls");
}
```

- **Fonction `color` :**

La fonction `color` est utilisée pour changer la couleur du texte affiché dans le terminal en utilisant des codes de couleurs ANSI. Elle prend deux paramètres, `t` pour la couleur du texte et `f` pour la couleur de fond. La fonction utilise `SetConsoleTextAttribute` pour appliquer les couleurs sur Windows.

```
void color(int t, int f){
    HANDLE H = GetStdHandle(STD_OUTPUT_HANDLE);
    SetConsoleTextAttribute(H, f * 16 + t);
}
```

- **Fonction 'afficherErreurAvecPause' :**

La fonction 'afficherErreurAvecPause' est une fonction qui affiche un message d'erreur en couleur rouge, met en pause l'exécution du programme pendant une durée spécifiée, puis réinitialise la couleur du texte à sa valeur par défaut.

```
#ifdef _WIN32
#define PAUSE(milliseconds) Sleep(milliseconds)
#endif
void afficherErreurAvecPause(const char *message, int dureeEnMillisecondes) {
    printf(ANSI_COLOR_RED "Erreur: %s\n" ANSI_COLOR_RESET, message);
    PAUSE(dureeEnMillisecondes);
    color(15,18);
}
```

La fonction color(15, 18); est appelée pour réinitialiser la couleur du texte à sa valeur par défaut (texte blanc sur fond noir).
Ceci garantit que les prochaines sorties seront affichées avec les couleurs normales du terminal.

Ces fonctions d'affichage sont essentielles pour améliorer l'expérience utilisateur du jeu FreeCell en fournissant un affichage clair, des messages d'erreur visuellement distincts et en garantissant une interface utilisateur propre et mise à jour.

➢ Fonctions pour manipuler les cartes

- **Fonction 'printCardSymbol' :**

- Cette fonction prend en paramètre le type d'une carte (pique, trèfle, cœur, carreau) et affiche le symbole associé à l'écran. Elle utilise des caractères Unicode pour représenter les symboles des cartes.

```
void printCardSymbol(const char *type) {
    if (strcmp(type, "P") == 0) {
        printf("%s ", CLUB);
    } else if (strcmp(type, "D") == 0) {
        printf("%s ", DIAMOND);
    } else if (strcmp(type, "C") == 0) {
        printf("%s ", HEART);
    } else if (strcmp(type, "T") == 0) {
        printf("%s ", SPADE);
    }
}
```

```
#define HEART "\u2665"
#define DIAMOND "\u2666"
#define CLUB "\u2663"
#define SPADE "\u2660"
```

- **Fonction `printCard` :**

- Affiche une carte spécifique à l'écran en utilisant des codes ANSI pour la couleur du texte et de l'arrière-plan.
- La couleur de la carte dépend du type de carte (cœur et carreau en rouge, pique et trèfle en noir).
- Affiche la valeur de la carte avec le symbole correspondant.

```c
void printCard(Carte *card) {
    if (card == NULL) {
        printf("Carte invalide\n");
        return;
    }
    // Definir les codes ANSI pour les couleurs
    const char *resetColor = "\033[0m";
    const char *whiteBackground = "\033[47m";
    const char *redText = "\033[91m";
    const char *blackText = "\033[30m";
    // Choisir la couleur en fonction du type de carte
    const char *cardColor;
    if (card->type[0] == 'C' || card->type[0] == 'D') {
        cardColor = redText; // Rouge pour c ur et carreau
    } else {
        cardColor = blackText; // Noir pour pique et tr fle
    }
    // Imprimer la carte avec le symbole d fini
    switch (card->value) {
        case 1:
            printf("%s%s A  ", whiteBackground, cardColor);
            printCardSymbol(card->type);
            printf("%s%s", resetColor, resetColor);
            color(15, 18);
            printf(" ");
            break;
        case 10 :
            printf("%s%s 10 ", whiteBackground, cardColor);
            printCardSymbol(card->type);
            printf("%s%s", resetColor, resetColor);
            color(15, 18);
            printf(" ");
            break;
        case 11:
            printf("%s%s J  ", whiteBackground, cardColor);
            printCardSymbol(card->type);
            printf("%s%s", resetColor, resetColor);
            color(15, 18);
            printf(" ");
            break;
        case 12:
            printf("%s%s Q  ", whiteBackground, cardColor);
            printCardSymbol(card->type);
            printf("%s%s", resetColor, resetColor);
            color(15, 18);
            printf(" ");
            break;
        case 13:
            printf("%s%s K  ", whiteBackground, cardColor);
            printCardSymbol(card->type);
            printf("%s%s", resetColor, resetColor);
             color(15, 18);
            printf(" ");
            break;
        default:
            printf("%s%s %d  ", whiteBackground, cardColor, card->value);
            printCardSymbol(card->type);
            printf("%s%s", resetColor, resetColor);
            color(15, 18);
            printf(" ");
            break;
    }
}
```

- **Fonction `initializeBoard`:**

- Initialise la structure du plateau de jeu en allouant de la mémoire pour chaque pile dans les zones de jeu (zone1, zone2, zone3).
- Retourne un pointeur vers la structure du plateau de jeu nouvellement créée.

```
Board *initializeBoard() {
    Board *gameBoard = (Board *)malloc(sizeof(Board));
    for (int i = 0; i < 8; i++) {
        gameBoard->zone1[i] = createPile();
    }
    for (int i = 0; i < 4; i++) {
        gameBoard->zone2[i] = createPile();
        gameBoard->zone3[i] = createPile();
    }
    return gameBoard;
}
```

- **Fonction `createCard` :**

- Crée une nouvelle carte avec une valeur spécifiée et un type (cœur, pique, carreau, trèfle).
- Alloue dynamiquement de la mémoire pour la carte et retourne un pointeur vers la carte créée.

```
Carte *createCard(int value, const char *type) {
    Carte *newCard = (Carte *)malloc(sizeof(Carte));
    newCard->value = value;
    newCard->type = strdup(type);
    return newCard;
}
```

- **Fonction `remplirPiles` :**

- Remplit les piles du plateau de jeu avec des cartes mélangées.
- Utilise un tableau temporaire pour stocker toutes les cartes, puis les mélange avant de les distribuer dans les piles.

```
void remplirPiles(Board *gameBoard) {
    srand((unsigned int)time(NULL)); // Initialisation du generateur de nombres aleatoires
    int cardCount = 0;
    // Creer et initialiser les valeurs et types pour chaque couleur
    const char *types[4] = { "C", "P", "D", "T" }; // C pour coeur P pour pique D pour Carreau T pour trefle
    int values[13] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13 };
    // Creer un tableau pour stocker toutes les cartes
    Carte *allCards[52];
    int index = 0;
    // Ajouter toutes les cartes dans le tableau
    for (int suit = 0; suit < 4; suit++) {
        for (int value = 0; value < 13; value++) {
            allCards[index++] = createCard(values[value], types[suit]);
        }
    }
    /* Melanger les cartes dans le tableau en utilisant l'algorithme de melange */ /**Fisher-Yates**/
    for (int i = 51; i > 0; i--) {
        int j = rand() % (i + 1);
        Carte *temp = allCards[i];
        allCards[i] = allCards[j];
        allCards[j] = temp;
    }
    // Ajouter les cartes melangees aux piles du plateau de jeu
    index = 0;
    for (int col = 0; col < 8; col++) {
        int cardsToAdd = (col < 4) ? 7 : 6; // Determiner le nombre de cartes a ajouter a chaque pile
        for (int i = 0; i < cardsToAdd; i++) {
            pushCard(gameBoard->zone1[col], allCards[index++]);
            cardCount++;
        }
    }
}
```

- **Fonction `printVertically` :**

- Affiche les cartes d'une pile verticalement à l'écran.
- Utilise un tableau temporaire pour stocker les cartes, puis les affiche de bas en haut.

```
void printVertically(Pile *pile) {
    Element *current = pile->top;
    // Stocker les elements dans un tableau temporaire
    Carte *cards[12];
    int count = 0;
    while (current != NULL && count < 12) {
        cards[count++] = current->card;
        current = current->next;
    }
    // Afficher chaque carte de la pile verticalement dans l'ordre inverse
    for (int i = count - 1; i >= 0; i--) {
            color(15, 18);
        printCard(cards[i]);
        printf("\n");
    }
    printf(" ");
}
```

- **Fonction `printBoard` :**

- Affiche l'état actuel du plateau de jeu, y compris les cartes dans les différentes zones (zone1, zone2, zone3).
- Utilise les fonctions `printCard` et `printVertically` pour afficher les cartes.

Ces fonctions constituent une partie cruciale du projet FreeCell en gérant l'aspect visuel du jeu et en permettant aux joueurs d'interagir avec les cartes sur le plateau de jeu.

➢ Fonctions de déplacement des cartes

- **Fonction `isMoveValid` :**

- Vérifie si le déplacement d'une carte depuis une source vers une destination est valide.
- La validité dépend de deux critères principaux : la valeur de la carte source doit être inférieure d'une unité à la valeur de la carte destination, et les couleurs des cartes doivent être opposées (rouge et noir).

```
int isMoveValid(Carte *sourceCard, Carte *destinationCard) {
    if (sourceCard == NULL && destinationCard == NULL) {
        return 0;
    }
    // V□rifier si la carte destination est de rang imm□diatement sup□rieur
    if (sourceCard->value == destinationCard->value - 1) {
        // V□rifier si les couleurs sont oppos□es
        if (sourceCard->type[0] == 'C' && (destinationCard->type[0] == 'P' || destinationCard->type[0] == 'T'))
            return 1;
        else if (sourceCard->type[0] == 'P' && (destinationCard->type[0] == 'C' || destinationCard->type[0] == 'D'))
            return 1;
        else if (sourceCard->type[0] == 'T' && (destinationCard->type[0] == 'C' || destinationCard->type[0] == 'D'))
            return 1;
        else if (sourceCard->type[0] == 'D' && (destinationCard->type[0] == 'T' || destinationCard->type[0] == 'P'))
            return 1;
    }
    return 0;
}
```

- **Fonction `deplacerCarte` :**

- Déplace la carte du dessus de la pile source vers la pile destination.
- Utilise les fonctions `popCard` et `pushCard` pour effectuer le déplacement.

```
void deplacerCarte(Pile *source, Pile *destination) {
    Carte *carteDeplacee = popCard(source);
    if (carteDeplacee != NULL) {
        pushCard(destination, carteDeplacee);
        // Ajouter des espaces vides dans la colonne source
    for (int i = 0; i < source->size; i++) {
        printf("          ");
    }
    }
}
```

- **Fonction `deplacerCarteEntreZones` :**

- Déplace une carte d'une zone source vers une zone destination spécifiée.
- Vérifie la validité du déplacement en utilisant `isMoveValid`.
- Gère les déplacements entre les différentes zones du plateau de jeu (zone1, zone2, zone3).

- **Fonction `deplacerZone1VersZone2` :**

- Déplace une carte de la zone1 vers la zone2 en utilisant `deplacerCarteEntreZones`.
- Appelée lorsque le joueur veut déplacer une carte de la colonne principale vers une FreeCell.

- **Fonction `deplacerZone1VersZone1` :**

- Déplace une carte de la zone1 vers la zone1 en utilisant `deplacerCarte`.
- Appelée lorsque le joueur veut déplacer une carte de la colonne principale vers une autre colonne principale.

- **Fonction `deplacerZone1VersZone3`:**

- Déplace une carte de la zone1 vers la zone3 (fondation) en utilisant `deplacerCarte`.
- Appelée lorsque le joueur veut déplacer une carte de la colonne principale vers la fondation.

- **Fonction `deplacerZone2VersZone3` :**

- Déplace une carte de la zone2 vers la zone3 (fondation) en utilisant `deplacerCarte`.
- Appelée lorsque le joueur veut déplacer une carte d'une FreeCell vers la fondation.

- **Fonction `deplacerZone2VersZone1` :**

- Déplace une carte de la zone2 vers la zone1 en utilisant `deplacerCarte`.
- Appelée lorsque le joueur veut déplacer une carte d'une FreeCell vers la colonne principale.

Ces fonctions forment la logique de déplacement des cartes dans le jeu FreeCell, permettant aux joueurs d'interagir avec les cartes sur le plateau de jeu en respectant les règles du jeu.

➢ Fonctions de jeu

- **Fonction `playGame` :**

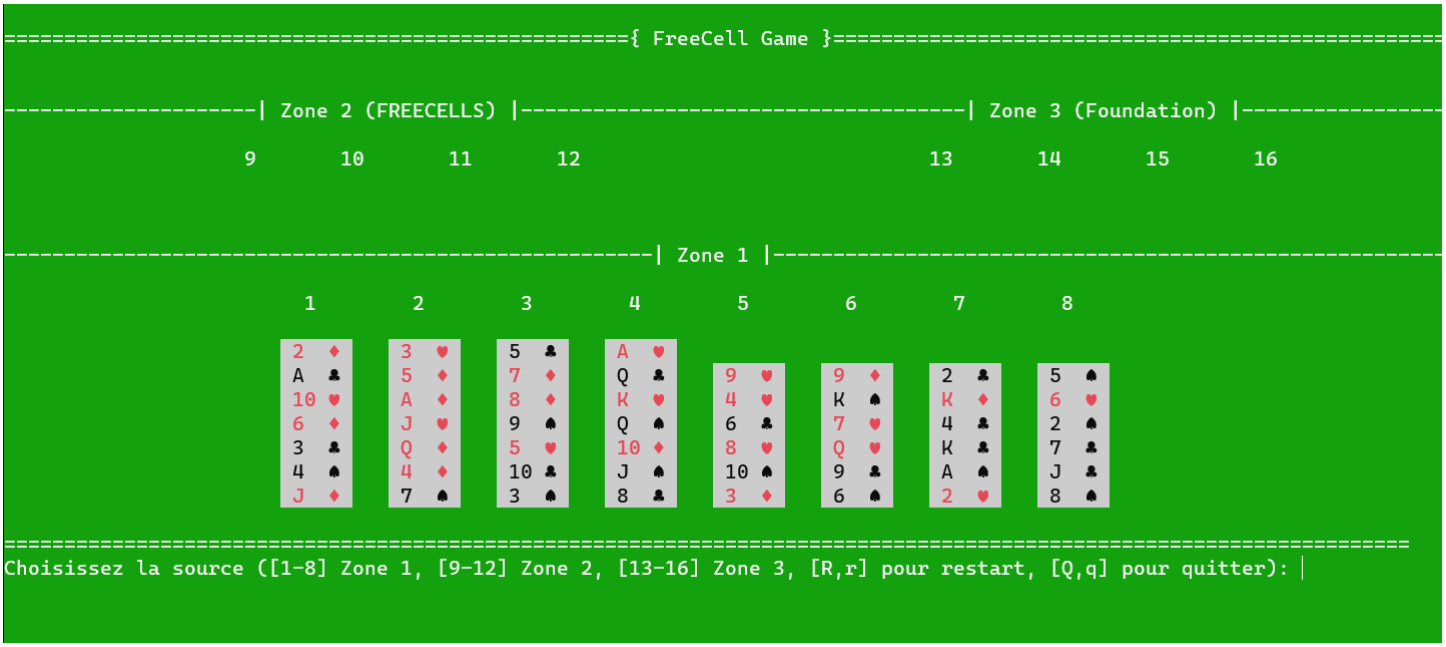
Cette fonction représente le déroulement du jeu. Elle initialise le plateau de jeu, remplit les piles avec des cartes, puis commence une boucle principale où le joueur effectuent leurs mouvements jusqu'à ce que la condition de victoire soit remplie ou que le joueur choisisse de quitter.

✓ Initialisation du plateau de jeu :
- Appelle `initializeBoard` pour créer et initialiser le plateau de jeu.
- Appelle `remplirPiles` pour remplir les piles avec des cartes mélangées.

✓ Boucle principale du jeu :
- Affiche le plateau de jeu avec `printBoard`.
- Demande au joueur de choisir une source et une destination pour son mouvement.
- Effectue le mouvement en appelant les fonctions appropriées (ex. `deplacerZone1VersZone1`, `deplacerZone1VersZone2`, etc.).
- Répète la boucle jusqu'à ce que la condition de victoire (`youWin`) soit remplie ou que le joueur choisisse de quitter.

✓ Condition de victoire :
- Vérifie si le joueur a gagné en appelant la fonction `youWin`.
- Si la victoire est atteinte, affiche un message de félicitations avec `printWinMessage`.

✓ Recommencer la partie :
- Si le joueur choisit de recommencer, appelle `restartGame` pour réinitialiser le plateau de jeu.



- **Fonction `youWin` :**

Cette fonction vérifie si le joueur a gagné en vérifiant si chaque pile de fondation contient un Roi (valeur 13) comme carte supérieure.

```
int youWin(Board *gameBoard) {
    for (int i = 0; i < 4; i++) {
        Pile *foundationPile = gameBoard->zone3[i];
        // Verifier si la pile est vide
        if (foundationPile == NULL || isEmpty(foundationPile)) {
            return 0;
        }
        // Verifier si la carte supérieure est un Roi (valeur = 13)
        if (peekCard(foundationPile)->value != 13) {
            return 0;
        }
    }
    return 1; // Toutes les piles de fondation ont un Roi en haut
}
```

- **Fonction `restartGame` :**

Cette fonction réinitialise le plateau de jeu en libérant la mémoire des cartes existantes et en remplissant à nouveau les piles avec des cartes mélangées.

```
void restartGame(Board *gameBoard) {
    // Liberer la memoire des cartes existantes dans le plateau de jeu et reinitialiser les piles
    for (int i = 0; i < 8; i++) {
        while (gameBoard->zone1[i]->size > 0) {
            popCard(gameBoard->zone1[i]);
        }
    }
    for (int i = 0; i < 4; i++) {
        while (gameBoard->zone2[i]->size > 0) {
            popCard(gameBoard->zone2[i]);
        }
        while (gameBoard->zone3[i]->size > 0) {
            popCard(gameBoard->zone3[i]);
        }
    }

    // Remplir les piles avec des cartes uniques
    remplirPiles(gameBoard);
}
```

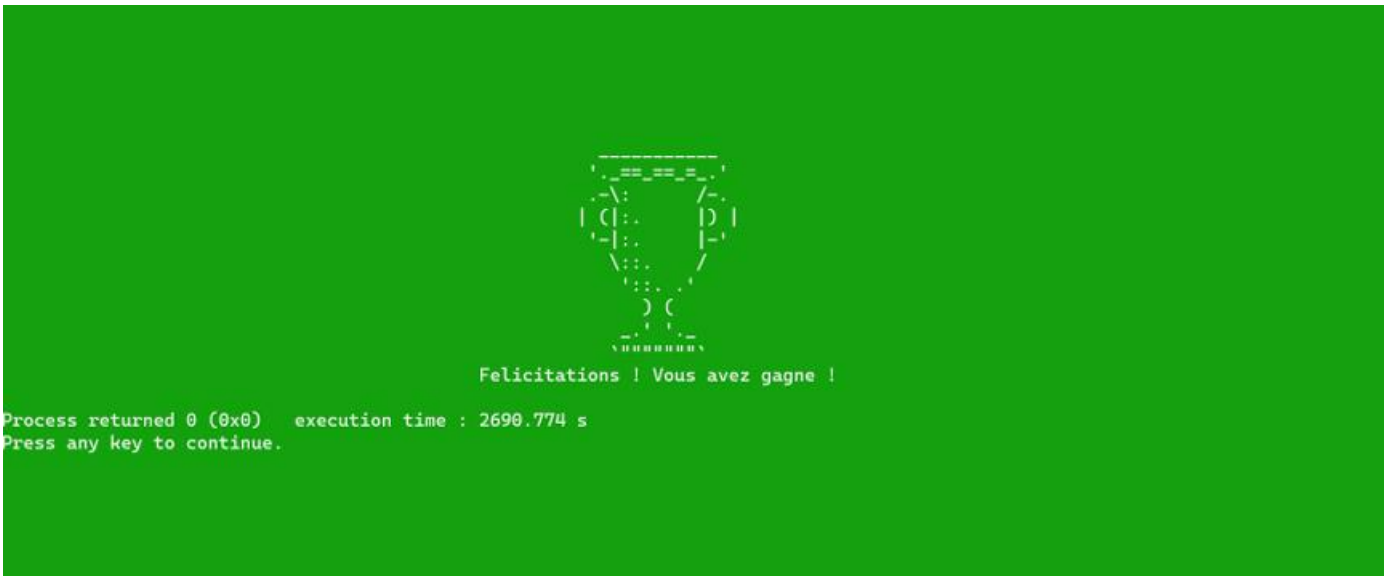
- **Fonction `main`**

Dans le main, on initialise la sortie de la console pour prendre en charge les caractères Unicode, définit la couleur de la console, et lance le jeu en appelant playGame.

```
int main() {
    SetConsoleOutputCP(65001);
            system("color 2F");
            color(15, 2);
            playGame();
    return 0;
}
```

- **Fonction `printWinMessage` :**

Affiche un message de félicitations lorsque le joueur gagne la partie.



➢ Conclusion

Le projet visant à développer un jeu FreeCell en langage C a été une expérience enrichissante. La réalisation de ce jeu a permis de mettre en pratique divers concepts de programmation, notamment la manipulation de structures de données, la gestion des entrées utilisateur, et la mise en œuvre de règles de jeu complexes. La modularité du code, avec l'utilisation de structures telles que `Carte`, `Element`, `Pile`, et `Board`, a facilité la gestion du jeu .

Possibles Améliorations et État de Blocage :

Malgré les réalisations, quelques points d'amélioration et des états de blocage ont été identifiés.

- Amélioration de l'Interface Utilisateur (UI) :

L'interface utilisateur pourrait être plus conviviale en intégrant des fonctionnalités telles que des indications visuelles plus claires, des animations de cartes, ou un système d'aide interactif.

- Extensions du Jeu :

L'ajout de fonctionnalités supplémentaires, telles que des niveaux de difficulté, fin de jeu dans l'état de blocage, pourrait rendre le jeu plus attrayant et diversifié.

En conclusion, ce projet a fourni une excellente occasion d'appliquer les connaissances en programmation dans un contexte concret.