Rapport de développement

- Projet Crazy Circus -

Projet : Développer un jeu de logique en C incluant la génération combinatoire de cartes et la gestion de podiums.

Par

Walim Amor-Chelihi

Equipe n°2

Institut Universitaire Technologique de Paris (Rives de Seine)

S1 - Période B - Décembre 2025

Table des matières

Introduction

1.1 Présentation du projet

Ce projet s'inscrit dans le cadre de la **SAE S1.02 du BUT Informatique**. Il consiste à réaliser une application en langage **C** permettant de gérer une **partie complète** du jeu *Crazy Circus*, conçu par Dominique Ehrhard.

Le jeu met en scène plusieurs **animaux** répartis sur deux podiums (bleu et rouge). Les animaux sont empilés sur chaque podium et peuvent être déplacés en exécutant des **ordres précis**. L'objectif d'un tour de jeu est de trouver une **séquence d'ordres** permettant de passer d'une situation initiale à une situation objectif.

Le programme permet à **plusieurs joueurs** de s'affronter. Chaque joueur propose, à tour de rôle, une séquence d'ordres. Le premier joueur proposant une **séquence correcte** remporte le tour. La partie se termine lorsque toutes les **cartes objectif** ont été utilisées, et le joueur ayant remporté le plus de tours gagne la partie.

1.2 Fonctionnalités principales

Le programme développé propose les fonctionnalités suivantes :

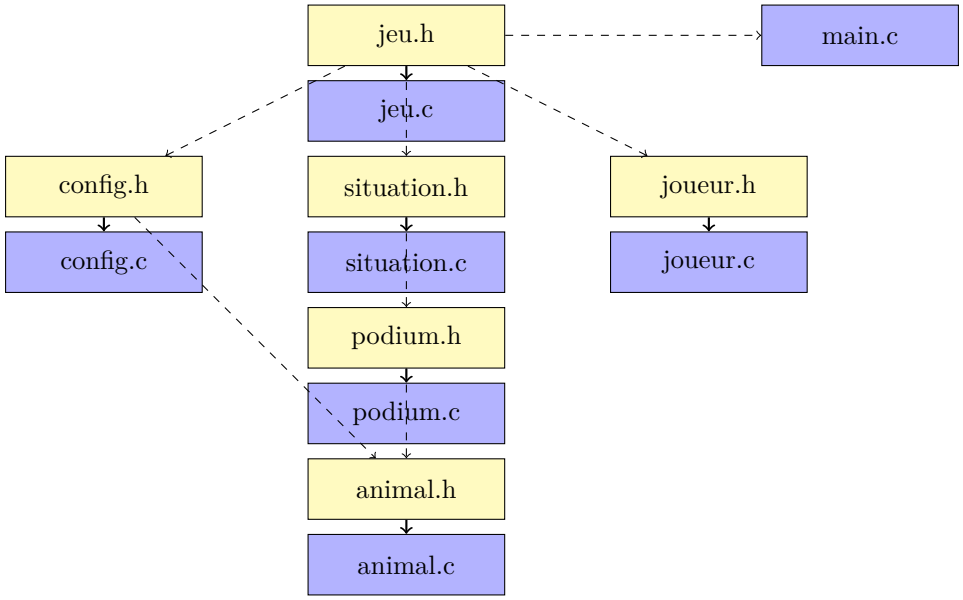
- **Lecture et validation de la configuration** du jeu à partir du fichier `crazy.cfg`, incluant la liste des animaux et les ordres autorisés ;
- **Gestion de plusieurs joueurs**, dont les identités sont fournies en paramètre de la ligne de commande, avec vérification de leur validité (nombre minimum et noms distincts) ;
- **Affichage des ordres disponibles** et de la **situation de jeu** (situation initiale et situation objectif) en respectant strictement le format imposé ;
- **Exécution des ordres du jeu** (KI, LO, SO, NI, MA) et simulation d'une séquence d'ordres proposée par un joueur ;
- **Vérification de la validité des séquences** saisies par les joueurs et gestion des erreurs (ordre inexistant, séquence incorrecte, joueur non autorisé), avec affichage de messages informatifs ;
- **Gestion complète d'une partie**, incluant le tirage aléatoire des cartes objectif, l'enchaînement des tours et l'attribution des points ;
- **Affichage des scores** et du classement final des joueurs en fin de partie.

Graphe de dépendance

Le graphe de dépendance ci-dessous représente l'organisation des fichiers du projet :

— une flèche pleine indique qu'un fichier `.c` implémente un fichier `.h` ;

— une flèche en pointillés représente une dépendance créée par une instruction `#include`.



Le composant `animal` définit la structure `Animal` ainsi que les fonctions de base permettant d'initialiser, d'afficher et de comparer les animaux. Ce composant constitue un élément fondamental du projet, car les animaux sont utilisés dans plusieurs autres composants. Centraliser leur définition dans un composant unique garantit la cohérence des données et facilite les évolutions futures.

Le composant `podium` représente un podium sous la forme d'une pile d'animaux. Il fournit toutes les opérations nécessaires à la manipulation de cette pile (empiler, dépiler, accéder au sommet ou à la base). Il dépend logiquement du composant `animal`, mais reste indépendant des autres composants du jeu.

Le composant `situation` regroupe deux podiums (bleu et rouge) et représente un état complet du jeu. Il contient les fonctions permettant de copier, comparer et afficher une situation, ainsi que l'exécution des différents ordres du jeu (KI, LO, SO, NI et MA). Cette organisation permet d'isoler la logique liée aux règles du jeu dans un composant dédié.

Le composant `config` est chargé de la lecture du fichier `crazy.cfg`. Il stocke la liste des animaux et des ordres autorisés pour la partie. Ce composant permet de séparer clairement la configuration du jeu de sa logique, rendant le programme plus flexible et plus lisible.

Le composant `joueur` gère les informations relatives aux joueurs (nom, score, possibilité de jouer). Il fournit les fonctions nécessaires à l'initialisation des joueurs, à la recherche d'un joueur par son nom et à l'affichage des scores en fin de partie. Ce composant est indépendant de la logique interne du jeu.

Le composant `jeu` centralise la logique globale de la partie. Il regroupe la configuration, les joueurs, les situations de jeu et les cartes objectifs. Ce composant coordonne le déroulement des tours, le traitement des saisies des joueurs et l'exécution des séquences d'ordres, sans gérer directement les détails internes des autres composants.

Le fichier `main.c` constitue le point d'entrée du programme. Il se limite à l'initialisation du jeu, à la récupération des paramètres de la ligne de commande et au lancement de la partie. La logique du jeu est volontairement absente de ce fichier, ce qui permet de conserver un point d'entrée simple et lisible.

Tests Unitaires

3.1 Résultats des tests unitaires

Fonction testée	Objectif du test	Résultat
`testAnimal`	Vérifier l'initialisation et la comparaison de deux animaux	Validé
`testPodium`	Vérifier les opérations sur un podium (empiler, dépiler, sommet, base)	Validé
`testMonterBase`	Vérifier le déplacement de l'animal situé à la base vers le sommet du podium	Validé
`testSituation`	Vérifier l'initialisation d'une situation et la gestion des deux podiums	Validé
`testOrdreKI`	Vérifier l'exécution de l'ordre KI (bleu vers rouge)	Validé
`testOrdreSO`	Vérifier l'exécution de l'ordre SO (échange des sommets)	Validé
`testConfig`	Vérifier la lecture et la validation du fichier `crazy.cfg`	Validé

3.2 Code source du fichier de test (test.c)

```c
#include "animal.h"
#include "podium.h"
#include "situation.h"
#include "config.h"
#include <stdio.h>
#include <assert.h>

void testAnimal() {
    Animal a1, a2;
    initAnimal(&a1, "LION");
    initAnimal(&a2, "LION");
    assert(animauxEgaux(&a1, &a2));

    initAnimal(&a2, "OURS");
    assert(!animauxEgaux(&a1, &a2));
    printf("Va | Valide - Test Animal \n\n");
}

void testPodium() {
    Podium p;
    initPodium(&p);
    assert(estVidePodium(&p));

    Animal lion, ours;
    initAnimal(&lion, "LION");
```

```
    initAnimal(&ours, "OURS");

    empilerPodium(&p, &lion);
    assert(!estVidePodium(&p));
    assert(p.nb == 1);

    empilerPodium(&p, &ours);
    assert(p.nb == 2);

    Animal sommet;
    sommetPodium(&p, &sommet);
    assert(animauxEgaux(&sommet, &ours));

    Animal base;
    basePodium(&p, &base);
    assert(animauxEgaux(&base, &lion));

    Animal depile;
    depilerPodium(&p, &depile);
    assert(animauxEgaux(&depile, &ours));
    assert(p.nb == 1);

    printf("Va | Valide - Test Podium\n\n");
}

void testMonterBase() {
    Podium p;
    initPodium(&p);

    Animal lion, ours, elephant;
    initAnimal(&lion, "LION");
    initAnimal(&ours, "OURS");
    initAnimal(&elephant, "ELEPHANT");

    empilerPodium(&p, &lion);
    empilerPodium(&p, &ours);
    empilerPodium(&p, &elephant);

    monterBasePodium(&p);

    Animal sommet;
    sommetPodium(&p, &sommet);
    assert(animauxEgaux(&sommet, &lion));

    printf("Va | Valide - Test monterBase\n\n");
}

void testSituation() {
    Situation s;
```

```
    initSituation(&s);

    Animal lion, ours;
    initAnimal(&lion, "LION");
    initAnimal(&ours, "OURS");

    empilerPodium(&s.bleu, &lion);
    empilerPodium(&s.rouge, &ours);

    assert(!estVidePodium(&s.bleu));
    assert(!estVidePodium(&s.rouge));

    printf("Va | Valide - Test Situation\n\n");
}

void testOrdreKI() {
    Situation s;
    initSituation(&s);

    Animal lion, ours;
    initAnimal(&lion, "LION");
    initAnimal(&ours, "OURS");

    empilerPodium(&s.bleu, &lion);
    empilerPodium(&s.bleu, &ours);

    executerKI(&s);

    assert(s.bleu.nb == 1);
    assert(s.rouge.nb == 1);

    Animal sommetRouge;
    sommetPodium(&s.rouge, &sommetRouge);
    assert(animauxEgaux(&sommetRouge, &ours));

    printf("Va | Valide - Test ordre KI\n\n");
}

void testOrdreSO() {
    Situation s;
    initSituation(&s);

    Animal lion, ours;
    initAnimal(&lion, "LION");
    initAnimal(&ours, "OURS");

    empilerPodium(&s.bleu, &lion);
    empilerPodium(&s.rouge, &ours);
```

```
    executerSO(&s);

    Animal sommetBleu, sommetRouge;
    sommetPodium(&s.bleu, &sommetBleu);
    sommetPodium(&s.rouge, &sommetRouge);

    assert(animauxEgaux(&sommetBleu, &ours));
    assert(animauxEgaux(&sommetRouge, &lion));

    printf("Va | Valide - Test ordre SO\n\n");
}

void testConfig() {
    Config cfg;

    if (chargerConfig(&cfg)) {
        assert(cfg.nbAnimaux >= 2);
        assert(cfg.nbOrdres >= 3);
        printf("Va | Valide - Test Config\n\n");
    } else {
        printf("X  | ECHEC - Test Config (fichier crazy.cfg manquant ou
        ↪  invalide)\n");
    }
}

int main() {
    printf("\n\n________ TESTS UNITAIRES ________\n\n");
    testAnimal();
    testPodium();
    testMonterBase();
    testSituation();
    testOrdreKI();
    testOrdreSO();
    testConfig();
    printf("\n________ TOUS LES TESTS PASSES AVEC SUCCESS ________\n\n");
    return 0;
}
```

Bilan du projet

4.1 Difficultés rencontrées

J'ai rencontré plusieurs difficultés durant ce projet. La première a été d'organiser correctement les fichiers du programme. Trouver une structure claire demandait une vraie gymnastique mentale, car il fallait comprendre comment chaque partie du code s'articulait avec les autres. J'ai aussi eu du mal à adapter l'algorithme de Heap pour générer automatiquement les cartes : il fallait l'ajuster au fonctionnement des podiums et au nombre d'animaux, ce qui n'était pas évident au début. La configuration dynamique du fichier crazy.cfg a également posé problème, car le programme devait s'adapter au nombre d'animaux et aux ordres disponibles, ce qui demandait une logique précise pour éviter les erreurs. La gestion d'une partie complète n'était pas simple non plus : il y a beaucoup d'éléments qui interagissent, et dès qu'un détail ne fonctionnait plus, il devenait difficile de retrouver l'origine du problème. Enfin, respecter strictement le format d'affichage demandé a été une difficulté supplémentaire, notamment les espaces dynamiques. La quantité de travail pour ce projet était quand même supérieure au projet précédent, et le faire seul était tout de même une difficulté.

4.2 Points réussis

Globalement, j'ai réussi à gérer correctement une partie du début à la fin. Toutes les fonctionnalités principales du jeu fonctionnent bien, que ce soit la lecture de la configuration, la génération des cartes, la gestion des joueurs ou l'application des ordres. La structure du projet me paraît solide et cohérente, ce qui rend le code lisible et assez facile à maintenir. Malgré les difficultés rencontrées, le résultat final est stable et conforme aux attentes dans le moindre détails.

4.3 Améliorations possibles

Même si le programme fonctionne correctement, plusieurs amélioroorations seraient possibles. La gestion des erreurs pourrait être rendue plus précise afin de mieux identifier l'origine des problèmes lors de l'exécution.

Une autre amélioration possible serait de mieux exploiter la bibliothèque standard du C pour simplifier certains traitements et réduire la quantité de code manuel. Enfin, ajouter plus de tests unitaires permettrait de vérifier automatiquement le comportement du programme dans différents cas

Code source

5.1 Code - animal.c

```c
#include "animal.h"
#include <stdio.h>
#include <string.h>
#include <assert.h>

void initAnimal(Animal* a, const char* nom) {
    assert(strlen(nom) < MAX_NOM_ANIMAL);
    strcpy(a->nom, nom);
}

void afficheAnimal(const Animal* a) {
    printf("%s", a->nom);
}

int animauxEgaux(const Animal* a1, const Animal* a2) {
    return strcmp(a1->nom, a2->nom) == 0;
}
}
```

5.2 Code - animal.h

```c
/**
 * @file animal.h
 * @brief Gestion des animaux
 */

#pragma once

enum { MAX_NOM_ANIMAL = 20 };

/**
 * @brief Structure représentant un animal
 */

typedef struct {
    char nom[MAX_NOM_ANIMAL]; // Nom de l'animal
} Animal;

/**
 * @brief Initialise un animal avec son nom
 * @param[out] a Pointeur vers l'animal à initialiser
 * @param[in] nom Nom de l'animal
 */
```

```c
void initAnimal(Animal* a, const char* nom);

/**
 * @brief Affiche le nom d'un animal
 * @param[in] a Pointeur vers l'animal
 */

void afficheAnimal(const Animal* a);

/**
 * @brief Compare deux animaux
 * @param[in] a1 Premier animal
 * @param[in] a2 Second animal
 * @return 1 si identiques et 0 sinon
 */

int animauxEgaux(const Animal* a1, const Animal* a2);
```

5.3 Code - config.c

```c
#include "config.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>

#pragma warning (disable : 4996)

const char* const FILE_NAME = "crazy.cfg";

int chargerConfig(Config* cfg) {

    FILE* f = fopen(FILE_NAME, "r");

    if (f == NULL) {
        printf("X  | ERREUR - Fichier non accessible\n");
        return 0;
    }

    cfg->nbAnimaux = 0;
    cfg->nbOrdres = 0;

    enum { BUFFER_SIZE = 10 };
    char buffer[BUFFER_SIZE];

    for (int numLigne = 0; numLigne < 2; numLigne++) {
```

```
    char* line = (char*)calloc(1, 1);
    char* s = fgets(buffer, BUFFER_SIZE, f);
    while (s != NULL) {
        size_t taille = strlen(buffer);
        assert(taille != 0);
        if (buffer[taille - 1] == '\n') {
            buffer[taille - 1] = '\0';
            if (taille > 1) {
                line = (char*)realloc(line, strlen(line) + taille);
                strcat(line, buffer);
            }
            break;
        }
        else {
            line = (char*)realloc(line, strlen(line) + taille + 1);
            strcat(line, buffer);
        }
        s = fgets(buffer, BUFFER_SIZE, f);
    }

    int offset = 0;
    int caracteresLus = 0;
    char mot[MAX_NOM_ANIMAL];

    while (sscanf(line + offset, "%s%n", mot, &caracteresLus) == 1) {
        if (numLigne == 0) {
            if (cfg->nbAnimaux < MAX_ANIMAUX) {
                initAnimal(&cfg->animaux[cfg->nbAnimaux], mot);
                cfg->nbAnimaux++;
            }
        } else {
            if (cfg->nbOrdres < MAX_ORDRES) {
                if (strcmp(mot, "KI") == 0 || strcmp(mot, "LO") == 0 ||
                    strcmp(mot, "SO") == 0 || strcmp(mot, "NI") == 0 ||
                    strcmp(mot, "MA") == 0) {
                    strcpy(cfg->ordres[cfg->nbOrdres], mot);
                    cfg->nbOrdres++;
                }
            }
        }
        offset += caracteresLus;
    }
    free(line);
}

fclose(f);

if (cfg->nbAnimaux < 2) {
    printf("X  | ERREUR - Il faut au moins 2 animaux\n");
```

```
        return 0;
    }
    if (cfg->nbOrdres < 3) {
        printf("X  | ERREUR - Il faut au moins 3 ordres\n");
        return 0;
    }

    return 1;
}

void afficherOrdres(const Config* cfg) {
    for (int i = 0; i < cfg->nbOrdres; i++) {
        if (i > 0) printf(" | ");

        if (strcmp(cfg->ordres[i], "KI") == 0) printf("KI (B -> R)");
        else if (strcmp(cfg->ordres[i], "LO") == 0) printf("LO (B <- R)");
        else if (strcmp(cfg->ordres[i], "SO") == 0) printf("SO (B <-> R)");
        else if (strcmp(cfg->ordres[i], "NI") == 0) printf("NI (B ^)");
        else if (strcmp(cfg->ordres[i], "MA") == 0) printf("MA (R ^)");
    }
    printf("\n");
}

int ordreExiste(const Config* cfg, const char* ordre) {
    for (int i = 0; i < cfg->nbOrdres; i++) {
        if (strcmp(cfg->ordres[i], ordre) == 0) {
            return 1;
        }
    }
    return 0;
}
```

5.4 Code - config.h

```
/**
 * @file config.h
 * @brief Gestion de la configuration du jeu
 */

#pragma once
#include "animal.h"

enum { MAX_ANIMAUX = 10, MAX_ORDRES = 5, MAX_NOM_ORDRE = 3};

/**
 * @brief Structure représentant la configuration du jeu
 */
```

```c
typedef struct {
    Animal animaux[MAX_ANIMAUX]; // Liste des animaux
    int nbAnimaux; // Nombre d'animaux
    char ordres[MAX_ORDRES][MAX_NOM_ORDRE]; // Liste des ordres
    int nbOrdres; // Nombre d'ordres
} Config;

/**
 * @brief Charge la configuration depuis le fichier crazy.cfg
 * @param[out] cfg Pointeur vers la configuration
 * @return 1 si succès et 0 si erreur
 */

int chargerConfig(Config* cfg);

/**
 * @brief Affiche les ordres disponibles
 * @param[in] cfg Pointeur vers la configuration
 */

void afficherOrdres(const Config* cfg);

/**
 * @brief Vérifie si un ordre existe dans la configuration
 * @param[in] cfg Pointeur vers la configuration
 * @param[in] ordre Ordre à vérifier
 * @return 1 si l'ordre existe et 0 sinon
 */

int ordreExiste(const Config* cfg, const char* ordre);
```

5.5 Code - jeu.c

```c
#include "jeu.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#pragma warning(disable: 4996)

int calculerNbCartes(int n) {
    if (n < 0) return 0;
    int resultat = 1;
    for (int i = 2; i <= n + 1; i++) {
        resultat *= i;
    }
```

```
    return resultat;
}

int compterCartesUtilisees(const Jeu* jeu) {
    int compte = 0;
    for (int i = 0; i < jeu->nbCartes; i++) {
        if (jeu->cartes[i].utilisee) compte++;
    }
    return compte;
}

int initJeu(Jeu* jeu, char** noms, int nbJoueurs) {
    if (!chargerConfig(&jeu->cfg)) return 0;
    if (!initJoueurs(&jeu->joueurs, noms, nbJoueurs)) return 0;

    int nbAnimaux = jeu->cfg.nbAnimaux;
    int nbCartesMax = calculerNbCartes(nbAnimaux);

    jeu->cartes = malloc(nbCartesMax * sizeof(Carte));
    if (!jeu->cartes) {
        printf("Erreur: allocation mémoire pour les cartes\n");
        return 0;
    }

    genererCartes(jeu);
    melangerCartes(jeu);

    jeu->indexProchaineCarte = 0;
    return 1;
}

void genererCartes(Jeu* jeu) {
    jeu->nbCartes = 0;
    int n = jeu->cfg.nbAnimaux;
    if (n <= 0) return;

    int perm[MAX_ANIMAUX];
    int c[MAX_ANIMAUX];
    for (int i = 0; i < n; i++) {
        perm[i] = i;
        c[i] = 0;
    }

    int nbCartesMax = calculerNbCartes(n);

    while (1) {
        for (int coupe = 0; coupe <= n && jeu->nbCartes < nbCartesMax; coupe++) {
            Situation s;
            initSituation(&s);
```

```
            for (int i = 0; i < coupe; i++) {
                empilerPodium(&s.bleu, &jeu->cfg.animaux[perm[i]]);
            }
            for (int i = coupe; i < n; i++) {
                empilerPodium(&s.rouge, &jeu->cfg.animaux[perm[i]]);
            }

            jeu->cartes[jeu->nbCartes].sit = s;
            jeu->cartes[jeu->nbCartes].utilisee = 0;
            jeu->nbCartes++;
        }

        int i = 0;
        while (i < n && c[i] >= i) {
            c[i] = 0;
            i++;
        }
        if (i >= n) break;

        if (i % 2 == 0) {
            int temp = perm[0];
            perm[0] = perm[i];
             perm[i] = temp;
        } else {
            int temp = perm[c[i]];
            perm[c[i]] = perm[i];
            perm[i] = temp;
        }
        c[i]++;

    }
}

void melangerCartes(Jeu* jeu) {
    for (int i = jeu->nbCartes - 1; i >= 1; i--) {
        int j = rand() % (i + 1);
        Carte temp = jeu->cartes[i];
        jeu->cartes[i] = jeu->cartes[j];
        jeu->cartes[j] = temp;
    }
}

int tirerCarte(Jeu* jeu) {
    int idx = jeu->indexProchaineCarte;
    int nb = jeu->nbCartes;
    if (idx < nb) {
        jeu->objectif = jeu->cartes[idx].sit;
        jeu->indexProchaineCarte++;
        return 1;
```

```
    }
    return 0;
}

void afficherSituationJeu(const Jeu* jeu) {
    int largeurs[4] = {4, 4, 4, 4};
    const Podium* piles[4];

    piles[0] = &jeu->actuelle.bleu;
    piles[1] = &jeu->actuelle.rouge;
    piles[2] = &jeu->objectif.bleu;
    piles[3] = &jeu->objectif.rouge;

    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < piles[i]->nb; j++) {
            int taille = (int)strlen(piles[i]->animaux[j].nom);
            if (taille > largeurs[i]) {
                largeurs[i] = taille;
            }
        }
    }

    int h1, h2, hMax;

    if (piles[0]->nb > piles[1]->nb) h1 = piles[0]->nb;
    else h1 = piles[1]->nb;

    if (piles[2]->nb > piles[3]->nb) h2 = piles[2]->nb;
    else h2 = piles[3]->nb;

    if (h1 > h2) hMax = h1;
    else hMax = h2;

    for (int lig = hMax - 1; lig >= -2; lig--) {
        for (int col = 0; col < 4; col++) {
            const char* mot = "";

            if (lig == -1) {
                mot = "----";
            } else if (lig == -2) {
                if (col % 2 == 0) mot = "BLEU";
                else mot = "ROUGE";
            } else {
                if (lig < piles[col]->nb) {
                    mot = piles[col]->animaux[lig].nom;
                }
            }

            printf("%s", mot);
```

```
            int n = (int)strlen(mot);
            int nbEspaces = (largeurs[col] - n) + 2;
            for (int e = 0; e < nbEspaces; e++) {
                printf(" ");
            }

            if (col == 1) {
                if (lig == -1) {
                    printf("==>  ");
                } else {
                    printf("     ");
                }
            }
        }
        printf("\n");
    }
    printf("\n");
}

int executerSequence(const Jeu* jeu, const char* sequence, Situation* resultat, char*
↪  ordreEchoue) {
    copierSituation(&jeu->actuelle, resultat);

    int taille = strlen(sequence);

    for (int i = 0; i < taille; i += 2) {
        if (i + 1 >= taille) {
            if (ordreEchoue) {
                ordreEchoue[0] = sequence[i];
                ordreEchoue[1] = '\0';
            }
            return 0;
        }

        char ordre[3] = {sequence[i], sequence[i+1], '\0'};

        if (!ordreExiste(&jeu->cfg, ordre)) {
            if (ordreEchoue) {
                ordreEchoue[0] = ordre[0];
                ordreEchoue[1] = ordre[1];
                ordreEchoue[2] = '\0';
            }
            return 0;
        }

        if (strcmp(ordre, "KI") == 0) {
            executerKI(resultat);
        } else if (strcmp(ordre, "LO") == 0) {
```

```
            executerLO(resultat);
        } else if (strcmp(ordre, "SO") == 0) {
            executerSO(resultat);
        } else if (strcmp(ordre, "NI") == 0) {
            executerNI(resultat);
        } else if (strcmp(ordre, "MA") == 0) {
            executerMA(resultat);
        }
    }

    if (ordreEchoue) ordreEchoue[0] = '\0';
    return 1;
}

void traiterSaisie(Jeu* jeu, const char* saisie, int* finPartie) {
    char nomJoueur[MAX_NOM_JOUEUR];
    char sequence[100];

    if (sscanf(saisie, "%s %s", nomJoueur, sequence) != 2) {
        printf("Saisie invalide\n\n");
        return;
    }

    int idxJoueur = trouverJoueur(&jeu->joueurs, nomJoueur);
    if (idxJoueur == -1) {
        printf("Joueur inconnu\n\n");
        return;
    }

    if (!jeu->joueurs.joueurs[idxJoueur].peutJouer) {
        printf("%s ne peut pas jouer\n\n", nomJoueur);
        return;
    }

    Situation resultat;
    char ordreEchoue[3] = {'\0'};
    if (!executerSequence(jeu, sequence, &resultat, ordreEchoue)) {
        if (ordreEchoue[1] != '\0') {
            printf("L'ordre %s n'existe pas\n\n", ordreEchoue);
        } else {
            printf("L'ordre %c n'existe pas\n\n", ordreEchoue[0]);
        }
        return;
    }

    if (situationsEgales(&resultat, &jeu->objectif)) {
        printf("%s gagne un point\n\n", nomJoueur);
        jeu->joueurs.joueurs[idxJoueur].score++;
        copierSituation(&jeu->objectif, &jeu->actuelle);
```

```
        if (jeu->indexProchaineCarte > 0) {
            jeu->cartes[jeu->indexProchaineCarte - 1].utilisee = 1;
        }
        if (compterCartesUtilisees(jeu) >= jeu->nbCartes) {
            *finPartie = 1;
            return;
        }

        if (tirerCarte(jeu)) {
            reinitTour(&jeu->joueurs);
            afficherSituationJeu(jeu);
        } else {
            *finPartie = 1;
        }
    } else {
        printf("La sequence ne conduit pas a la situation attendue -- %s ne peut plus
        ↪  jouer durant ce tour\n\n", nomJoueur);

        jeu->joueurs.joueurs[idxJoueur].peutJouer = 0;

        if (nbJoueursActifs(&jeu->joueurs) == 1) {
            for (int i = 0; i < jeu->joueurs.nb; i++) {
                if (jeu->joueurs.joueurs[i].peutJouer) {
                    printf("%s gagne un point car lui seul peut encore jouer durant
                    ↪  ce tour\n\n", jeu->joueurs.joueurs[i].nom);
                    jeu->joueurs.joueurs[i].score++;
                    break;
                }
            }
            copierSituation(&jeu->objectif, &jeu->actuelle);
            if (jeu->indexProchaineCarte > 0) {
                jeu->cartes[jeu->indexProchaineCarte - 1].utilisee = 1;
            }

            if (tirerCarte(jeu)) {
                reinitTour(&jeu->joueurs);
                afficherSituationJeu(jeu);
            } else {
                if (jeu->indexProchaineCarte > 0) {
                    jeu->cartes[jeu->indexProchaineCarte - 1].utilisee = 1;
                }
                if (compterCartesUtilisees(jeu) >= jeu->nbCartes) {
                    *finPartie = 1;
                } else {
                    printf("X  | ERREUR - Plus de cartes disponibles (une err c'est
                    ↪  produite).\n\n");
                    *finPartie = 1;
                }
            }
```

```
        }
    }
}

void lancerPartie(Jeu* jeu) {
    afficherOrdres(&jeu->cfg);
    printf("\n");

    if (!tirerCarte(jeu)) {
        printf("X  | ERREUR - Aucune carte disponible.\n\n");
        return;
    }

    copierSituation(&jeu->objectif, &jeu->actuelle);
    if (jeu->indexProchaineCarte > 0) {
        jeu->cartes[jeu->indexProchaineCarte - 1].utilisee = 1;
    }

    if (!tirerCarte(jeu)) {
        printf("X  | ERREUR - Pas assez de cartes.\n\n");
        return;
    }

    afficherSituationJeu(jeu);

    int finPartie = 0;
    while (!finPartie) {
        char saisie[200];
        if (fgets(saisie, 200, stdin) != NULL) {
            for (int i = 0; saisie[i] != '\0'; i++) {
                if (saisie[i] == '\n') {
                    saisie[i] = '\0';
                    break;
                }
            }
            traiterSaisie(jeu, saisie, &finPartie);
        }
    }

    afficherScores(&jeu->joueurs);
}

void quitterJeu(Jeu* jeu) {
    if (jeu->cartes) {
        free(jeu->cartes);
        jeu->cartes = NULL;
    }
}
```

5.6 Code - jeu.h

```c
/**
 * @file jeu.h
 * @brief Logique principale du jeu
 */

#pragma once
#include "config.h"
#include "situation.h"
#include "joueur.h"

/**
 * @brief Calcule le nombre total de cartes selon la formule (n+1)!
 * @param[in] n Nombre d'animaux
 * @return Nombre de cartes = (n+1)!
 */
int calculerNbCartes(int n);

/**
 * @brief Structure représentant une carte objectif
 */

typedef struct {
    Situation sit; // Situation de la carte
    int utilisee; // 1 si déjà utilisée et 0 sinon
} Carte;

/**
 * @brief Structure représentant le jeu
 */

typedef struct {
    Config cfg; // Configuration
    Situation actuelle; // Situation actuelle
    Situation objectif; // Situation à atteindre
    Joueurs joueurs; // Liste des joueurs
    Carte* cartes; // Pointeur vers les cartes (allocation dynamique)
    int nbCartes; // Nombre de cartes
    int indexProchaineCarte; // Index de la prochaine carte à tirer
} Jeu;

/**
 * @brief Initialise le jeu
 * @param[out] jeu Pointeur vers le jeu
 * @param[in] noms Noms des joueurs
 * @param[in] nbJoueurs Nombre de joueurs
 * @return 1 si succès et 0 si erreur
 */
```

```c
int initJeu(Jeu* jeu, char** noms, int nbJoueurs);

/**
 * @brief Génère toutes les cartes possibles
 * @param[in,out] jeu Pointeur vers le jeu
 */

void genererCartes(Jeu* jeu);

/**
 * @brief Mélanger les Cartes du jeu
 * @param[in,out] jeu Pointeur vers le jeu
 */

void melangerCartes(Jeu* jeu);

/**
 * @brief Tire une carte au hasard
 * @param[in,out] jeu Pointeur vers le jeu
 * @return 1 si une carte a été tirée et 0 si plus de cartes disponibles
 */

int tirerCarte(Jeu* jeu);

/**
 * @brief Affiche la situation actuelle et l'objectif
 * @param[in] jeu Pointeur vers le jeu
 */

void afficherSituationJeu(const Jeu* jeu);

/**
 * @brief Exécute une séquence d'ordres
 * @param[in] jeu Pointeur vers le jeu
 * @param[in] sequence Séquence d'ordres
 * @param[out] resultat Situation résultante
 * @return 1 si valide et 0 sinon
 */

int executerSequence(const Jeu* jeu, const char* sequence, Situation* resultat, char*
↪  ordreEchoue);

/**
 * @brief Traite la saisie d'un joueur
 * @param[in,out] jeu Pointeur vers le jeu
 * @param[in] saisie Saisie du joueur
 * @param[out] finPartie Pointeur pour indiquer la fin de la partie
 */
```

```c
void traiterSaisie(Jeu* jeu, const char* saisie, int* finPartie);

/**
 * @brief Lance une partie complète
 * @param[in,out] jeu Pointeur vers le jeu
 */

void lancerPartie(Jeu* jeu);

/**
 * @brief Libère les ressources du jeu
 * @param[in,out] jeu Pointeur vers le jeu
 */

void quitterJeu(Jeu* jeu);
```

5.7 Code - joueur.c

```c
#include "joueur.h"
#include <stdio.h>
#include <string.h>
#include <assert.h>

int initJoueurs(Joueurs* js, char** noms, int nbNoms) {
    if (nbNoms < 2) {
        printf("ERREUR | Il faut au moins 2 joueurs\n");
        return 0;
    }

    if (nbNoms > MAX_JOUEURS) {
        printf("ERREUR | Trop de joueurs (max %d)\n", MAX_JOUEURS);
        return 0;
    }

    for (int i = 0; i < nbNoms; i++) {
        for (int j = i + 1; j < nbNoms; j++) {
            if (strcmp(noms[i], noms[j]) == 0) {
                printf("ERREUR | Les noms des joueurs doivent etre distincts\n");
                return 0;
            }
        }
    }

    js->nb = nbNoms;
    for (int i = 0; i < nbNoms; i++) {
        assert(strlen(noms[i]) < MAX_NOM_JOUEUR);
        strcpy(js->joueurs[i].nom, noms[i]);
```

```
        js->joueurs[i].score = 0;
        js->joueurs[i].peutJouer = 1;
    }

    return 1;
}

int trouverJoueur(const Joueurs* js, const char* nom) {
    for (int i = 0; i < js->nb; i++) {
        if (strcmp(js->joueurs[i].nom, nom) == 0) {
            return i;
        }
    }
    return -1;
}

void reinitTour(Joueurs* js) {
    for (int i = 0; i < js->nb; i++) {
        js->joueurs[i].peutJouer = 1;
    }
}

int nbJoueursActifs(const Joueurs* js) {
    int nb = 0;
    for (int i = 0; i < js->nb; i++) {
        if (js->joueurs[i].peutJouer) {
            nb++;
        }
    }
    return nb;
}

void afficherScores(const Joueurs* js) {
    Joueur tri[MAX_JOUEURS];
    for (int i = 0; i < js->nb; i++) {
        tri[i] = js->joueurs[i];
    }

    for (int i = 0; i < js->nb - 1; i++) {
        for (int j = i + 1; j < js->nb; j++) {
            if (tri[i].score < tri[j].score || (tri[i].score == tri[j].score &&
            ↪  strcmp(tri[i].nom, tri[j].nom) > 0)) {
                Joueur tmp = tri[i];
                tri[i] = tri[j];
                tri[j] = tmp;
            }
        }
    }
```

```
    for (int i = 0; i < js->nb; i++) {
        printf("%s %d\n", tri[i].nom, tri[i].score);
    }
}
```

5.8 Code - joueur.h

```
/**
 * @file joueur.h
 * @brief Gestion des joueurs
 */

#pragma once

enum { MAX_NOM_JOUEUR = 30, MAX_JOUEURS = 10 };

/**
 * @brief Structure représentant un joueur
 */

typedef struct {
    char nom[MAX_NOM_JOUEUR]; // Nom du joueur
    int score; // Score du joueur
    int peutJouer; // 1 si peut jouer ce tour et 0 sinon
} Joueur;

/**
 * @brief Liste de joueurs
 */

typedef struct {
    Joueur joueurs[MAX_JOUEURS]; // Tableau des joueurs
    int nb; // Nombre de joueurs
} Joueurs;

/**
 * @brief Initialise la liste de joueurs
 * @param[out] js Pointeur vers la liste
 * @param[in] noms Tableau des noms
 * @param[in] nbNoms Nombre de noms
 * @return 1 si succès et 0 si erreur
 */

int initJoueurs(Joueurs* js, char** noms, int nbNoms);

/**
 * @brief Trouve un joueur par son nom
```

```
 * @param[in] js Pointeur vers la liste
 * @param[in] nom Nom du joueur
 * @return Index du joueur ou -1 si non trouvé
 */

int trouverJoueur(const Joueurs* js, const char* nom);

/**
 * @brief Réinitialise les joueurs pour un nouveau tour
 * @param[in,out] js Pointeur vers la liste
 */

void reinitTour(Joueurs* js);

/**
 * @brief Compte le nombre de joueurs pouvant encore jouer
 * @param[in] js Pointeur vers la liste
 * @return Nombre de joueurs pouvant jouer
 */

int nbJoueursActifs(const Joueurs* js);

/**
 * @brief Affiche les scores finaux
 * @param[in] js Pointeur vers la liste
 */

void afficherScores(const Joueurs* js);
```

5.9 Code - main.c

```
#include "jeu.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char* argv[]) {
    unsigned int seed = (unsigned int)time(NULL);
    srand(seed);

    Jeu jeu;
    if (!initJeu(&jeu, argv + 1, argc - 1)) {
        return 1;
    }

    lancerPartie(&jeu);
    quitterJeu(&jeu);
```

```
    return 0;
}
```

5.10 Code - podium.c

```
#include "podium.h"
#include <stdio.h>
#include <assert.h>

void initPodium(Podium* p) {
    p->nb = 0;
}

int estVidePodium(const Podium* p) {
    return p->nb == 0;
}

void empilerPodium(Podium* p, const Animal* a) {
    assert(p->nb < MAX_ANIMAUX_PODIUM);
    p->animaux[p->nb] = *a;
    p->nb++;
}

void depilerPodium(Podium* p, Animal* a) {
    assert(!estVidePodium(p));
    p->nb--;
    *a = p->animaux[p->nb];
}

void sommetPodium(const Podium* p, Animal* a) {
    assert(!estVidePodium(p));
    *a = p->animaux[p->nb - 1];
}

void basePodium(const Podium* p, Animal* a) {
    assert(!estVidePodium(p));
    *a = p->animaux[0];
}

void monterBasePodium(Podium* p) {
    assert(!estVidePodium(p));
    if (p->nb > 1) {
        Animal base = p->animaux[0];
        for (int i = 0; i < p->nb - 1; i++) {
            p->animaux[i] = p->animaux[i + 1];
        }
        p->animaux[p->nb - 1] = base;
    }
```

```
}

void copierPodium(const Podium* p1, Podium* p2) {
    p2->nb = p1->nb;
    for (int i = 0; i < p1->nb; i++) {
        p2->animaux[i] = p1->animaux[i];
    }
}

void affichePodium(const Podium* p) {
    for (int i = p->nb - 1; i >= 0; i--) {
        afficheAnimal(&p->animaux[i]);
        if (i > 0) printf(" ");
    }
}

int podiumsEgaux(const Podium* p1, const Podium* p2) {
    if (p1->nb != p2->nb) return 0;
    for (int i = 0; i < p1->nb; i++) {
        if (!animauxEgaux(&p1->animaux[i], &p2->animaux[i])) {
            return 0;
        }
    }
    return 1;
}
```

5.11 Code - podium.h

```
/**
 * @file podium.h
 * @brief Gestion d'un podium (pile d'animaux)
 */

#pragma once
#include "animal.h"

enum { MAX_ANIMAUX_PODIUM = 10 };

/**
 * @brief Structure représentant un podium (pile d'animaux)
 */

typedef struct {
    Animal animaux[MAX_ANIMAUX_PODIUM]; // Tableau des animaux
    int nb; // Nombre d'animaux sur le podium
} Podium;

/**
```

```
 * @brief Initialise un podium vide
 * @param[out] p Pointeur vers le podium
 */

void initPodium(Podium* p);

/**
 * @brief Vérifie si un podium est vide
 * @param[in] p Pointeur vers le podium
 * @return 1 si vide et 0 sinon
 */

int estVidePodium(const Podium* p);

/**
 * @brief Ajoute un animal au sommet du podium
 * @param[in,out] p Pointeur vers le podium
 * @param[in] a Pointeur vers l'animal à ajouter
 */

void empilerPodium(Podium* p, const Animal* a);

/**
 * @brief Retire l'animal au sommet du podium
 * @param[in,out] p Pointeur vers le podium
 * @param[out] a Pointeur où stocker l'animal retiré
 */

void depilerPodium(Podium* p, Animal* a);

/**
 * @brief Obtient l'animal au sommet sans le retirer
 * @param[in] p Pointeur vers le podium
 * @param[out] a Pointeur où stocker l'animal
 */

void sommetPodium(const Podium* p, Animal* a);

/**
 * @brief Obtient l'animal à la base sans le retirer
 * @param[in] p Pointeur vers le podium
 * @param[out] a Pointeur où stocker l'animal
 */

void basePodium(const Podium* p, Animal* a);

/**
 * @brief Fait monter l'animal de la base au sommet
 * @param[in,out] p Pointeur vers le podium
```

```
 */

void monterBasePodium(Podium* p);

/**
 * @brief Copie un podium dans un autre
 * @param[in] p1 Podium à copier
 * @param[out] p2 Podium copier
 */

void copierPodium(const Podium* p1, Podium* p2);

/**
 * @brief Affiche un podium
 * @param[in] p Pointeur vers le podium
 */

void affichePodium(const Podium* p);

/**
 * @brief Compare deux podiums
 * @param[in] p1 Premier podium
 * @param[in] p2 Second podium
 * @return 1 si identiques et 0 sinon
 */

int podiumsEgaux(const Podium* p1, const Podium* p2);
```

5.12 Code - situation.c

```
#include "situation.h"
#include <stdio.h>

void initSituation(Situation* s) {
    initPodium(&s->bleu);
    initPodium(&s->rouge);
}

void copierSituation(const Situation* s1, Situation* s2) {
    copierPodium(&s1->bleu, &s2->bleu);
    copierPodium(&s1->rouge, &s2->rouge);
}

int situationsEgales(const Situation* s1, const Situation* s2) {
    return podiumsEgaux(&s1->bleu, &s2->bleu) &&
           podiumsEgaux(&s1->rouge, &s2->rouge);
}
```

```
void afficheSituation(const Situation* s) {
    affichePodium(&s->bleu);
    printf("\n");
    printf("---- ----\n");
    printf("BLEU ROUGE");
}

void executerKI(Situation* s) {
    if (!estVidePodium(&s->bleu)) {
        Animal a;
        depilerPodium(&s->bleu, &a);
        empilerPodium(&s->rouge, &a);
    }
}

void executerLO(Situation* s) {
    if (!estVidePodium(&s->rouge)) {
        Animal a;
        depilerPodium(&s->rouge, &a);
        empilerPodium(&s->bleu, &a);
    }
}

void executerSO(Situation* s) {
    if (!estVidePodium(&s->bleu) && !estVidePodium(&s->rouge)) {
        Animal aBleu, aRouge;
        depilerPodium(&s->bleu, &aBleu);
        depilerPodium(&s->rouge, &aRouge);
        empilerPodium(&s->bleu, &aRouge);
        empilerPodium(&s->rouge, &aBleu);
    }
}

void executerNI(Situation* s) {
    if (!estVidePodium(&s->bleu)) {
        monterBasePodium(&s->bleu);
    }
}

void executerMA(Situation* s) {
    if (!estVidePodium(&s->rouge)) {
        monterBasePodium(&s->rouge);
    }
}
```

5.13 Code - situation.h

```
/**
```

```
 * @file situation.h
 * @brief Gestion d'une situation de jeu (2 podiums)
 */

#pragma once
#include "podium.h"

/**
 * @brief Structure représentant une situation de jeu
 */

typedef struct {
    Podium bleu;  // Podium bleu
    Podium rouge; // Podium rouge
} Situation;

/**
 * @brief Initialise une situation vide
 * @param[out] s Pointeur vers la situation
 */

void initSituation(Situation* s);

/**
 * @brief Copie une situation dans une autre
 * @param[in]  s1 Situation initial
 * @param[out] s2 Situation final
 */

void copierSituation(const Situation* s1, Situation* s2);

/**
 * @brief Compare deux situations
 * @param[in] s1 Première situation
 * @param[in] s2 Seconde situation
 * @return 1 si identiques et 0 sinon
 */

int situationsEgales(const Situation* s1, const Situation* s2);

/**
 * @brief Affiche une situation de jeu
 * @param[in] s Pointeur vers la situation
 */

void afficheSituation(const Situation* s);

/**
 * @brief Exécute l'ordre KI (bleu vers rouge)
```

```
 * @param[in,out] s Pointeur vers la situation
 */

void executerKI(Situation* s);

/**
 * @brief Exécute l'ordre LO (rouge vers bleu)
 * @param[in,out] s Pointeur vers la situation
 */

void executerLO(Situation* s);

/**
 * @brief Exécute l'ordre SO (échange des sommets)
 * @param[in,out] s Pointeur vers la situation
 */

void executerSO(Situation* s);

/**
 * @brief Exécute l'ordre NI (base vers sommet du bleu)
 * @param[in,out] s Pointeur vers la situation
 */

void executerNI(Situation* s);

/**
 * @brief Exécute l'ordre MA (base vers sommet du rouge)
 * @param[in,out] s Pointeur vers la situation
 */

void executerMA(Situation* s);
```

5.14 Code - crazy.cfg

```

```

OURS ELEPHANT LION KI LO SO NI MA