# IUT de Paris – Rives de Seine

## SAE S1.02 – Comparaison d'approches algorithmiques

### Rapport du projet Octo-Verso

EL-SAYED Mohamed-Yazan, groupe 101 & KESKAS Loqman, groupe 105

## Table des matières



## 1. Introduction

Ce projet a pour but de développer un logiciel permettant de disputer des parties de *Octo-Verso.* Un jeu qui se joue à deux ou avec deux groupes, chaque joueur ou équipe se voit attribué aléatoirement 12 chevalets sur lesquels sont inscrits une lettre sur le recto et le verso de ceux-ci.

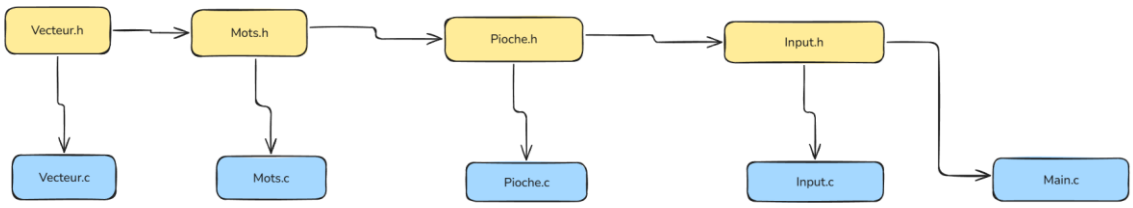
A partir de ces chevalets, chaque joueur compose au départ un mot de quatre lettres qui sera posé sur un rail qui peut être inversé et doit toujours contenir 8 chevalets. Ensuite, chacun des deux joueurs doit former un nouveau mot dans le rail (inversé ou non) en utilisant ses chevalets et au minimum deux chevalets présents sur l'une des extrémités du rail. La formation de nouveaux mots à partir d'une extrémité du rail fait tomber les chevalets présents de l'autre extrémité qui seront récupérés par l'adversaire.

Si un joueur parvient à former un mot de 8 lettres (un Octo), alors celui-ci peut alors rendre un de ses chevalets à la pioche. Au contraire, si un joueur n'est pas très inspiré, il peut échanger un de ses chevalets avec un présent dans la pioche. Ainsi, une partie se termine lorsqu'un joueur s'est débarrassé de tous ses chevalets.

## 2. Diagramme de dépendances

Notre programme a été architecturé autour de 4 composants : Vecteur, Pioche, Mots et Input. Le composant Vecteur nous permet de manipuler les éléments constituant le rail. Le composant Pioche étant de type Vecteur, hérite des facilités de manipulation des éléments du rail mais propose différentes fonctionnalités permettant aux joueurs de manipuler son jeu de chevalets au cours de la partie, nous considérons le composant Pioche comme un élément de gestion des chevalets plus affiné que le composant Vecteur. Le composant Mots est un “historique” des mots joués. Enfin le composant Input est composé de fonctions permettant de traiter les messages échangés avec les joueurs.

Voici le diagramme montrant les dépendances des composants au travers des directives d’inclusion :




## 3. Tests unitaires

Les composants *Vecteur*, *Pioche* , *Mots* et *Input* ont été testés au travers de trois fonctions que voici :

*Vecteur :*

```
void testVecteur() {
	Vecteur v;
	initVecteur(&v, 8);
	assert(taille(&v) == 0);

	ajouter(&v, 2);
	assert(obtenir(&v, 0) == 2);
	modifier(&v, 0, 3);
	assert(obtenir(&v, 0) == 3);

	ajouter(&v, 10);
	assert(taille(&v) == 2);
	supprimer(&v, 0);
	assert(obtenir(&v, 0) == 10);

	detruireVecteur(&v);
}
```

*Pioche :*

```
void testPioche() {
    int COMPO_LETTRES_DEPART[NB_LETTRES_DEPART] = { 9, 1, 2, 3, 14, 1, 1, 1, 7, 1, 5, 3,
6, 0, 5, 2, 1, 6, 7, 6, 5, 2, 0, 0, 0, 0 };
    Pioche p;
    initJeu(&p, 1, COMPO_LETTRES_DEPART);
    modifier(&p, 0, 'E');
    assert(verifieLettre(&p, 'E') == 1);

    ajoute("AS", &p);
    assert(taille(&p) == 3);
    assert(indexLettre('D') == 4);

    Pioche p2;
    initVecteur(&p2, 8);
    inverseRail(&p, &p2);
    assert(p2.elements[0] == 'S');
    assert(verifMot("SAE", &p2) == 1);

    ajouteDeb("RENDU", &p2);
    assert(verifMot("RENDUSAE", &p2) == 1);
    assert(p.elements[0] == 'R');
    ajoute("RAPPORT", &p);
    assert(verifMot("RAPPORT", &p) == 1);

    transferer(&p2, &p, 1, 5);
    assert(verifMot("RENDU", &p) == 1);
    echangeChevalet(&p, 'R', COMPO_LETTRES_DEPART);
    assert(p.elements[0] != 'R');

    detruireVecteur(&p);
    detruireVecteur(&p2);
}
```

*Mots :*

```
   void testMots() {
       Mots m;
       initMots(&m, "un", 10);
       assert(MotConnu(&m, "un") == 1);
       assert(MotConnu(&m, "deux") == 0);

       ajouterMot(&m, "deux");
       assert(MotConnu(&m, "deux") == 1);

       detruireMots(&m);
   }
```

*Input :*

```
void testInput() {
    Input p;
    char mot[NB_LETTRES_RAIL + 3] = "TEST(ER)";// ( = 1 , ) = 1 , \0 = 1 :
1+1+1 = 3
    extraire_entre_parenthese(mot, p.EntreParenthese);
    assert(strcmp(p.EntreParenthese, "ER") == 0);
    Extraire_Hors_Parenthese(mot, p.HorsParenthese);
    assert(strcmp(p.HorsParenthese, "TEST") == 0);
    forme_mot(mot, p.MotForme);
    assert(strcmp(p.MotForme, "TESTER") == 0);

    Pioche joueur1;
    char teste1[NB_CHEVALETS_DEB + 1] = "CEEEENNNRSTU";
    initVecteur(&joueur1, NB_CHEVALETS_DEB);
    for (int i = 0; i < NB_CHEVALETS_DEB ; ++i) {
        ajouter(&joueur1, teste1[i]);
    }
    prelever(&joueur1, "CENE");

    char tmp[NB_CHEVALETS_DEB + 1 - 4 ]; // 4 = strlen("CENE")
    int i;
    for (i = 0; i < joueur1.nbElements; ++i)
        tmp[i] = obtenir(&joueur1, i);
    tmp[i] = '\0';
    assert(strcmp(tmp, "EENNRSTU") == 0);

    Pioche p2;
    initVecteur(&p2, NB_CHEVALETS_DEB);
    mettre_a_jour(&joueur1, &p2);
    for (i = 0; i < p2.nbElements; ++i)
        assert(obtenir(&joueur1, i) == obtenir(&p2, i));


    detruireVecteur(&joueur1);
    detruireVecteur(&p2);
}
```

## 4. Trace d’exécution

Au début de chaque partie, afin de déterminer l’ordre de jeu, chaque joueur donne un mot qu’il souhaite composer avec ses chevalets. Les mots sont placés dans le rail dans l’ordre alphabétique et le joueur qui possède le mot qui vient en premier dans l’ordre alphabétique commence.

Pour le bon déroulé de la partie, à chaque tour s’affiche la situation courante du jeu ainsi que le numéro du joueur qui doit jouer son tour. Et ce jusqu’à la fin de la partie.

```
1 : AAEEILNNOORT
2 : AEELNQRSSSTU

1> RIEL
2> LUTE

1 : AAENNOOT
2 : AENQRSSS
R : LUTERIEL
V : LEIRETUL

2> V RENA(LE)

1 : AAEELNNOOTTU
2 : QSSS
R : RIELANER
V : RENALEIR

1> V ANNELU(RE)

1 : AEOOTT
2 : AEILNQRSSS
R : ERULENNA
V : ANNELURE

-1> T

1 : AEOOT
2 : AEILNQRSSS
R : ERULENNA
V : ANNELURE
```

```
2> V (RE)INSERA

1 : AAEELNNOOTU
2 : LQSS
R : ARESNIER
V : REINSERA

-2> Q

1 : AAEELNNOOTU
2 : LSS
R : ARESNIER
V : REINSERA

1> V (RA)TONNEE

1 : AALOU
2 : EEILNRSSS
R : EENNOTAR
V : RATONNEE

-1> A

1 : ALOU
2 : EEILNRSSS
R : EENNOTAR
V : RATONNEE

2> R (AR)LESIEN
```

```
1 : AEELNNOOTU
2 : RSS
R : ARLESIEN
V : NEISELRA

-2> S

1 : AEELNNOOTU
2 : RS
R : ARLESIEN
V : NEISELRA

1> R (EN)TONNE

1 : AELOU
2 : AELRRSS
R : IENTONNE
V : ENNOTNEI

2> R LASSER(IE)

1 : AEELNNNOOTU
2 : R
R : LASSERIE
V : EIRESSAL

-2> R

Fin du jeu
```

## 5. bilan

Le projet a été assez méticuleux à mettre en place, d'une part lors du choix de l'architecture et des composants qui ont été choisi rigoureusement dans le but d'optimiser au maximum notre programme ainsi que son développement.

De plus, la découverte et la prise en main de la compilation séparée nous a permis d'obtenir une meilleure **organisation du code**. En séparant les déclarations dans les fichiers d'en-tête et les définitions dans les fichiers sources, le projet est devenu plus lisible et plus facile à maintenir. Chaque module étant indépendant, la répartition des tâches était plus simple tandis que la détection et la résolution d'erreurs ne nous prenaient moins de temps qu'auparavant.

# 6. Annexe – Code source

## Input.h

```
#pragma once
#include <stdio.h>
#include <stdlib.h>
//#include <assert.h>
#include "pioche.h"

typedef struct {
    char EntreParenthese[NB_LETTRES_RAIL + 1];
    char HorsParenthese[NB_LETTRES_RAIL + 1];
    char MotForme[NB_LETTRES_RAIL + 1];
}Input;

/**
 * @brief prend une chaîne de caractère(un mot) et copie ce qui est entre
parenthèses dans resultat
 * @param[in] input : la chaîne de caractère à traiter
 * @param[out] resultat : ce qui est entre parenthèses
 */
void Extraire_Hors_Parenthese(const char* input, char* resultat);

/**
 * @brief extrait ce qui est entre parenthèses dans une chaîne de caractère.
 * @param[in] entre : La chaîne de caractère à traiter
 * @param[out] motExtrait : mot extrait des parenthèses
 */
void extraire_entre_parenthese(const char* entre, char* motExtrait);

/**
 * @brief prend une chaîne de caractère(un mot) et enlève tous les caractères '('
rt ')'
 * @param[in] input : la chaîne de caractère à traiter
 * @param[out] MotForme : la chaîne de caractère sans '(' et ')'
 */
void forme_mot(const char* input, char* MotForme);

/**
 * @brief demande un joueur de former un mot en indiquant le coté du rail où il
doit être posé par la lettre correspondante
 (R ou V) suivi du mot.
 * @param[in] j : la pioche du joueur .
 * @param[in] rail : le rail recto (R).
 * @param[in] railInverse : le rail verso (V).
 * @param[in] dict : le dictionnaire des mots fraincais.
 * @param[out] inputJoueur : l'entré du joueur .
 * @param[in] mot_Joue : la liste des mots déja joués au cours du jeu .
 * @param[in] joueur : le numéro du joueur .
 * @param[in,out] commande : la commande entrée au premier coup (R ou V).
 * @return : si le coup est validée renvoie le code d'entrée du mot formé (coté
du rail traiter  1 = gauche , 2 = droite ).
 */
int saisieMotJoueur(const Pioche* j, const Pioche* rail,const Pioche*
railInverse,const Mots* dict,
    Input* inputJoueur,const Mots* mot_Joue, int joueur, char* commande);

/**
 * @brief enlever de la pioche du joueur les lettres dont il a utilisé pour
former le mot
```

```
 * @param[in] joueur
 * @param[in] mot
 */
void prelever(Pioche* joueur, char* mot);

/**
 * @brief demande aux joueurs de débarrasser d'une lettre de sa pioche
 * @param[in] joueur
 * @param[in] nbjoueur
 */
void debarasser(Pioche* joueur, int nbjoueur);

/**
 * @brief mets à jour P2 de P1 (copie p1 dans p2)
 * @param[in] p1
 * @param[in,out] p2
 */
void mettre_a_jour(const Pioche* p1, Pioche* p2);


/**
 * @brief Teste les foncrions liées au composant Input.
 */
void testInput();
```

## input.c

```
#pragma once
#include "input.h"

#pragma warning(disable : 4996 6031)

void extraire_entre_parenthese(const char* input, char* resultat) {
    int i = 0, j = 0;
    int dansParenthese = 0;

    while (input[i] != '\0') {

        if (input[i] == '(') {
            dansParenthese = 1;
        }
        else if (input[i] == ')') {
            dansParenthese = 0;
        }
        else if (dansParenthese) {
            resultat[j++] = input[i];
        }
        i++;
    }
    resultat[j] = '\0';
}

void Extraire_Hors_Parenthese(const char* input, char* resultat) {
    int i = 0, j = 0;
    int dansParenthese = 0;  // Indicateur pour savoir si nous sommes dans les
parenthèses

    // Parcourir chaque caractère de la chaîne d'entrée
    while (input[i] != '\0') {
```

```
        // Si nous rencontrons une parenthèse ouvrante, on entre dans la section
des parenthèses
        if (input[i] == '(') {
            dansParenthese = 1;
        }
        // Si nous rencontrons une parenthèse fermante, on sort de la section des
parenthèses
        else if (input[i] == ')') {
            dansParenthese = 0;
        }
        // Si nous ne sommes pas dans les parenthèses, on copie
        else if (!dansParenthese) {
            resultat[j++] = input[i];
        }
        i++;
    }
    resultat[j] = '\0';
}

void forme_mot(const char* input, char* MotForme) {
    int i = 0, j = 0;
    while (input[i] != '\0') {
        if (input[i] != '(' && input[i] != ')')
            MotForme[j++] = input[i];
        ++i;
    }
    MotForme[j] = '\0';
}

int saisieMotJoueur(const Pioche* j, const Pioche* rail, const Pioche*
railInverse, const Mots* dict,
    Input* inputJoueur, const Mots* mot_Joue, int joueur, char* commande){

    char EntreJoueur[NB_LETTRES_RAIL + 3]; // ( = 1 , ) = 1 , \0 = 1 : 1+1+1 = 3
    int codeEntre = 0;
    int coupe = 0;
    int a = 0;
    int b = 0;
    int c = 0;
    do {
        codeEntre = 0;

        if (coupe != 0) {
            printf("%d> ", joueur);
            scanf(" %c", commande);
            if (*commande != 'R' && *commande != 'V' && *commande != 'r' &&
*commande != 'v') return codeEntre;
        }
        scanf(" %s", EntreJoueur);

        char p = EntreJoueur[0];
        if (p == '(') codeEntre = 2; else codeEntre = 1;

        extraire_entre_parenthese(EntreJoueur, inputJoueur->EntreParenthese);
        Extraire_Hors_Parenthese(EntreJoueur, inputJoueur->HorsParenthese);
        forme_mot(EntreJoueur, inputJoueur->MotForme);
        ++coupe;
        a = verifMot(inputJoueur->HorsParenthese, j);
        b = verifier(mot_Joue, inputJoueur->MotForme, dict);
        c = verifier_lettres_extremite(*commande == 'R' || *commande == 'r' ?
rail : railInverse, inputJoueur->EntreParenthese, codeEntre);
    } while (//strlen(EntreJoueur) > NB_LETTRES_RAIL + 1 || // pas sur de cette
condition
```

```
        (a == 0) || (b == 0) || (c == 0));

    return codeEntre;
}

void prelever(Pioche* joueur, char* mot) {
    int n = (int)strlen(mot);
    for (int i = 0; i < n; ++i) {
        char lettre = mot[i];
        for (int j = 0; j < taille(joueur); ++j) {
            if (obtenir(joueur, j) == lettre) {
                supprimer(joueur, j);
                break;

            }
        }
    }
}

void debarasser(Pioche* joueur, int nbjoueur) {
    char lettre_a_debarasser;
    do {
        printf("-%d> ", nbjoueur);
        scanf(" %c", &lettre_a_debarasser);

    } while (verifieLettre(joueur, lettre_a_debarasser) == 0);

    for (int i = 0; i < taille(joueur); ++i) {
        if (obtenir(joueur, i) == lettre_a_debarasser) {
            supprimer(joueur, i);
            break;
        }
    }
}

void mettre_a_jour(const Pioche* p1, Pioche* p2) {
    if (taille(p2) == 0)
        for (int i = 0; i < p1->nbElements; ++i) {
            ajouter(p2, obtenir(p1, i));
        }
    else {
        for (int i = 0; i < taille(p1); ++i) {
            if (i > p2->nbElements - 1)
                ajouter(p2, obtenir(p1, i));
            else
                modifier(p2, i, obtenir(p1, i));
        }
    }
}

void testInput() {
    Input p;
    char mot[NB_LETTRES_RAIL + 3] = "TEST(ER)";// ( = 1 , ) = 1 , \0 = 1 : 1+1+1
= 3
    extraire_entre_parenthese(mot, p.EntreParenthese);
    assert(strcmp(p.EntreParenthese, "ER") == 0);
    Extraire_Hors_Parenthese(mot, p.HorsParenthese);
    assert(strcmp(p.HorsParenthese, "TEST") == 0);
    forme_mot(mot, p.MotForme);
    assert(strcmp(p.MotForme, "TESTER") == 0);

    Pioche joueur1;
    char teste1[NB_CHEVALETS_DEB + 1] = "CEEEENNNRSTU";
```

```
    initVecteur(&joueur1, NB_CHEVALETS_DEB);
    for (int i = 0; i < NB_CHEVALETS_DEB ; ++i) {
        ajouter(&joueur1, teste1[i]);
    }
    prelever(&joueur1, "CENE");

    char tmp[NB_CHEVALETS_DEB + 1 - 4 ]; // 4 = strlen("CENE")
    int i;
    for (i = 0; i < joueur1.nbElements; ++i)
        tmp[i] = obtenir(&joueur1, i);
    tmp[i] = '\0';
    assert(strcmp(tmp, "EENNRSTU") == 0);

    Pioche p2;
    initVecteur(&p2, NB_CHEVALETS_DEB);
    mettre_a_jour(&joueur1, &p2);
    for (i = 0; i < p2.nbElements; ++i)
        assert(obtenir(&joueur1, i) == obtenir(&p2, i));


    detruireVecteur(&joueur1);
    detruireVecteur(&p2);
}
```

## Mots.h

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <assert.h>

typedef struct {
    char** mots;
    int nbMots;
    int capacite;
} Mots;

enum {
   // pour COMPO_LETTRES_DEPART
     A = 9
    ,B = 1
    ,C = 2
    ,D = 3
    ,E = 14
    ,F = 1
    ,G = 1
    ,H = 1
    ,I = 7
    ,J = 1
    ,K = 5
    ,L = 3
    ,M = 6
    ,N = 0
    ,O = 5
    ,P = 2
    ,Q = 1
    ,R = 6
    ,S = 7
    ,T = 6
    ,U = 5
    ,V = 2
    ,W = 0
```

```
    ,X = 0
    ,Y = 0
    ,Z = 0
};


/**
 * @brief Initialise la structure contenant des mots .
 * @param[in] mots La structure contenant les mots qui seront joués à
initialiser.
 * @param[in] m Premier mot à ajouter.
 */
void initMots(Mots* m, const char* mot, int capacite);

/**
 * @brief Ajoute un mot à une structure les mots si celui-ci n'a pas déjà été
utilisé.
 * @param[in] m La structure contenant les mots déjà utilisés.
 * @param[in] mot Le mot à ajouter.
 * @return 1 si le mot a bien été ajouté, 0 sinon.
 */
int ajouterMot(Mots* m, const char* mot);

/**
 * @brief Verifie si un mot a déjà été joué.
 * @param[in] m La structure contenant des mots à traiter.
 * @param[in] mot Le mot à verifier.
 * @return 1 si le mot a déjà été joué , 0 sinon.
 */
int MotConnu(const Mots* m, const char* mot);

/**
 * @brief Libère toute la mémoire allouée dans la structure Mots et réinitialise
ses champs.
 * @param[in/out] m Pointeur vers la structure Mots à détruire.
 *
 * @note Après l'appel à cette fonction, le tableau de mots sera vide et `m-
>mots` sera
 *       mis à NULL, tandis que `m->nbMots` sera réinitialisé à 0.
 */
void detruireMots(Mots* m);

/**
 * @brief Teste les fonctions spécifiques au composant Mots.
 */
void testMots();
```

## mots.c

```
#pragma warning (disable : 4996)

#pragma once
#include "mots.h"

void initMots(Mots* m, const char* mot, int capacite) {
    assert(capacite > 0);
    m->capacite = capacite;  // Initialiser la capacité
    m->mots = (char**)malloc(sizeof(char*) * capacite);  // Allouer de l'espace
pour "capacite" mots
    if (mot != NULL && m->mots != NULL) {  // Si un mot est fourni, on l'ajoute
        m->mots[0] = (char*)malloc(sizeof(char) * (strlen(mot) + 1));
        if(m->mots[0]!= NULL) strcpy(m->mots[0], mot);
```

```
        m->nbMots = 1;
    }
    else {
        m->nbMots = 0;  // Aucun mot initial
    }
}

int MotConnu(const Mots* m, const char* mot) {
    int debut = 0;
    int fin = m->nbMots - 1;

    while (debut <= fin) {
        int milieu = debut + (fin - debut) / 2;
        int cmp = strcmp(m->mots[milieu], mot);

        if (cmp == 0) {
            return 1;  // Mot trouvé :)
        }
        else if (cmp < 0) {
            debut = milieu + 1;  // Chercher dans la moitié droite
        }
        else {
            fin = milieu - 1;  // Chercher dans la moitié gauche
        }
    }

    return 0;  // Mot non trouve :(
}

int ajouterMot(Mots* m, const char* mot) {
    if (!MotConnu(m, mot)) {  // Vérifier si le mot est déjà présent
        if (m->nbMots >= m->capacite) {  // Agrandir la capacité si nécessaire
            int nouvelleCapacite = m->capacite * 2;
            char** nouveauTableau = (char**)realloc(m->mots, sizeof(char*) *
nouvelleCapacite);
            if (nouveauTableau == NULL) {
                return 0;  // Erreur d'allocation
            }
            m->mots = nouveauTableau;
            m->capacite = nouvelleCapacite;
        }

        // Allocation et copie du nouveau mot
        m->mots[m->nbMots] = (char*)malloc(sizeof(char) * (strlen(mot) + 1));
        if (m->mots[m->nbMots] == NULL) {
            return 0;  // Erreur d'allocation pour le mot
        }

        strcpy(m->mots[m->nbMots], mot);  // Copier le mot
        m->nbMots++;  // Incrémenter le nombre de mots
        return 1;  // Succès
    }
    return 0;  // Mot déjà utilisé
}

void detruireMots(Mots * m) {
    for (int i = 0; i < m->nbMots; ++i) {
        free(m->mots[i]);
    }
    free(m->mots);
    m->mots = NULL;
    m->nbMots = 0;
    m->capacite = 0;
```

```
}

void testMots() {
    Mots m;
    initMots(&m, "un", 10);
    assert(MotConnu(&m, "un") == 1);
    assert(MotConnu(&m, "deux") == 0);

    ajouterMot(&m, "deux");
    assert(MotConnu(&m, "deux") == 1);

    detruireMots(&m);
}
```

## pioche.h

```
#pragma once
#include <stdio.h>
#include <stdlib.h>
#include "vecteur.h"
#include "mots.h"

typedef Vecteur Pioche;

enum { NB_CHEVALETS_DEB = 12, NB_MAX_CHEVALETS = 88, NB_LETTRES_DEPART = 26,
NB_LETTRES_MOT = 4, NB_LETTRES_RAIL = 8};
extern const int LETTERS[NB_LETTRES_DEPART];

/**
 * @brief Choisis aléatoirement un chevalet de la pioche.
 * @param[in] COMPO_LETTRES_DEPART La pioche contenant les chevalets.
 * @return Un chevalet de la pioche.
 */
ItemV lettreAleatoire(int* COMPO_LETTRES_DEPART);

/**
 * @brief Initialise le jeu d'un joueur.
 * @param[in,out] j La pioche du joueur.
 * @param[in] nb_chevalets Le nombre de chevalets que devra avoir le joueur.
 * @param[in] COMPO_LETTRES_DEPART La pioche contenant les chevalets.
 */
void initJeu(Pioche* j, int nb_chevalets, int* COMPO_LETTRES_DEPART);

/**
 * @brief Insere un élément dans une pioche. Cet élément est ajouté au bon
endroit (garder l'ordre alphabétique).
 * @param[in,out] f L'adresse de la pioche.
 * @param[in] it L'élément devant être ajouté.
 * @return 0 en cas d'échec (manque de mémoire disponible) et 1 en cas de succès.
 */
int inserer(Pioche* f, ItemV it);

/**
 * @brief Affiche les chevalets
 * @param[in,out] j La pioche du joueur.
 * @param[in] nb_chevalets Le nombre de chevalets que devra avoir le joueur.
 * @param[in] COMPO_LETTRES_DEPART La pioche contenant les chevalets.
 */
void affiche(Pioche* p);

/**
```

```
 * @brief Cherche la position d'une lettre dans l'alphabet.
 * @param[in] c La lettre dont on cherche la position.
 * @return L'indice correspondant à la position de la lettre dans l'alphabet.
 */
int indexLettre(char c);

/**
 * @brief Vérifie si un joueur a les lettres necessaires pour former un mot.
 * @param[in] mot Le mot que l'on va vérifier.
 * @param[in] p Le joueur dont on va accéder à ses chevalets pour vérifier si il
peut composer le mot.
 * @return 1 si c'est possible de composer le mot et 0 sinon.
 */
int verifMot(char* mot, const Pioche* p);

/**
 * @brief Demande à un joueur de saisir un mot compatbile aux règles et ne
s'arrête pas tant que cette condition n'est pas validée.
 * @param[in] j Le jeu du joueur qui saisit son mot.
 */
void saisieMot(Pioche* j, Mots* dict, char* mot, int numJoueur, Mots* m);

/**
 * @brief Ajoute le mot saisi par un joueur dans le rail.
 * @param[in] mot Le mot que l'on ajoute au rail.
 * @param[in, out] rail Le rail dans lequel on ajoute le mot saisi.
 */
void ajoute(char* mot, Pioche* rail);

/**
 * @brief Ajoute le mot saisi par un joueur au début du rail.
 * @param[in] mot Le mot que l'on ajoute au rail.
 * @param[in, out] rail Le rail dans lequel on ajoute le mot saisi.
 */
void ajouteDeb(char* mot, Pioche* rail);

/**
 * @brief Inverse les lettres présenentes sur le rail.
 * @param[in] j Le jeu du joueur qui saisit son mot.
 */
void inverseRail(Pioche* rail, Pioche* railInverse);

/**
 * @brief Affiche la situation courrante du jeu.
 * @param[in] j1 Le jeu du joueur 1.
 * @param[in] j2 Le jeu du joueur 2.
 * @param[in] r Le rail.
 * @param[in] rI Le rail inversé.
 */
void situation(Pioche* j1, Pioche* j2, Pioche* r, Pioche* rI);

/**
 * @brief Vérifie les lettres aux extrémités d'un rail.
 * @param[in] rail Le rail auquel on verifie l'extrémité.
 * @param[in] motExtrait Mot dont on a extirpé les parenthèses à verifier.
 * @param[in] codeEntre Chiffre indiquant quel coté du rail  verifier ( 1 =
gauche , 2 = droite ).
 * @return 1 si le mot peut bien être formé et 0 sinon.
 */
int verifier_lettres_extremite( const Pioche* rail,const char* motExtrait, int
codeEntre);

/**
```

```
 * @brief Verifie : si le mot et conforme à la longueur max (8),
 * si le mot existe dans le dictionnaire,
 * si le mot a déjà été joué.
 * @param[in] mot_joue Liste contenant les mots déja joués.
 * @param[in] motExtrait Mot formé à verifier
 * @param[in] dict Structure contenant le dictionnaire.
 * @return 1 si le mot est conforme à toutes les conditions et 0 sinon.
 */
int verifier(const Mots* mot_joue, const char* motExtrait,const Mots* dict);

/**
 * @brief Transfere des chevalets de rail à un joueur.
 * @param[in] rail Le rail dont on extrait les chevalets.
 * @param[in, out] joueur Le joueur qui va recevoir les chevalets
 * @param[in] codeEntre Chiffre indiquant quel coté du rail traiter ( 1 = gauche
, 2 = droite ).
 * @param[in] nb_lettres_a_transferer Le nombre de letteres à transferer.
 */
void transferer(Pioche* rail, Pioche* joueur, int codeEntre, int
nb_lettres_a_transferer);

/**
 * @brief Verifie si une lettre est présente dans le jeu d'un joueur.
 * @param[in] pioche La pioche du jouer dont on va vérifier la présence de la
lettre.
 * @param[in] lettre Le chevalet on dont vérifie la présence.
 * @return 0 si la lettre est absente et 1 si elle est présente dans le jeu.
 */
int verifieLettre(Pioche* pioche, char lettre);

/**
 * @brief Echange le chevalet du joueur avec un de la pioche aléatoirement.
 * @param[in, out] joueur Le joueur dont on va échanger le chevalet.
 * @param[in] lettre Le chevalet qu'on change avec un autre de la pioche.
 * @param[in, out] pioche La pioche contenant les chevalets.
 */
void echangeChevalet(Pioche* joueur, ItemV lettre, int* pioche);

/**
 * @brief Ajoute le mot saisi par un joueur dans le rail qui est vide.
 * @param[in] mot Le mot que l'on ajoute au rail.
 * @param[in, out] rail Le rail dans lequel on ajoute le mot saisi.
 */
void ajoute1(char* mot, Pioche* rail);

/**
 * @brief Teste les fonctions spécifiques au composant Pioche.
 */
void testPioche();
```

## pioche.c

```
#pragma once
#include "pioche.h"

#pragma warning(disable : 4996 6031)

const char LETTRES[NB_LETTRES_DEPART] = { 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H',
'I', 'J', 'K', 'L', 'M' , 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W' ,'X'
,'Y' ,'Z' };
```

```
char lettreAleatoire(int* COMPO_LETTRES) {
    int ind;

    do {
        ind = rand() % NB_LETTRES_DEPART;
    } while (COMPO_LETTRES[ind] == 0);

    --COMPO_LETTRES[ind]; //Je retire le chevalet du stock

    return LETTRES[ind];
}

int inserer(Pioche* f, ItemV it) {
    int sortie = ajouter(f, it);
    for (int i = f->nbElements - 1; i > 0; --i) {
        ItemV a = obtenir(f, i);
        ItemV b = obtenir(f, i - 1);
        if (a < b) {
            modifier(f, i - 1, a);
            modifier(f, i, b);
        }

    }
    return sortie;
}

void initJeu(Pioche* j, int nb_chevalets, int* COMPO_LETTRES_DEPART) {
    initVecteur(j, nb_chevalets);
    for (int i = 0; i < nb_chevalets; ++i) {
        inserer(j, lettreAleatoire(COMPO_LETTRES_DEPART));
    }

}

void affiche(Pioche* p) {
    for (int i = 0; p->nbElements > i; ++i) {
        printf("%c", obtenir(p, i));
    }
    printf("\n");
}

int indexLettre(char c) {
    return c - 'A';  // Se sert du code ASCII
}

int verifMot(char* mot, const Pioche* p) {
int cptPioche[NB_LETTRES_DEPART] = { 0 };  // Tableau pour compter les lettres
dans la pioche
    int cptMot[NB_LETTRES_DEPART] = { 0 };     // Tableau pour compter les
lettres dans le mot

    // Comptabiliser les lettres dans la pioche
    for (int k = 0; k < p->nbElements; ++k) {
        char lettre = obtenir(p, k);
        cptPioche[indexLettre(lettre)]++;
    }

    // Comptabiliser les lettres dans le mot
    size_t n = strlen(mot);
    for (int i = 0; i < n; ++i) {
        char lettre = mot[i];
```

```
        cptMot[indexLettre(lettre)]++;
    }

    // Vérifier si la pioche a assez de lettres pour chaque lettre du mot
    for (int i = 0; i < NB_LETTRES_DEPART; ++i) {
        if (cptMot[i] > cptPioche[i]) {
            return 0;
        }
    }

    return 1;
}

void saisieMot(Pioche* j, Mots* dict, char* mot, int numJoueur, Mots* m) {
    do {
        printf("%d> ", numJoueur);
        scanf("%s", mot);

    } while (strlen(mot) != NB_LETTRES_MOT || MotConnu(dict, mot) == 0 ||
verifMot(mot, j) == 0 || MotConnu(m, mot) == 1);

}

void ajoute1(char* mot, Pioche* rail) {
    for (int i = 0; i < NB_LETTRES_MOT; ++i) {
        ajouter(rail, mot[i]);
    }
}

void ajoute(char* mot, Pioche* rail) {

    int n = (int)strlen(mot);
    // Décale les lettres existantes vers la gauche pour faire de la place à la
fin.
    for (int i = 0; i < NB_LETTRES_RAIL - n; ++i) {
        modifier(rail, i, obtenir(rail, i + n));
    }

    // Place le mot à la fin du rail.
    for (int i = 0; i < n; ++i) {
        modifier(rail, NB_LETTRES_RAIL - n + i, mot[i]);
    }
}

void ajouteDeb(char* mot, Pioche* rail) {
    int n = (int)strlen(mot);
    for (int i = NB_LETTRES_RAIL - 1; i >= n; --i) {
        modifier(rail, i, obtenir(rail, i - n));
    }

    // Place le mot au début du rail en écrasant les premières lettres.
    for (int i = 0; i < n; ++i) {
        modifier(rail, i, mot[i]);
    }
}

void inverseRail(Pioche * rail, Pioche * railInverse) {
    for (int j = rail->nbElements - 1; j >= 0; --j) {
        ajouter(railInverse, obtenir(rail, j));
    }

}
```

```
void situation(Pioche* j1, Pioche* j2, Pioche* r, Pioche* rI) {
    printf("1 : ");
    affiche(j1);
    printf("2 : ");
    affiche(j2);
    printf("R : ");
    affiche(r);
    printf("V : ");
    affiche(rI);
}



int verifier_lettres_extremite(const Pioche* rail, const char* motExtrait, int
codeEntre) {
int len_motExtrait = (int)strlen(motExtrait);
    if (taille(rail) < len_motExtrait) return 0;
    if (codeEntre == 1) {
        for (int i = 0, j = 0; i < len_motExtrait; ++i, ++j) {
            if (motExtrait[i] != obtenir(rail, j))
                return 0;
        }
    }
    else if (codeEntre == 2) {
        for (int i = len_motExtrait - 1, j = taille(rail) - 1; i >= 0; --i, --j)
{
            if (motExtrait[i] != obtenir(rail, j))
                return 0;
        }
    }
    return 1;
}

int verifier(const Mots* mot_joue, const char* motExtrait, const Mots* dict){
if (strlen(motExtrait) > 8) return 0;
    if (!MotConnu(dict, motExtrait)) return 0;
    if (MotConnu(mot_joue, motExtrait)) return 0;
    return 1;
}

void transferer(Pioche* rail, Pioche* joueur, int codeEntre, int
nb_lettres_a_transferer) {
    if (codeEntre == 1) {
        int tmp = taille(rail);
        for (int i = 0, j = taille(rail) - 1; i < nb_lettres_a_transferer; ++i, -
-j) {
            inserer(joueur, obtenir(rail, j));
            //supprimer(rail, taille(rail) - 1);
        }
        //rail->nbElements = tmp;
    }
    else if (codeEntre == 2) {
        for (int i = nb_lettres_a_transferer - 1, j = 0; i >= 0; --i, ++j) {
            inserer(joueur, obtenir(rail, j));
            //supprimer(rail, 0);
        }
    }
}

int verifieLettre(Pioche* pioche, char lettre) {
    for (int i = 0; i < pioche->nbElements; ++i) {
        if (pioche->elements[i] == lettre) {
            return 1;
```

```
        }
    }
    return 0;
}

void echangeChevalet(Pioche* joueur, ItemV lettre, int* pioche) {
    char nvChevalet = lettreAleatoire(pioche);
    int i;
    for (i = 0; i < taille(joueur); ++i) {
        if (obtenir(joueur, i) == lettre) {
            break;
        }
    }
    supprimer(joueur, i);
    inserer(joueur, nvChevalet);
    int index = indexLettre(lettre);
    pioche[index]++;

}

void testPioche() {
    int COMPO_LETTRES_DEPART[NB_LETTRES_DEPART] = { 9, 1, 2, 3, 14, 1, 1, 1, 7,
1, 5, 3, 6, 0, 5, 2, 1, 6, 7, 6, 5, 2, 0, 0, 0, 0 };
    Pioche p;
    initJeu(&p, 1, COMPO_LETTRES_DEPART);
    modifier(&p, 0, 'E');
    assert(verifieLettre(&p, 'E') == 1);

    ajoute("AS", &p);
    assert(taille(&p) == 3);
    assert(indexLettre('D') == 4);

    Pioche p2;
    initVecteur(&p2, 8);
    inverseRail(&p, &p2);
    assert(p2.elements[0] == 'S');
    assert(verifMot("SAE", &p2) == 1);

    ajouteDeb("RENDU", &p2);
    assert(verifMot("RENDUSAE", &p2) == 1);
    assert(p.elements[0] == 'R');
    ajoute("RAPPORT", &p);
    assert(verifMot("RAPPORT", &p) == 1);

    transferer(&p2, &p, 1, 5);
    assert(verifMot("RENDU", &p) == 1);
    echangeChevalet(&p, 'R', COMPO_LETTRES_DEPART);
    assert(p.elements[0] != 'R');

    detruireVecteur(&p);
    detruireVecteur(&p2);
}
```

## vecteur.h

```
#pragma once

typedef char ItemV;
/**
 * @brief Conteneur stockant des éléments accessibles en
```

```c
 * fonction de leur position (indice).
 */
typedef struct {
      int nbElements; ///< Nombre d'éléments présents dans le vecteur.
      int capacite;      ///< Nombre d'éléments maximal du vecteur.
      ItemV* elements; ///< Tableau (dynamique) de taille <code>capacite</code>.
} Vecteur;

/**
 * @brief Initialise un vecteur d'une capacité donnée contenant aucun élément.
 * Après son utilisation, la mémoire occupée par un vecteur doit être libérée
 * en invoquant la fonction @ref detruireVecteur.
 * @param[out] v L'adresse du vecteur à initialiser.
 * @param[in] capacite La capacité initiale du vecteur.
 * @return 0 en cas d'échec (manque de mémoire disponible) et 1 en cas de succès.
 * @pre <code>capacite</code> doit être supérieur ou égal à 1.
 */
int initVecteur(Vecteur* v, int capacite);

/**
 * @brief Retourne le nombre d'éléments présents dans un vecteur.
 * @param[in] v L'adresse du vecteur.
 * @return Le nombre d'éléments contenu dans <code>v</code>.
 */
int taille(const Vecteur* v);

/**
 * @brief Ajoute un élément dans un vecteur. Cet élément est ajouté après ceux
déjà présents.
 * @param[in,out] v L'adresse du vecteur.
 * @param[in] it L'élément devant être ajouté.
 * @return 0 en cas d'échec (manque de mémoire disponible) et 1 en cas de succès.
 */
int ajouter(Vecteur* v, ItemV it);

/**
 * @brief Retourne l'élément d'un vecteur se trouvant à une position donnée.
 * @param[in] v L'adresse du vecteur.
 * @param[in] i La position (i.e. l'indice).
 * @return L'élément de <code>v</code> se trouvant à l'indice <code>i</code>.
 * @pre La valeur de <code>i</code> doit être comprise entre 0 et
 * <code>(taille(v) - 1)</code> (inclus).
 */
ItemV obtenir(const Vecteur* v, int i);

/**
 * @brief Modifie un élément d'un vecteur.
 * @param[in,out] v L'adresse du vecteur.
 * @param[in] i La position (i.e. l'indice) de l'élément devant être modifié.
 * @param[in] it La nouvelle valeur de l'élément.
 * @pre La valeur de <code>i</code> doit être comprise entre 0 et
 * <code>(taille(v) - 1)</code> (inclus).
 */
void modifier(Vecteur* v, int i, ItemV it);

/**
 * @brief Supprime un élément d'un vecteur.
 * @param[in,out] v L'adresse du vecteur.
 * @param[in] i La position (i.e. l'indice) de l'élément devant être supprimé.
 * @pre La valeur de <code>i</code> doit être comprise entre 0 et
 * <code>(taille(v) - 1)</code> (inclus).
 */
void supprimer(Vecteur* v, int i);
```

```c
/**
 * @brief Modifie la capacité courante d'un vecteur. Si la nouvelle capacité du
vecteur
 * excède le nombre d'éléments qu'il contient, tous ses éléments sont conservés.
Dans
 * le cas contraire, les éléments excédentaires sont perdus.
 * @param[in,out] v L'adresse du vecteur.
 * @param[in] taille La nouvelle capacité.
 * @return 0 en cas d'échec (manque de mémoire disponible) et 1 en cas de succès.
 * @pre La nouvelle capacité doit être supérieure ou égale à 1.
 */
int retailler(Vecteur* v, int taille);

/**
 * @brief Libère l'espace mémoire occupé par un vecteur. Après avoir été détruit,
il ne doit
 * pas être ré-employé sans avoir été ré-initialisé. Toute autre opération peut
donner des
 * résultats incohérent ou même provoquer l'arrêt brutal du programme.
 * @param[in,out] v L'adresse du vecteur.
 */
void detruireVecteur(Vecteur* v);

/**
 * @brief Renvoi le dernier element ajouté
 * @param[in,out] v L'adresse du vecteur.
 * @pre Le vecteur doit avoir au moins 1 element  .
 */
ItemV dernier(Vecteur* v);

/**
 * @brief Teste les fonctions spécifiques au composant Vecteur.
 */
void testVecteur();
```

## vecteur.c

```c
#include <assert.h>
#include <stdlib.h>
#include "vecteur.h"

int initVecteur(Vecteur* v, int capacite) {
	assert(capacite > 0);
	v->capacite = capacite;
	v->nbElements = 0;
	v->elements = (ItemV*)malloc(sizeof(ItemV) * capacite);
	return v->elements != NULL;
}

int taille(const Vecteur* v) {
	return v->nbElements;
}

int ajouter(Vecteur* v, ItemV it) {
	const int FACTEUR = 2;
	if (v->nbElements == v->capacite) {
		ItemV* tab = (ItemV*)realloc(v->elements, sizeof(ItemV) * v-
>capacite * FACTEUR);
		if (tab == NULL)
```

```
                return 0;
            v->capacite *= FACTEUR;
            v->elements = tab;
        }
        v->elements[v->nbElements++] = it;
        return 1;
}

ItemV obtenir(const Vecteur* v, int i) {
        assert(i >= 0 && i < v->nbElements);
        return v->elements[i];
}

void modifier(Vecteur* v, int i, ItemV it) {
        assert(i >= 0 && i < v->nbElements);
        v->elements[i] = it;
}

void supprimer(Vecteur* v, int i) {
        assert(i >= 0 && i < v->nbElements);
        for (++i; i < v->nbElements; ++i)
            v->elements[i - 1] = v->elements[i];
        --v->nbElements;
}

int retailler(Vecteur* v, int taille) {
        assert(taille > 0);
        ItemV* tab = (ItemV*)realloc(v->elements, sizeof(ItemV) * taille);
        if (tab == NULL)
            return 0;
        v->elements = tab;
        v->capacite = taille;
        if (v->nbElements > taille)
            v->nbElements = taille;
        return 1;
}

void detruireVecteur(Vecteur* v) {
        free(v->elements);
}


ItemV dernier(Vecteur* v) {
        assert(taille > 0);
        return obtenir(v, taille(v) - 1);
}

void testVecteur() {
        Vecteur v;
        initVecteur(&v, 8);
        assert(taille(&v) == 0);

        ajouter(&v, 2);
        assert(obtenir(&v, 0) == 2);
        modifier(&v, 0, 3);
        assert(obtenir(&v, 0) == 3);

        ajouter(&v, 10);
        assert(taille(&v) == 2);
        supprimer(&v, 0);
        assert(obtenir(&v, 0) == 10);

        detruireVecteur(&v);
```

```
}
```

## main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <assert.h>

#pragma once
#include "input.h"
#pragma warning(disable : 4996 6031)

enum{FORMAT = 4};
const char format[FORMAT] = "%29s";

void initRandom() {
    srand((unsigned int)time(NULL));
}

int main(){
    char mot1[NB_LETTRES_MOT + 1];
    char mot2[NB_LETTRES_MOT + 1];
    Mots mot_Joue;
    Mots dict;
    FILE* f = fopen("ods4.txt", "r");
    if (f == NULL) {
        printf("fichier non accessible\n");
        return;
    }
    if (ferror(f)) {
        printf("erreur - %s\n", strerror(errno));
        return ;
    }

    char mot[30] = { 0 };
    int n;
    n = fscanf(f, format, mot);
    initMots(&dict, mot, 500);
    n = fscanf(f, format, mot);
    while (n == 1) {
        ajouterMot(&dict, mot);
        n = fscanf(f, format, mot);
    }
    fclose(f);

    int COMPO_LETTRES_DEPART[NB_LETTRES_DEPART] = { A, B, C, D, E, F, G, H, I, J,
K, L, M, O, P, Q, R, S, T, U, V, W ,X ,Y ,Z };
    Pioche joueur1, joueur2, rail;

    initRandom(COMPO_LETTRES_DEPART);
    initJeu(&joueur1, NB_CHEVALETS_DEB, COMPO_LETTRES_DEPART);
    initJeu(&joueur2, NB_CHEVALETS_DEB, COMPO_LETTRES_DEPART);


    initVecteur(&rail, NB_LETTRES_RAIL);
    printf("Joueur 1: ");
    affiche(&joueur1);
    printf("Joueur 2: ");
    affiche(&joueur2);
    printf("\n");
```

```
    int premier_a_saisir;

    saisieMot(&joueur1, &dict, mot1, 1, &mot_Joue);
    initMots(&mot_Joue, mot1, 50);
    prelever(&joueur1, mot1);

    saisieMot(&joueur2, &dict, mot2, 2, &mot_Joue);
    ajouterMot(&mot_Joue, mot2);
    prelever(&joueur2, mot2);

    if (strcmp(mot1, mot2) <= 0) {
        ajoute1(mot1, &rail); ajoute1(mot2, &rail);
        premier_a_saisir = 1;
    }
    else {
        ajoute1(mot2, &rail); ajoute1(mot1, &rail);
        premier_a_saisir = 2;
    }

    Pioche railInverse;
    initVecteur(&railInverse, NB_LETTRES_RAIL);
    inverseRail(&rail, &railInverse);

    //-------------------------------------------------------------------------
-----------//
    int joueur = premier_a_saisir;
    printf("\n");

    situation(&joueur1, &joueur2, &rail, &railInverse);
    printf("\n");
    Pioche railtmp;
    Pioche railInversetmp;
    Pioche joueur1tmp;
    Pioche joueur2tmp;
    initVecteur(&railtmp,NB_LETTRES_RAIL);    initVecteur(&railInversetmp,
NB_LETTRES_RAIL); initVecteur(&joueur1tmp, NB_LETTRES_RAIL);
initVecteur(&joueur2tmp, NB_LETTRES_RAIL);

    fflush(stdin); // nettoie le buffer au cas où
    while (joueur1.nbElements > 0 && joueur2.nbElements > 0) {

        char commande;
        printf("%d> ", joueur);

        scanf(" %c", &commande);

        Input inputJoueur;

        if (commande == 'r' || commande == 'v') {
            int codeEntre = saisieMotJoueur(joueur == 1 ? &joueur2tmp :
&joueur1tmp,
                &railtmp, &railInversetmp, &dict, &inputJoueur, &mot_Joue, joueur
, &commande);
            printf("\n");
            if (codeEntre != 0)
            debarasser((joueur == 1 ? &joueur1 : &joueur2), joueur);
            printf("\n");
            situation(&joueur1, &joueur2, &rail, &railInverse);
            printf("\n");

            printf("%d> ", joueur);

            scanf(" %c", &commande);
```

```
        }
        if (commande == 'R' || commande == 'V') {

            int codeEntre = saisieMotJoueur(joueur == 1 ? &joueur1 : &joueur2,
                &rail, &railInverse, &dict, &inputJoueur, &mot_Joue, joueur,
&commande);
            if (codeEntre != 0) {

                //----------sauvegarde la situation du jeu----------//
                mettre_a_jour(&rail, &railtmp);
                mettre_a_jour(&railInverse ,&railInversetmp);
                mettre_a_jour(&joueur1,&joueur1tmp );
                mettre_a_jour(&joueur2 ,&joueur2tmp);
                //----------sauvegarde la situation du jeu----------//


                ajouterMot(&mot_Joue, inputJoueur.MotForme);
                transferer(commande == 'R' ? &rail : &railInverse, joueur == 2 ?
&joueur1 : &joueur2, codeEntre, (int)strlen(inputJoueur.HorsParenthese));
                codeEntre == 1 ? ajouteDeb(inputJoueur.HorsParenthese, commande
== 'R' ? &rail : &railInverse) : ajoute(inputJoueur.HorsParenthese, commande ==
'R' ? &rail : &railInverse);
                prelever(joueur == 1 ? &joueur1 : &joueur2,
inputJoueur.HorsParenthese);


                // mettre à jour les rails du jeu
                if (commande == 'V') {
                    for (int i = 0, n = NB_LETTRES_RAIL - 1; i < NB_LETTRES_RAIL
&& n >= 0; ++i, --n) {
                        modifier(&rail, i, obtenir(&railInverse, n));

                    }
                }
                else
                {
                    for (int i = 0, n = NB_LETTRES_RAIL - 1; i < NB_LETTRES_RAIL
&& n >= 0; ++i, --n) {
                        modifier(&railInverse, i, obtenir(&rail, n));
                    }
                }

                if (strlen(inputJoueur.MotForme) == 8 && ( ( joueur == 1  ?
joueur1 : joueur2 ).nbElements > 0)) {
                    printf("\n");
                    situation(&joueur1, &joueur2, &rail, &railInverse);
                    printf("\n");
                    debarasser((joueur == 1 ? &joueur1 : &joueur2), joueur);
                }
                if (joueur1.nbElements > 0 && joueur2.nbElements > 0) {
                    printf("\n");
                    situation(&joueur1, &joueur2, &rail, &railInverse);
                    printf("\n");
                }
            }
        }
        else if (commande == '-') {
            ItemV lettre;
            do {
                scanf("%c", &lettre);
            } while (!verifieLettre(joueur == 1 ? &joueur1 : &joueur2, lettre));
```

```
            echangeChevalet(joueur == 1 ? &joueur1 : &joueur2, lettre,
COMPO_LETTRES_DEPART);
            printf("\n");
            situation(&joueur1, &joueur2, &rail, &railInverse);
            printf("\n");
        }

        if (commande == 'r' || commande == 'v' || commande == '-' || commande ==
'R' || commande == 'V') {
            joueur = (joueur == 1 ? 2 : 1);
        }
    }
    detruireMots(&dict);
    detruireMots(&mot_Joue);

    detruireVecteur(&rail);
    detruireVecteur(&railInverse);
    detruireVecteur(&joueur1);
    detruireVecteur(&joueur2);

    detruireVecteur(&railtmp);
    detruireVecteur(&railInversetmp);
    detruireVecteur(&joueur1tmp);
    detruireVecteur(&joueur2tmp);
}
```