Institut Galilée
UNIVERSITÉ PARIS 13

UNIVERSITÉ PARIS 13

Projet de programmation impérative

Le jeu SHOGI

Réalisé par : KESSI Ferhat, MEZOUANI Sofiane & TALBI Katia

Informatique Licence 1, Semestre 2

Groupe 5 et 7_ Section A

Rapport de projet de programmation impérative

Enseignants (es) M.Sdiri et M.Rikotonarivo

Année 2016/2017

Adresse : 99 av. Jean-Baptiste Clément, 93430 Villetaneuse

Sommaire

Introduction

Pour notre premier projet en programmation impérative, on a eu la consigne de programmer en C le jeu Shôgi qui est une sorte de jeu d'échecs japonais. L'objectif de ce jeu de société traditionnel est de prendre le roi adverse et, tout comme le jeu d'échecs, la partie se termine lorsque le roi ne peut éviter d'être pris au coup suivant. On dit alors que le roi est « Mat ». L'échiquier du Shogi se nomme « Shôgi ban » ou tablier et comporte 81 cases. Les règles du jeu sont principalement les mêmes que le jeu d'échecs occidental. Les seuls points distincts entre ces deux jeux sont les deux règles supplémentaires : promotion d'une pièce et la possibilité de réutilisation de cette pièce ultérieurement.

Le but du projet est donc, d'écrire un programme proposant à un joueur plusieurs fonctionnalités tels que jouer une nouvelle partie (contre un autre joueur sur la même machine), sauvegarder la partie, continuer une partie déjà sauvegarder ou visualiser la partie.

I. Petite recherche sur Shôgi et répartition du travail

Pour commencer notre projet, on a voulu faire un plan de travail afin d'aborder notre sujet étape par étape.

La première étape était de bien comprendre la consigne et de rassembler les idées. On a donc commencé par réfléchir à ce que l'on sait déjà sur le jeu pour ensuite organiser ces idées tout en notant les points importants dont on n'avait aucune information. Après cela, on a attaqué les moteurs de recherche. On a visité plusieurs sites internet qui nous ont aidés à accueillir toutes les informations nécessaires.

Une fois qu'on a bien cerné notre sujet, on est passé à la deuxième étape : l'analyse et la déclaration ainsi que la définition des fonctions dont on avait besoin pour écrire un programme complet du jeu (qo'on va aborder au fur et à mesure dans ce rapport). Et comme on avait plusieurs fonctions et surtout qu'au début elles n'étaient pas très compliquées, nous nous sommes répartis les tâches afin de faciliter les recherches pour chaque membre de l'équipe. La répartition s'est faite comme suit : Sofiane s'est occupé de l'implémentation du code de la structure d'une pièce ainsi que les fonctions piece_creer, piece_jouer et piece_identifier, Ferhat s'est occupé de la définition de la structure de la liste des coups joués et celle d'un coup ainsi que le code des fonctions piece_caractere et case_vide. Katia s'est occupée de la définition des fonctions piece_afficher, modifier_case , changer_joueur et afficher_plateau.

Cependant, en avançant dans l'écriture du programme, les fonctions commençaient à nous paraître un peu plus complexes qu'au début. Et en travaillant chacun de son côté, on avait le sentiment qu'on avançait très lentement surtout qu'on n'avait pas la même vision pour chaque problème. Alors, pour éviter ce gros obstacle, on a décidé de travailler sur les fonctions un peu difficiles ensemble, tous les trois, comme ça on avait à chaque fois presque trois solutions ou trois idées différentes.

Malgré qu'on a employé cette méthode dans le codage de plusieurs fonctions telles que deplacement, annuler_deplacement, partie_sauvegarder, partie_charger ainsi que d'autres ; on ne l'a pas conservé par la suite, car en arrivant aux déplacements valides on a recommencé à coder les fonctions individuellement. Où Ferhat s'est chargé du deplacement_valide_tour, deplacement_valide_fou, deplacement_valide_horse (pour le déplacement d'un fou promu) et deplacement_valide_roi. Katia s'est chargée du deplacement_valide_cavalier, deplacement_valide_lance et du deplacement_valide_silver (pour le général d'argent). Quant à Sofiane, il s'est occupé des fonctions deplacement_valide_dragon (pour le déplacement d'une tour promue), deplacement_valide_pion et deplacement_valide_gold (pour le général d'or).

II. Les bases du programmes

1. Les pièces

Au début de la partie, chaque joueur dispose de 20 pièces : un roi, une tour, un fou, deux généraux d'or, deux généraux d'argent, deux cavaliers, deux lances, et neuf pions.

Comme on l'a déjà expliqué, dans cette partie on n'a pas rencontré de problèmes mis à part deux fonctions :

- ✓ **void changer_statut(piece_t *piece)** : au début on testait si la pièce est promue alors on change le caractère en utilisant un switch. A l'intérieur du switch on vérifie si le caractère représentant le type est majuscule pour qu'on teste que sur les pièces dont les caractères sont majuscules exemple : « case 'D': piece->type = 'P'; break ; ». Et on fait de même si le caractère est minuscule : « case 'd': piece->type='p'; break ; ». Et ce même travail on le répète avec les pièces non promues ce qui nous mène à faire une duplication du code. Pour cela, on a trouvé une solution optimale dont le principe est de louer de l'espace mémoire pour deux tableaux qu'on remplit qu'avec des caractères minuscules (les pièces appartenant au joueur 1) où l'un contiendra les pièces non promues (tab2) et l'autre les pièces promues (tab1), de telle sorte que l'indice d'une pièce avant promotion (dans tab2) est le même après sa promotion (dans tab1). Ensuite, on teste si la pièce donnée appartient au joueur 0 dans quel cas on parcourt les deux tableaux en changeant les caractères en majuscules et en sortant de la boucle on vérifie si la pièce est promue : on la cherche dans tab1 pour ensuite changer son type en lui affectant le caractère correspondant (après promotion) qui se trouve dans tab2 et on change son statut. On fait la même chose si la pièce est non promue sauf que cette fois ci on la cherche dans tab2 et on lui affecte le caractère correspondant de tab1.
  On n'oublie pas de libérer l'espace mémoire occupé pas les deux tableaux.

- ✓ **piece_t piece_identifier (char type**) : au début on a fait un switch case, mais comme pour la fonction changer_statut, cette idée nous semblait très couteuse car on dupliquait le code. Alors on a décidé de faire un algorithme plus simple. On alloue de l'espace pour un tableau de caractères afin de stocker les lettres minuscules qui représentent les pièces du joueur 1. Le principe de cette algorithme c'est de teste si le type rentré en paramètre est majuscule (if(isupper(type)), dans ce cas-là on va créer une pièce appartenant au joueur1. Alors on cherche, en modifiant les caractères en majuscules, dans la première partie du tableau (qui concerne les pièces non promues) si le type existe on crée et on retourne une pièce de ce type, sinon on termine la

recherche dans la deuxième partie du tableau (qui concerne les pièces promues) et on fait de même.
Dans le cas où le type rentré en paramètre est minuscule, on fait le parcours directement et tout ce qui suit sans changer les caractères.
On n'oublie pas de libérer l'espace mémoire.

2. La réserve de chaque joueur

Ici, la réserve fait partie du tablier artificiellement. Elle représente les cases de la ligne 0 et de la colonne 0 pour le joueur 0 (sauf les qui cases qui ont comme indice de ligne ou de colonne 10).

3. La liste des coups joués

La liste des coups joués « liste_coups_ joues » fait partie de la structure d'une partie, on y stock l'ensemble des coups joués.

4. La partie

Dans la fonction **case_vide**, on retourne VRAI si le type de la pièce donnée en paramètre est un point, FAUX sinon. Car, on représente la case vide par une pièce dont le type est le caractère point '.' (comme on l'a déjà expliqué dans le programme écrit dans le fichier partie.c).

Dans la fonction **modifier_case** on change juste la case d'arrivée par la pièce rentrée en paramètre. Donc, on change le type, le statut ainsi que le joueur à qui appartient la pièce se trouvant dans cette case.

Pour la fonction **partie_créer**, en plus de l'allocation de l'espace mémoire pour une partie « partie_t *partie = malloc(sizeof(partie_t))» on a rajouté quelques allocations dynamiques. Premièrement, on a alloué un tableau de coordonnées KING (déjà déclaré comme variable globale) constitué de deux cases, celle de l'indice 0 concerne le joueur 0 et celle de l'indice 1 concerne le joueur 1, pour stocker les coordonnées de départ et d'arrivée du roi après chaque déplacement car on en avait besoin dans d'autres fonctions (qu'on va voir un peu plus tard dans le rapport).
On a aussi alloué de l'espace pour une liste chainée LC (déclarée aussi comme variable globale). Cette liste nous sert à conserver les captures de chaque joueur; cela nous aidera ultérieurement dans l'écriture de la fonction replay_charger. Elle est de type liste_capture, qui est une nouvelle structure contenant la taille (un entier) et le début (un pointeur vers un maillon_capture qui est aussi une nouvelle structure qu'on a créé pour la manipulation de la liste LC).

maillon_capture contient la coordonnée de la pièce capturée « coord_capture » ainsi qu'un pointeur vers le maillon_capture suivant « suiv ».
On a, bien entendu, initialisé les deux liste (la liste des captures « LC » et la liste des coups joués « liste_coup_joues ») : leurs tailles à 0, les débuts à Null et la fin à Null également. Quant au tableau KING, on a initialisé ses deux cases à case vide.

Comme son nom l'indique, **partie_detruire** sert à détruire la partie c,à,d à libérer l'espace mémoire occupé par la partie (qu'on a alloué dans partie_creer). D'abord, pour libérer l'espace occupé par la liste des coups joués on a écrit une fonction « **extraire_fin_liste** » qui prend un pointeur vers une liste et extrait le dernier maillon de cette liste en décrémentant la taille, et on extrait le dernier coup de la liste tant que sa taille (nb_coup) est positive tout en libérant le maillon occupé par le coup qu'on a extrait.
A la fin de cette boucle, on aurait détruit la liste des coups joués, on appelle alors, la fonction detruire_liste_LC qui détruit la liste des captures.
detruire_liste_LC est une fonction qu'on a rajouté afin de détruire la liste des capture. Cette fonction extrait le début de la liste LC et le détruit tout en décrémentant la taille.
Après la destruction des listes, on libère l'espace occupé par la partie ainsi que le tableau KING. (free(partie); free(KING);).
Toutefois, cette fonction n'était pas la même au début, car avant de faire les fonctions deplacement et annuler_deplacement, on n'avait pas encore créé la liste des capture LC ni le tableau KING. Donc au début on a libéré l'espace occupé que par la liste des coups joués et de la partie.

On appelle la fonction **partie_nouvelle** pour créer une nouvelle partie en mettant les pièces dans le tablier à l'état initial du jeu. Ici aussi, on a alloué de l'espace mémoire pour stocker dans un tableau de caractères « chaine » les lignes de 1 à trois du tablier. On a rempli les lignes 0, 4, 5, 6, 7 et 10 avec des cases vides en utilisant la fonction piece_identifier. Et les lignes entre 0 et 4 (non compris), on les a remplis en même temps que les lignes entre taille-1 et taille-5 (non compris) en utilisant le tableau de caractères « chaine ». (voir la fonction partie_nouvelle dans le fichier partie.c).

Partie_jouer est la partie qui propose le menue à l'utilisateur et qui, en conséquence, fait appel aux fonctions test_coordonnées, deplaceent_valide, deplacement, annuler_deplacement, changer_joueur, partie_sauvegarder, replay_sauvegarder, en fonction de la réponse du joueur (voir fichier note.txt). C'est la fonction qu'on appelle si l'utilisateur n'a pas donné d'argument en exécutant le programme ou s'il a choisi de continuer une partie.

Le menu proposé à chaque tour est :
S'il n y a pas de coups déjà joués (liste_coup_joues est vide) : faire un déplacement ou quitter en indiquant le joueur qui tient la main.
S'il existe un ou plusieurs coups joués : faire un déplacement, annuler un déplacement ou quitter.
Tant que l'utilisateur ne choisit pas de quitter (réponse != 'q') on reste dans la fonction et on réaffiche le menu.

Si le joueur indique qu'il veut :

- Faire un déplacement : on appelle la fonction promotion (qu'on expliquera juste après la fonction partie_jouer) ensuite on appelle la fonction test_coordonnees et on effectue le déplacement après avoir vérifié qu'il est valide.
  Sans oublier de tester s'il s'agit du déplacement du roi, dans quel cas on vérifie si le déplacement effectué ne met pas le roi en échec, sinon on l'annuler et on affiche un message d'erreur (attention vous ne pouvez pas effectuer ce déplacement car votre roi sera en échec) et on change le joueur pour permettre au même joueur de faire un autre déplacement (car après chaque déplacement on change le joueur).
  S'il ne s'agit pas du roi, on affiche le tablier, on teste s'il y a une pièce capturée (la taille de LC différente de 0) on examine la pièce capturée. Si c'est le roi alors le joueur qui a fait le déplacement gagne la partie, on le met au courant en affichant un message et on sort de la fonction en mettant la réponse à 'q'.
  On vérifie encore une autre condition après avoir fait le déplacement. C'est de voir si la case d'arrivée ne mets pas le roi du joueur adverse en échec, sinon on affiche un message pour l'avertir.
  Dans tous ces cas précédents, on a fait le déplacement, donc on doit changer le joueur.

- Annuler un déplacement : tant qu'ils existent des coups joués et le joueur veut annuler, on change le joueur, on annule le déplacement et on affiche le plateau. On le fait deux fois, afin d'annuler le coup de l'adversaire et le sien. S'il n y a plus aucun coup joué on affiche un message indiquant qu'il ne peut plus annuler un déplacement.

- Quitter la partie : d'abord on demande à l'utilisateur s'il veut sauvegarder sa partie, son plateau de jeu ou juste quitter.
  S'il décide de sauvegarder le plateau, on alloue de l'espace mémoire afin de stocker le chemin de la sauvegarde. On demande ainsi le nom du fichier (dont on prend que 10 lettres) qu'on concatène avec le chemin en ajoutant l'extension « .plt ». Enfin, on fait appel à la fonction partie_sauvegarder.
  S'il décide de sauvegarder toute la partie jouée, on teste déjà s'il a continué une partie qui existe déjà dans un fichier (dans ce cas il a lancé l'exécution en donnant le nom du fichier), alors, on sauvegarde le nom de ce fichier avec l'extension « .part » dans l'espace qu'on alloue pour le chemin. S'il a commencé une nouvelle partie, on lui demande alors le nom du fichier dont il veut sauvegarder et on refait les mêmes étapes pour récupérer le nom et le chemin du fichier. A la fin, on appelle la fonction replay_sauvegarde qui se chargera de la suavegarde.

On n'oublie pas de libérer l'espace mémoire occupé par le chemin.

La fonction **promotion** a pour rôle de parcourir les cases des lignes dont on peut faire la promotion, et vérifie s'il y a des pièces qui peuvent être promues mais qui ne le sont pas. Elle demande alors au joueur s'il veut les promouvoir.

Pour le reste des fonctions dont on n'avait pas de problèmes, on les a commentés lors de leurs définitions dans le fichier partie.c où on a expliqué aussi le rôle de chacune

5. Les déplacements

Il faut dire que c'est pour les déplacements qu'on a consacré la majorité de notre temps. D'un côté, en essayant d'optimiser nos algorithmes le plus possible. D'un autre côté, plus on avançait dans les autres parties du jeu, plus on rencontrait des cas qui nous posait problèmes avec les versions déjà faites de notre déplacement.

Dans ce qui suit, on va exposer les points qui nous ont posé des problèmes et les solutions qu'on a utilisés pour y remédier.

La fonction **deplacement** :

```
70  void deplacement(partie_t *partie,coordonnees_t coord_depart,coordonnees_t coord_arrivee)
```

On a alloué de l'espace mémoire pour une variable qui pointe vers un coup qu'on utilisera pour ajouter, à chaque déplacement, le coup joué à la liste des coups liste_coups_joues. Et la première condition qu'on a vérifiée dans cette fonction était l'existence d'une pièce à capturer dans la case d'arrivée `75  if(!case_vide(partie->tablier[coord_arrivee.x][coord_arrivee.y]))`. Si la condition est vraie, cela signifie qu'on doit déplacer la pièce capturée dans la réserve. Donc, on a cherché une case vide dans la réserve du joueur qui a fait le déplacement. Les coordonnées de la case vide sont aux indices i et j mais on ne mettra la pièce dans la réserve qu'un peu plus tard. A priori, on change son propriétaire.

On a mis bool_capture à vrai ce qui signifie qu'on a une capture à ajouter à la liste des capture LC (dont on a déjà parlé). Et on a vérifié si c'est le roi qui a été capturé, on a donc, affiché un message pour dire que c'est la fin de la partie ! Après avoir traité tous les cas pour la capture on l'a déplacé dans la réserve (la case d'indices i et j) et on a changé son type en fonction de son nouveau propriétaire.

On a fait le déplacement de la pièce de la case de départ vers la case d'arrivée et on a vider sa case de départ.

La deuxième condition qu'on a vérifié est la promotion de la pièce après l'avoir déplacé. En effet, si elle n'est pas promue et elle n'est pas un des généraux d'or ou un des rois des deux joueurs, on a cherché si elle est déplacée vers une des lignes de promotions (les trois premières lignes pour le joueur 0 et les trois dernières lignes pour le joueur 1). Si c'est le cas, on a vérifié :

- si elle est l'une des pièces dont la promotion est obligatoire dès qu'elles sont dans les lignes de promotion dans ce cas, on a changé le statut de la pièce (promouvoir la pièce).
- sinon, on a demandé à l'utilisateur s'il veut la promouvoir directement, ou il préfère le faire plus tard.

On a mis bool_promotion à vrai.

Enfin, on a un nouveau coup à ajouter à la liste des coups. On a donc affecté à ce nouveau coup les booléens de la promotion et de la capture ainsi que les coordonnées de départ et d'arrivée de la pièce qu'on a déplacé.

Avant d'arriver à cette version finale de la fonction déplacement, on a rencontré quelques problèmes qu'on a voulu citer.
On a rencontré des problèmes quand on a compris qu'un joueur peut promouvoir une pièce (ou plusieurs) a n'importe quel moment tant que c'est son tour de jouer. Donc on a ajouté (traité) les promotions à l'intérieur de déplacement comme suit:

- après avoir fait un déplacement le joueur peut promouvoir sa pièce si elle est dans les trois premières cases de l'adversaire tout en vérifiant si c'est une pièce qui doit obligatoirement être promue (telle la lance) alors elle le sera sans qu'on demande l'avis du joueur.

Pour pouvoir annuler les promotions faites par le même joueur (un problème rencontré lors de l'écriture de la fonction annuler_deplacement), on a sauvegardé ces promotions dans la liste des coup_joués, et le principe était d'ajouter la promotion au début de la liste au fur et à mesure.
Par ailleurs, on a rajouté une fonction promotion_avant (qu'on appelle juste avant de faire un déplacement) pour regarder s'il y'a une ou plusieurs pièce(s) qui peuvent être promues et donc demander à l'utilisateur s'il veut les ou la promouvoir.

La fonction **annuler_deplacement** :

La première condition qu'on a testé était le nombre de coups joués. Si il n'y a pas plus d'un coup joué, on ne peut pas annuler le déplacement (car on annule deux déplacement à la fois : celui de l'adversaire ainsi que celui du joueur qui est en train d'annuler) donc on teste (if(partie >coups_joues.nb_coups > 1)) on détache le dernier maillon (coup) de l'avant dernier. Le champ fin (de la liste) reçoit l'adresse de l'avant dernier coup qui devient donc le dernier et on décrémente la taille de la liste.
On remet la pièce qui se trouve dans la case d'arrivée dans sa case de départ, et on vide la case d'arrivée.
La seconde condition à vérifier est la capture. S'il y'a eu une capture, on extrait le début de la liste des capture (LC) et on remet la pièce capturée à sa place dans le tablier sans oublier de vider l'emplacement (la case) de cette pièce dans la réserve.

. On change le propriétaire de la pièce après l'avoir remise à sa case de départ, ainsi que son type.

La troisième condition est la promotion. S'il y'a eu une promotion au cours du coup que l'on veut annuler, on change le statut de la pièce.

Quelques problèmes rencontrés avant d'arriver à la version finale (qu'on vient d'expliquer) :

- trouver l'emplacement de la dernière pièce capturée dans la réserve (le problème c'est le parachutage) donc pour régler cela, on a pensé à deux solutions qui sont : 1.parcourir la réserve inversement. 2.déclarer une liste de coordonnées GLOBALE (LC).

On a opté pour la deuxième !
D'autres sous problèmes qu'on a réussis à régler :

- détachement du maillon du dernier coup.
- déplacement de la pièce vers ses coordonner de départ.
- promotion (savoir s'il y'avait eu une promotion est l'annuler) problème des switch (on a remarqué qu'à chaque fois on doit vérifier la promotion des pièces donc on fait beaucoup de switch sur les types). Pour ce petit obstacle, on a pensé un écrire une fonction **changer_statut** qui prend une pièce change son statut en fonction de son type. Le principe est simple : on a testé si le caractère est majuscule ou minuscule (pour diminué le coup d'appelle) puis on a changé la promotion.

Nouveaux problèmes qu'on a rencontrés après avoir su qu'un joueur peut annuler plusieurs coups :
Comme on a déjà sauvegardé les promotions dans la liste des coups joués, on a qu'à annuler (extraire) les promotions jusqu'à ce qu'on arrive au coup joué pour l'annuler aussi, sachant qu'on différencie le coup de la promotion quand bool_joueur est différent du joueur qui a fait le dernier coup. L'annulation de ce coup se fait dans tous les cas même si il n'y a pas eu de promotions faites.

deplacement_valide_roi, la fonction qui valide (ou pas) le déplacement d'un roi, pour laquelle on donne les coordonnées de départ et celles d'arrivée ainsi qu'un pointeur vers la partie. On a distingué trois cas. Si le déplacement se fait dans l'une des cases qui suivent celle du roi et qui forment un carré autour d'elle (mais qui ne sont ni sur la même ligne ni sur la même colonne). Sinon, si la case d'arrivée et celle du départ sont sur la même colonne (ou sur la même ligne), on vérifie respectivement, que la ligne d'arrivée suit celle de départ (ou que la colonne d'arrivée suit celle du départ). Et si en plus la pièce qui se trouve à la case d'arrivée n'appartient pas au joueur qui fait le déplacement la fonction retourne vrai (une case vide a comme joueur -1).
On a utilisé exactement le même principe pour les autres pièces en fonction des cases vers lesquelles elles peuvent se déplacer. On a bien expliqué le rôle de ces fonctions (**deplacement_valide_cavalier**, **deplacement_valide_tour**, **deplacement_valide_horse**, etc).

Problèmes rencontrés et/ou solutions :

Pour les déplacements valides des deux généraux (d'argent et d'or) et celui du roi on a remarqué qu'il y'a des conditions communes pour que leurs déplacements soient valides, alors on a pensé à écrire des fonctions qui englobent chacune ces déplacements communs par exemple : le déplacement vers l'avant qu'on peut utiliser dans le déplacement valide des deux généraux (qui traitera les cas d'un déplacement d'une case vers l'avant, d'un déplacement d'une case vers l'avant gauche ou d'un déplacement d'une case vers l'avant droit). Mais finalement, on a trouvé dans le déplacement valide principal, qui réunit tous les déplacements

valides des pièces, il nous faut beaucoup de conditions pour l'appel de ces fonctions donc au lieu d'optimiser le programme on allait le rendre plus complexe (complexité en temps).

On a ajouté une fonction teste_coordonnees qu'on appelle au début de la fonction deplacement_valide afin de tester si les coordonnées données par l'utilisateur ne sortent pas du tablier (avec les deux réserves artificielles).

Pour sauvegarder les coordonnées d'arrivée lors du déplacement d'un roi, on a créé un tableau KING à deux cases de type coordonnées dont l'indice représente exactement le joueur qui fait le déplacement. et ce, afin de permettre à notre fonction parachutage (qu'on expliquera un peu plus tard dans le rapport) de vérifier si le roi n'est pas dans un état d'échec lors du parachutage d'une pièce.

En ce qui concerne la fonction **deplacement_valide**, d'abord on a devisé la fonction en deux grandes parties :

1. le cas d'un déplacement d'une pièce dans le tablier. On a commencé par voir si la pièce qu'on va déplacer est bien celle du joueur qui fait le déplacement. Et avec un switch sur le type de la pièce du départ on a déviser cette partie aussi en deux cas, le premier, pour les pièces promues, le second, pour celles qui ne le sont pas. Dans ces deux sous cas, on ertourne les fonctions déplacement valide correspondantes au type de la pièce. Sauf pour le roi, on vérifie d'abord si le déplacement est valide, on sauvegarde ses coordonnées d'arrivée dans le tableau KING et on retourne la valeur de son deplacement_valide_roi (VRAI ou FAUX).
2. le cas d'un parachutage, comme ce qu'on a fait avant, on vérifie que la pièce appartient au joueur qui fait le déplacement. Après cela, on s'assure que la case d'arrivée est vide (car dans un parachutage on ne capture pas de pièces). On retourne la fonction parachutage qui fait le déplacement.

Pour la fonction **parachutage** le principe est très simple (car on a déjà traité le cas ordinaire, qui est de déplacer une pièce de la réserve au tablier, dans la fonction déplacement), il se résume en ce qui suit :
Cas du parachutage d'un pion. On le traite en premier : on vérifie s'il n'y a aucun pion non_promue du joueur adverse dans la colonne d'arrivée, si c'est le cas, la fonction rend la valeur FAUX.
Dans tous les cas, on appelle la fonction déplacement pour parachuter la pièce.
On vérifie que cela ne mets pas le roi en échec, si c'est le cas, on annule le déplacement et la fonction retourne la valeur FAUX indiquant que le parachutage ne peut pas avoir lieu.
A la fin, on change le joueur et on affiche le plateau.

3. Enregistrer

Cette partie du projet sollicite de donner à l'utilisateur la possibilité de sauvegarder le plateau d'une partie dans un fichier spécifique (qui commence par PL et d'extension .plt) avec la fonction **partie_sauvegarde**. Dont le principe est simple, l'utilisateur entre en argument la partie et le nom du fichier dans lequel il veut faire la sauvegarde. On ouvre le fichier en mode « w+» et on teste si le fichier n'existe pas on affiche un message d'erreur. Sinon on commence par écrire PL en tête du fichier et on parcourt le tablier en écrivant, en même temps, dans le fichier le caractère (le type) de la pièce qui se trouve dans chaque case tout en traitant le retour à la ligne et l'entier 0 ou 1 représentant le joueur qui a la main pour jouer.
En effet, l'appel à cette fonction se fait lorsqu'un joueur décide de quitter le jeu et choisit de sauvegarder son plateau de jeu dans la fonction parte_jouer.

Problèmes rencontrés : à la fin de la partie quand le joueur aura choisi s'il veut quitter la partie, on lui donnera la possibilité de sauvegarder sa partie est d'enregistrer, dans un fichier, le plateau du dernier coup joué.
En plus on a trouvé que c'était possible de donner le nom et le chemin du fichier comme paramètres pour la fonction partie_sauvegarder en les concaténant pour gagner en espace mémoire c.à.d. partie_sauvegarder(partie,ctrcat(chemin,nom)). Mais, on a trouvé par la suite d'autres problèmes concernant la taille de la chaine du nom du fichier.

4. Charger les parties à partir d'un fichier

La fonction **partie_charger** prend en argument le nom du fichier qui contient la partie et renvoie un pointeur vers une partie « partie_t *partie_charger(const char *nom) ». Son rôle est de charger le plateau contenant dans le fichier « nom ».
Afin d'accéder à ce fichier, on alloue de l'espace mémoire pour un tableau de caractères « chemin » dont lequel on stock le nom du dossier « Plateaux » suivit du nom du fichier et de l'extension « .plt » : Plateaux/nom.plt
On vérifie si le fichier existe, et on récupère la première ligne dans une variable chaine (pour laquelle on aurait déjà alloué de l'espace), afin de tester si le fichier commence par « PL ». Si c'est le cas, on crée une partie grâce à la fonction partie_creer et on parcourt le fichier pour remplir le tablier de la partie crée. Le remplissage se fait comme suit : on parcourt le fichier en récupérant les mots un par un dans le tableau « chaine ». A chaque fois qu'on lit le mot, on parcourt ses lettres une par une en remplissant les cases du tablier.
Si le fichier n'existe pas, la fonction renvoie la partie NULL (voir la définition de la fonction dans le fichier partie.c qui est bien détaillée).
Les problèmes rencontrés :

- lors de la création de la partie.
  1- Créer la partie et la détruire si le fichier n'existe pas ou s'il ne contient pas PL.

2- Créer la partie après avoir testé la condition sur le fichier en initialisant la partie à null, comme ça si le fichier n'est pas bon en retourne null.

On a choisi la deuxième solution.

- lors du remplissage du tablier

Problème avec la promotion, on devait traiter les cas pour chaque type de pièce. Pour y remédier, on a utilisé la fonction piece_identifier qui crée une pièce en fonction du caractère qu'on a extrait du fichier et qu'on a ensuite affecté a son emplacement dans le tablier.

5. Rejouer une partie

Pour voir le replay d'une partie (la visualiser) on charge d'abord cette partie en appelant la fonction **replay_charger** qui prend en argument juste le nom du fichier contenant la partie et retourne un pointeur vers une partie. A l'intérieur de cette fonction, on alloue de l'espace pour stocker le chemin d'accès à ce fichier (comme on l'a fait dans partie_charger). On ouvre le fichier en mode lecture seule et s'il n'existe pas on affiche un message d'erreur. Si, au contraire il existe, on appelle la fonction partie nouvelle. Et on lit les caractères du fichier ligne par ligne pour créer un coup qu'on ajoute à la fin de la liste des coups. Après chaque lecture, on convertit le premier caractère de la ligne en entier qui représente la coordonnée de depart.x, le troisième caractère qui représente la coord. de départ.y, le sixième représente l'arrivée.x, le neuvième représente l'érrivée.y et enfin le douzième et le quatorzième représentent, respectivement, la promotion et la capture.
Comme vous avez pu le constater, en écrivant dans le ficher, on a laissé entre chaque deux caractère deux espaces (car les nombre se composé d'un ou de deux chiffres) mis à part la promotion et la capture.

Maintenant, pour visualiser la partie, on appelle la fonction **replay_jouer** après avoir chargé la partie grâce à replay_charger.

Cette fonction, propose un menu à l'utilisateur, où il peut choisir de voir le coup suivant (s'il existe) revenir au coup précédent (s'il existe aussi) ou quitter le replay. On a pour cela, crée deux fonctions supplémentaires : replay_suivant et replay_precedent (pour lesquelles on a expliqué le principe dans le fichier partie.c).

On a ajouté la fonction **test_coordonnees** qui prend en paramètres un pointeur sur les coordonnées de départ et un pointeur sur les coordonnées d'arriver. Son rôle est de demander les coordonnées à l'utilisateur, de vérifier si elles sont valides (départ et arrivée) et de redemander tant que ces dernières ne sont pas valides.
Sachant que cette fonction vérifie si les coordonnées de départ et d'arrivée sont dans le tablier (ou dans la réserve artificielle). Si elles sont hors du tablier elle renvoie un message d'erreur et les redemande à nouveau.

III. Exécution du programme

Exécution par défaut : shogi

Le programme lance une nouvelle partie car le nombre d'arguments égal à 1. On affiche un message à l'écran pour dire à l'utilisateur qu'il a choisi de jouer une nouvelle partie. On affecte à P (qui est un pointeur vers une partie) partie_nouvelle « «p=partie_nouvelle(); » et on appelle la fonction partie_jouer pour commencer le jeu.

Exécution à l'aide d'un fichier : shogi NomFichier

Le nombre d'arguments égal à deux. On loue de l'espace mémoire pour une chaine de caractère nom_argv qui contiendra le nom du fichier dans lequel l'utilisateur veut sauvegarder ou charger une partie. Après, on teste :

- Si le fichier existe dans le répertoire Plateaux et dans le répertoire Parties : On demande à l'utilisateur s'il veut sauvegarder ou bien charger une partie.
- Sinon si le fichier existe dans le répertoire Plateaux : on charge le plateau conservé dans le fichier «nom_argv » et on appelle partie_jouer qui permettra au joueur de continuer à jouer sur le plateau chargé (continuer la partie).
- Sinon si le fichier existe dans le répertoire Partie : on charge la partie et les déplacements ainsi que les booléens de capture et de promotion et on appelle la fonction replay_jouer qui permettra au joueur de visualiser la partie chargée.
- Sinon, le fichier n'existe pas, on affiche un message d'erreur pour expliquer pourquoi le programme s'arrête et on retourne un exit_success pour arrêter l'exécution du programme.

Le nombre d'arguments est supérieur à 2 (plus d'un paramètre pour le main) : on affiche un message d'erreur afin d'expliquer pourquoi le programme s'arrête et on retourne un exit_success pour arrêter l'exécution du programme.

ERREURS rencontrés lors de la compilation :

Erreur : abort trap 6

- Lors de la destruction de la partie qui avait une boucle dans laquelle on détruisait la liste des coups si la taille de cette liste est différente de 0. Dans cette boucle, tant que la taille de la liste est supérieure à 0 on extrait le début et on le détruit tout en décrémentant la taille. Or en sortant de la boucle on détruisait aussi la liste_coups_joués (free(&(partie->coups_joués)) qui n'était pas alloué dynamiquement et c'est de la que l'erreur est venue.

Solution :

- On a changé extraire_debut_liste de telle sorte que la décrémentation de la taille se fasse a l'intérieure de cette fonction et non pas dans partie_detruite. Et on a enlevé (free(&(partie-

>coups_joués)) qui ne sert finalement à rien vu que coup_joué est un champs de partie pour lequel on n'a alloué aucun espace mémoire

Erreurs trouvés lors d'une autre compilation :
erreur : Segmentation fault 11

- Dans deplacement on a oublié de tester si on fait le déplacement pour la première fois (coup_joues.nb_coups == 0) alors on colle le coup au début de la liste sinon on le colle à la fin.

Erreurs trouvés lors d'une autre compilation :
erreur : Segmentation fault 11

- Dans annuler_deplacement on a oublié de tester si on a fait plusieurs déplacements avant d'annuler. Alors, quand on annule le déplacement on enlève le dernier coup de la liste des coups joués et on met le suivant de l'avant dernier coup à NULL. Mais si c'est le premier déplacement qu'on veut annuler alors on n'a aucun avant dernier coup à remettre à NULL.
- Et dans la même fonction, quand on remet une pièce à sa case d'origine on a oublié de changer son type si elle est majuscule elle devient minuscule et vice versa.

IV. Partie avancée du projet

Déplacements avancés et fin de Partie

Echec : la fonction qui détermine le gagnant ! Elle prend en argument un pointeur vers une partie et renvoie un entier VRAI ou FAUX. Son fonctionnement consiste à vérifier si la position du roi du joueur en cours le met en échec. Pour cela, on parcourt les cases à droite, à gauche, avant et après la case contenant le roi ainsi que les diagonales et les cases pouvant contenir un cavalier adverse qui pourrait capturer le roi. Dès qu'on trouve une pièce dans l'une des cases parcourues on vérifie si elle est en mesure de capturer le roi dans ses coordonnées actuelles.

Conclusion

Enfin, après avoir terminé la programmation du jeu Shôgi en C, on espère avoir traité tous les cas possibles et avoir bien expliqué nos idées et notre démarche. On a beaucoup travaillé sur ce jeu. Il est intéressant et nécessite suffisamment de concentration tout comme le jeu d'échecs occidental. C'était une bonne découverte à nous trois. Certainement, on a appris pleins d'astuces et pleines de nouvelles fonctionnalités en ce qui concerne le langage C.

Ce projet nous a aidés à progresser et à acquérir plus de compétences dans la programmation impérative. Ça nous a poussés à aimer plus ce domaine et aimer le travail en équipe.