

# Rapport du projet de programmation Impérative: Stackchess

Loïc DUBARD

17 décembre 2018

## Table des matières

<b>1</b>	<b>Démarche générale</b>	<b>2</b>
1.1	Les principales structures de données utilisées ainsi que les différentes opérations implémentées sur ces structures	2
1.2	La gestion de l’affichage . . . . .	2
1.3	La fonction qui fait le lien entre l’utilisateur et le jeu . . . . .	3
1.4	Le calcul du déplacement d’une pièce ou d’une pile . . . . .	3
1.5	Le test de fin de partie . . . . .	5
<b>2</b>	<b>Fonctionnement général du jeu</b>	<b>5</b>

# Introduction

## 1 Démarche générale

### 1.1 Les principales structures de données utilisées ainsi que les différentes opérations implémentées sur ces structures

#### Pile

La pile est la structure la plus importante du jeu, puisque'elle constitue les cases du plateau. Pour l'implémenter, j'ai du choisir entre une structure de tableau de taille fixe et une structure de liste chaînée. Par soucis d'optimisation mémoire j'ai donc choisi de la représenter sous la forme d'une liste chaînée dont chaque maillons contient une pièce du jeu sous la forme d'une chaîne de 2 caractères (ex : "PN" pour pion noir) :

```
typedef struct _noeud * pile;
struct _noeud{char *chaine;pile next;};
```

Les différentes opérations implémentées sur la pile sont :

- savoir si une pile est vide
- retourner le sommet d'une pile
- empiler un nouvelle élément (= pièce) au dessus d'une pile (=case)
- dépiler le sommet d'une pile
- déterminer la longueur (nombre d'éléments) d'une pile
- déterminer le nombre de pièce d'une même couleur donnée au dessus d'une pile
- déterminer le nombre de cavaliers d'une même couleur donnée situés dans la pile
- vider (ou désallouer) une pile

#### Plateau

J'ai ensuite choisi de représenter le plateau par un tableau de pile en deux dimensions que j'ai alloué dans une fonction d'initialisation.

```
pile **initialisation(int N)
{
    int i,j;
    pile ** tableau;
    tableau = (pile**) malloc(N*sizeof(pile*));
    for (i=0 ; i<N ; i++){
        tableau[i] =(pile*)malloc(N*sizeof(pile));
        for (j=0 ; j<N ; j++)
            tableau[i][j] = NULL;
    }
    return tableau;
}
```

Les différentes opérations implémentées sur le plateau sont :

- poser les pièces sur la bonne case du plateau en début de partie
- nettoyer (ou désallouer) le plateau pour éviter les fuites de mémoire
- parcourir le plateau pour déterminer si la partie est finie ou pas
- déterminer si un déplacement est légal
- déplacer les pièces du plateau si le déplacement est légal

### 1.2 La gestion de l'affichage

L'affichage se fait en mode terminal, en utilisant les séquences d'échappement ANSI pour avoir de la couleur :

```
#define clrscr() printf("\033[H\033[2J") //rafraichi l'écran
#define NOIR    "\x1B[48;2;0;0;0m" //colore le fond d'une case en noir
#define PBLANCHE "\x1B[1;32m" //colore les caractères en gris
#define PNOIRE "\x1B[1;31m" //colore les caractères en rouge
#define BLANC    "\x1B[48;2;255;255;255m" //colore le fond d'une case en blanc
#define RESET "\x1B[0m" //remet l'affichage par défaut du terminal
```

On parcourt le plateau et par des test sur le sommet de chaque pile, on affiche pour chaque case :

```

//si on a une pièce noire sur une case noire :
printf(PNOIRE NOIR "%c " RESET, *sommet(tableau[i][j]));
//si on a une pièce blanche sur une case noire :
printf(PBLANCHE NOIR"%c " RESET, *sommet(tableau[i][j]));
//si on a une pièce noire sur une case blanche :
printf(PNOIRE BLANC"%c " RESET, *sommet(tableau[i][j]));
//si on a une pièce blanche sur une case blanche :
printf(PBLANCHE BLANC"%c " RESET, *sommet(tableau[i][j]));
//si on a une case blanche vide :
printf(BLANC " " RESET);
//si on a une case noire vide :
printf(NOIR " " RESET);

```



L'affichage de la pile sur le côté se fait en testant à chaque ligne du tableau si l'on doit rajouter en fin de ligne les caractères (avec le bon espacement) correspondant à la pile sélectionnée. La dernière ligne affichera toujours quelque chose en plus du tableau (soit : "Aucune selection", soit les coordonnées de la case sélectionnée ainsi que le dernier element de cette case si elle n'est pas vide)

### 1.3 La fonction qui fait le lien entre l'utilisateur et le jeu

En utilisant un switch sur un scanf, on execute selon le cas :

- si le caractère entré est 'c' : un scanf sur la case à selectionner
- si le caractère entré est 'd' : on fait un appel à la fonction qui se charge du déplacement des pièces de la case préalablement selectionnée
- si le caractère entré est 'a' : demande à l'utilisateur s'il souhaite abandonner et quitte le cas échéant en affichant le gagnant.
- si les caractères entrés ne correspondent à aucun des cas précédents, l'écran est rafraichi et on redemande à l'utilisateur de rentrer un caractère.

### 1.4 Le calcul du déplacement d'une pièce ou d'une pile

La partie la plus complexe du programme est celle qui se charge de determiner si un déplacement est valable. Pour cela, l'utilisateur ayant déjà selectionné la case à déplacer, l'affichage est rafraichi et si la case contient les pièces de la bonne couleur on demande alors le nombre de pièces à déplacer :



Si ce nombre est inférieur ou égale au nombre de pièces de la case, le programme rafraîchi l'écran et demande une case de destination à l'utilisateur. Puis viens la fonction qui :

- regarde d'abord si le nombre de pièces à déplacer est supérieur strictement au nombre de pièces adverses de la case de destination. Si ce n'est pas le cas renvoie un message d'erreur.
- si le nombre de cavaliers dans la pile à déplacer est supérieur ou égale à  $m/2$  (où  $m$  désigne le nombre de pièces totale à déplacer) alors teste si le déplacement est valable pour un cavalier et l'effectue si c'est le cas.
- sinon teste pour chaque pièce à déplacer si ce déplacement est valable et si une de ces pièces ne convient pas, renvoie un message d'erreur. Sinon effectue le déplacement par désempilement/réempilement.

Un exemple de message d'erreur lors de l'entrée d'un nombre de pièce à déplacer invalide :



Un exemple de message d'erreur lors d'un déplacement d'un fou :



Les fonctions de tests de validité d'un déplacement pour chaque pièce regardent si les coordonnées de destination sont bien dans les intervalles correspondants à chaque pièces et vérifient que l'on ne saute pas de pièce lors de ce déplacement.

Dans le cas du pion, la fonction de test associée différencie le cas où la case de destination contient au moins une pièce adverse et le cas où la case de destination ne contient aucune pièce adverse.

## 1.5 Le test de fin de partie

Les différents cas de fin de partie implémentés sont :

### Un des joueurs n'a plus aucune pièce sur le plateau

A chaque tour, on parcourt le plateau et on incrémente la longueur des piles contenant les pièces d'une couleur donnée dans un compteur pour chaque joueur. Ensuite, il suffit de tester si un des compteur est nul.

### Il ne reste qu'une pièce chacun sur le plateau

En effet, pour "manger" des pièces adverses il faut empiler sur la case adverse strictement plus de pièce que celle-ci ne contient, ce qui est impossible dans ce cas. On se trouve alors dans un cas d'égalité.

### Il ne reste à chacun qu'un fou et une tour

Comme ces deux pièces n'ont aucun déplacement en commun, il est impossible de les empiler en même temps sur la tour ou le fou adverse. Pour detecter ce cas on regarde si le nombre de pièces totale sur le plateau est de 4 et en parcourant celui-ci on compte le nombre de fou et de tour de chaque couleur.

## 2 Fonctionnement général du jeu

Le fonctionnement de la boucle de jeu est résumé dans l'algorithme suivant :

