Projet IPI : stackchess

Jeu d'échec à empilement de pièces

Auteur : ANTEUNIS Charles, ENSIIE 2018-2019

Table des matières

Implémentation du plateau et des pièces

Le plateau de jeu

Je vais représenter le plateau du jeu d'échec par un tableau à **une** dimension. Mon plateau est de taille variable n, je créé donc mon tableau avec :

```
tab = (stack *) malloc (n*n*sizeof(stack)) ;
```

J'ai besoin d'une implémentation de piles pour pouvoir empiler les pièces. Mes piles seront donc des tableaux de pièces, tableaux dynamiques pour une meilleure optimisation de la mémoire :

```
typedef struct stack {
     piece     *tab_stack ;
     int       summit ;
     int       size ;
} stack ;
```

Je vais allouer la mémoire pour le tableau `tab_stack` avec une fonction de prototype :

```
pile create(int n) ;
```

On peut ensuite initialiser le plateau avec les pièces à leurs positions de départ avec une fonction `init(tab, n) ;` et l'afficher avec une fonction `print_game(tab, n) ;`

Les pièces

On représente les pièces par leur type (P pour pion, F pour fou etc.) et leur couleur (N pour noir, B pour blanc). Je définis donc mon type `piece` :

```
typedef struct piece {
     char      nom[3] ;
} piece ;
```

Le nom est de taille 3 : deux cases pour le type et la couleur, une case pour le `'\0'`.

Récupération des caractères de choix

Les coordonnées des cases

Le jeu étant à taille variable, une coordonnée est constituée d'une lettre et d'un nombre à 1 ou 2 chiffres. Je choisi d'imposer une taille maximale de 26 en ayant la lettre comme contrainte. J'implémente deux fonctions permettant respectivement de vérifier la validité d'une coordonnée, et de convertir une coordonnée utilisateur en un indice valable dans mon tableau (par une correspondance entre matrice et tableau 1D) :

```
int is_valid_indice(char str[4], int n) ;
int get_indice(char str[4], int n) ;
```

J'utilise `str` de taille 4 car une coordonnée peut au maximum être de taille 3, plus le `'\0'`.

Pour traiter une coordonnée utilisateur, je traite d'abord la lettre :

```
col = str[0] - 'a' ;
```

puis je créé un pointeur `tmp` ne contenant plus que le nombre et j'utilise :

```
line = n - strtol(tmp, NULL, 10) ;
```

Ainsi je récupère l'indice correspondant dans mon tableau, qui sera `n*line + col`.
Mon problème ici était la récupération du nombre lorsqu'il avait 2 chiffres : je récupérais le nombre avec `str[1]` et `str[2]` mais il fallait faire une disjonction de cas si n>=10 et vérifier que les deux caractères était bien des nombres. L'implémentation est plus simple et efficace avec `strtol`.

Les choix utilisateurs

Le joueur choisit entre sélectionner une case (`c`), déplacer une pile (`d`), ou abandonner (`a`). Il doit aussi entrer un nombre de pièce à déplacer dans le cas échéant. Je récupère le choix et le stocke dans une variable `c` avec :

```
fflush(stdin) ;
scanf(%1s, buf) ;
c = buf[0] ;
```

Le `fflush(stdin)` permet d'enlever tout caractère du `stdin` qui serait resté après un `scanf` . Dans le cas du nombre de pièce, j'utilise :

```
fflush(stdin) ;
scanf(%2s, buf) ;
nbr_deplacement = strtol(buf, NULL, 10) ;
```

J'utilise le `%2s` car on ne peut déplacer théoriquement que 52 pièces d'un coup.

Boucles principales de jeu

Le jeu s'arrête lorsqu'un joueur n'a plus de pièces ou qu'un joueur abandonne. A l'aide de 3 variables, j'implémente donc la boucle principale du jeu :

```
while (abandon && nbr_black && nbr_white) {
     print_game() ;
     if (player_color) printf(« Les blancs jouent\n ») ;
     else printf(« Les noirs jouent\n ») ;
     /* deroulement du jeu */
}
```

A l'aide d'une variable `player_color`, je gère le passage du joueur des blancs au jouer des noirs. Il faut ensuite gérer la boucle de sélection : tant qu'un joueur choisit `c`, il faut afficher le plateau de jeu et le contenu de la case sélectionnée. Pour cela, j'ai utilisé un `while` avec les instructions `continue` et `break`. Il faut aussi une fonction `count_pcs` qui mette à jour les variables `nbr_black` et `nbr_white` lorsqu'il y a un déplacement :

```
while(1) {
     /* instructions de selection/choix */
     if (choix=='c') {
          print_game() ;
          continue; }
     else if (choix=='a') {
          abandon = 0 ;
          break ; }
     else if (choix=='d') {
          /* instructions pour le deplacement */
          player_color = 1 - player_color ;
          count_pcs(tab, n, &nbr_white, &nbr_black) ;
          break ;
}
```

Les fonctions de déplacement

Les validations d'un déplacement

Pour valider un déplacement, il faut vérifier que le déplacement soit légal pour toutes les pièces, mais aussi que les pièces ne passent pas au-dessus d'autres pièces, d'où les fonctions respectives :

```
int is_valid_deplacemt(stack *tab, int n, int nbr, int old, int new)
int is_jumping(stack *tab, int n, int old, int new)
```

Pour la fonction `is_valid_deplacemt`, je vérifie que toutes les pièces de la première jusqu'à la $\texttt{nbr}^{\text{ème}}$ puisse se déplacer à l'aide d'un `switch` :

```
switch(c) {
     case 'T' : if (/* deplacement autorise de la tour */)
                     break ;
                     else return 0 ;
                break ;
     case 'F' : if (/* deplacement autorise du fou */)
                     break ;
                     else return 0 ;
                break ;
     etc.
}
```

Je vérifie les déplacements pour chaque pièce à l'aide des variables `old` et `new` symbolisant la case de départ et d'arrivée. Je converti ces indices en coordonnées dans une matrice : `old` est en case (old_i, old_j) et `new` en case (new_i, new_j) de l'équivalent matriciel du tableau. On a : `old_i=old/n, old_j=old%n, new_i=new/n, new_j=new%n`
Ainsi, il est aisé de vérifier que `old` et `new` sont sur une même ligne, colonne, diagonale etc.

Les déplacements

Lors d'un déplacement, le joueur va soit déplacer sa pile de pièces sans autre effet, soit retirer du plateau les pièces ennemies correspondantes. J'implémente donc les fonctions :

```
void deplacer(int n, stack *tab, int old, int new) ;
void manger(int n, stack *tab, int old, int new) ;
```

Pour déplacer une pile de pièce en conservant l'ordre, j'utilise une pile temporaire `tmp`. Je dépile n éléments de la case `old` en empilant sur `tmp`, puis je dépile `tmp` en empilant sur la case `new`. Pour « manger » les pièces adverses, il suffit de dépiler le contenu de la case `new` puis de déplacer la pile.

Limites du jeu

Taille du plateau

Comme dit dans la section coordonnées des cases, la lettre est une composante de de la coordonnée, j'ai donc limité la taille du plateau à 26 pour ne pas avoir à définir une nouvelle façon de gérer les coordonnées. Il aurait été possible de numéroter les colonnes `'aa'`, `'ab'` après la `'z'`, mais cela implique de changer aussi l'implémentation des fonctions `is_valid_indice` et `get_indice`. La question aurait encore pu se poser après la colonne `'zz'`, il aurait donc fallu généraliser le processus.

Match Nul par placement des pièces

Il est possible d'arriver à une situation de match nul qui ne dépend pas du type des pièces mais de leur placement. En effet, prenons la situation ou un joueur possède 2 pions l'un sur l'autre, et l'ennemi possède deux pions situés sur la même colonne ou une colonne distante de plus de 1 colonne par rapport au premier joueur. En théorie, les joueurs disposent des types de pièces qui leurs permettent de manger des pièces adverses. Mais en pratique, ni l'un ni l'autre ne pourront manger de pions adverses du fait du placement des pions. Ce cas de « match nul par placement » n'est pas pris en compte, un joueur doit abandonner.

Match nul par type des pièces

blablabla