

# Rapport projet S5 : Mansuba

Chloé Le Maître, César Larragueta

Janvier 2023



## Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Description du projet</b>	<b>3</b>
<b>3</b>	<b>Organisation du travail</b>	<b>4</b>
3.1	Répartition du travail . . . . .	4
3.2	Outils de travail . . . . .	4
<b>4</b>	<b>Implémentation du code</b>	<b>5</b>
4.1	Organisation du code . . . . .	5
4.2	Les structures du projet . . . . .	6
4.2.1	Gestion du monde . . . . .	6
4.2.2	Création de plateaux de jeux . . . . .	7
4.2.3	Gestion des pièces des joueurs . . . . .	10
4.3	Implémentation des mouvements des pièces . . . . .	12
4.4	Déroulement d'une partie . . . . .	12
4.4.1	Compilation et choix de l'utilisateur . . . . .	12
<b>5</b>	<b>Tests de validation</b>	<b>13</b>
<b>6</b>	<b>Conclusion</b>	<b>13</b>

# 1 Introduction

Un Mansuba est un problème de Shatranj, un ancêtre perse du jeu d'échec. Le principe est de pouvoir déterminer une séquence de mouvements des pièces sur le plateau permettant d'atteindre la victoire d'un joueur.

Le projet *Mansuba* réalisé au cours du semestre 5, avait pour objectif de développer un programme en C permettant de simuler des parties de jeux de plateau variés. Le code réalisé devait ainsi comprendre un ensemble de structures de données et d'algorithmes facilement modulables afin de pouvoir l'adapter à n'importe quel type de jeu de plateau.

La version initiale du projet se base sur les règles du jeu de dames chinoises. Au fur et à mesure de l'avancée du projet, de nouvelles spécifications étaient demandées. Le challenge était donc d'essayer d'avoir un code clair, cohérent et bien structuré, de manière à ce qu'il soit facilement adaptable à ces nouvelles spécifications.

## 2 Description du projet

L'objectif de notre travail était tout d'abord d'implémenter les fonctions et les structures nécessaires pour la création d'un monde *world*, ensemble de  $WORLD\_SIZE$  points représentant l'ensemble des positions sur lesquelles les joueurs pourront jouer. Ce monde est caractérisé par deux constantes  $HEIGHT$  et  $WIDTH$  telles que  $WORLD\_SIZE = HEIGHT * WIDTH$ . Les positions sont numérotées de 0 à  $WORLD\_SIZE - 1$ . Le monde est ainsi représentable par une grille de points en 2D, de dimensions  $HEIGHT$  et  $WIDTH$ .

À chaque position devaient ensuite être associés deux entiers. Le premier correspond à une couleur : blanc et noir si l'un des deux joueur possède une pièce à cet emplacement, et pas de couleur si la place est libre. Le deuxième entier correspond au type de pièce présent sur cette case (0 si aucune pièce est présente, puis chaque entier naturel non nul correspond à un type de pièce différent). Ainsi, une position est dite libre si elle est incolore et sans pièce (les deux entiers valent 0).

Des relations entre les différents points devaient ensuite être définies pour permettre de savoir quelles positions du monde seront accessibles par les pièces à partir d'une place de départ donnée. La modification de ces relations entraîne donc la création de nouveaux plateaux de jeu. C'était l'objectif de l'achievement 2.

Le monde associé aux relations forment ensemble la base nécessaire pour l'implémentation des différentes fonctionnalités du jeu.

Durant la simulation, les joueurs doivent jouer chacun leur tour, faire bouger leurs pièces avec différents mouvements afin de remplir, avant nombre maximal de tours imposé, une condition de victoire définie. Deux conditions de victoires :

- victoire simple : une des pièces doit atteindre une position initiale de l'autre joueur
- victoire complexe : toutes les pièces doivent atteindre les positions initiales des pièces de l'autre joueur

Les déplacements des pièces dépendent de leur type et du type de plateau (relations entre les positions). Un déplacement peut aussi consister en une série de sauts au-dessus de pièces adverses. La version de base du jeu consistait à implémenter toutes ces fonctionnalités de base permettant de jouer une partie simple. Les achievements permettaient d'ajouter de nouvelles fonctionnalités : ajout de nouveaux types de pièce, création de nouveaux types de plateaux, implémentation d'une fonctionnalité permettant l'emprisonnement de pièces.

## 3 Organisation du travail

### 3.1 Répartition du travail

Ce projet nous a beaucoup appris sur l'organisation à avoir afin de travailler à plusieurs sur un même projet en informatique.

Lors des heures dédiées au travail de ce projet, nous avons pris l'habitude de faire dans un premier temps un point sur l'avancée du travail et de définir ensemble les objectifs de la séance actuelle. L'important lorsque l'on travaille à plusieurs est de bien communiquer. Il nous était nécessaire de discuter ensemble de la manière dont nous allions procéder pour coder certaines fonctionnalités du jeu. Nous avons beaucoup travaillé à deux sur le même ordinateur pendant ces séances, notamment pour les fonctions et les structures qui nous ont paru les plus complexes (en *peer-programming*). Nous nous répartissions ensuite du code à écrire pour la séance suivante, comme notamment les tests de nos fonctions.

### 3.2 Outils de travail

Afin de travailler à deux sur les mêmes fichiers, nous avons utilisé le logiciel de gestion de versions *git*. Cet outil s'est révélé très pratique pour se partager les fichiers entre nous ainsi que pour gérer les problèmes de versions du code. Il a été important d'être clair dans les commentaires descriptifs de nos commits, et de s'assurer que le code déposé compile toujours pour ne pas freiner ou troubler notre binôme dans son travail.

Il nous a paru important de toujours bien commenter les fonctions/structures implémentées afin de permettre une relecture et une compréhension rapide de notre code par le binôme.

Pour éditer le code nous avons utilisé dans un premier temps l'éditeur de texte *emacs* afin de se familiariser avec cet environnement découvert en début d'année en cours d'environnement de travail, avant de passer sur *VisualStudioCode*. Nous avons choisi d'utiliser ce dernier car son interface nous paraissait plus facile d'utilisation, et parce qu'il permet d'avoir des fonctionnalités telles que des indications de débogage claires, ou encore la complétion intelligente du code.

## 4 Implémentation du code

Notre code permet de simuler une partie de jeu entre deux joueurs sur un des trois plateaux que nous avons implémentés, en un nombre de tour et avec une condition de victoire définis.

### 4.1 Organisation du code

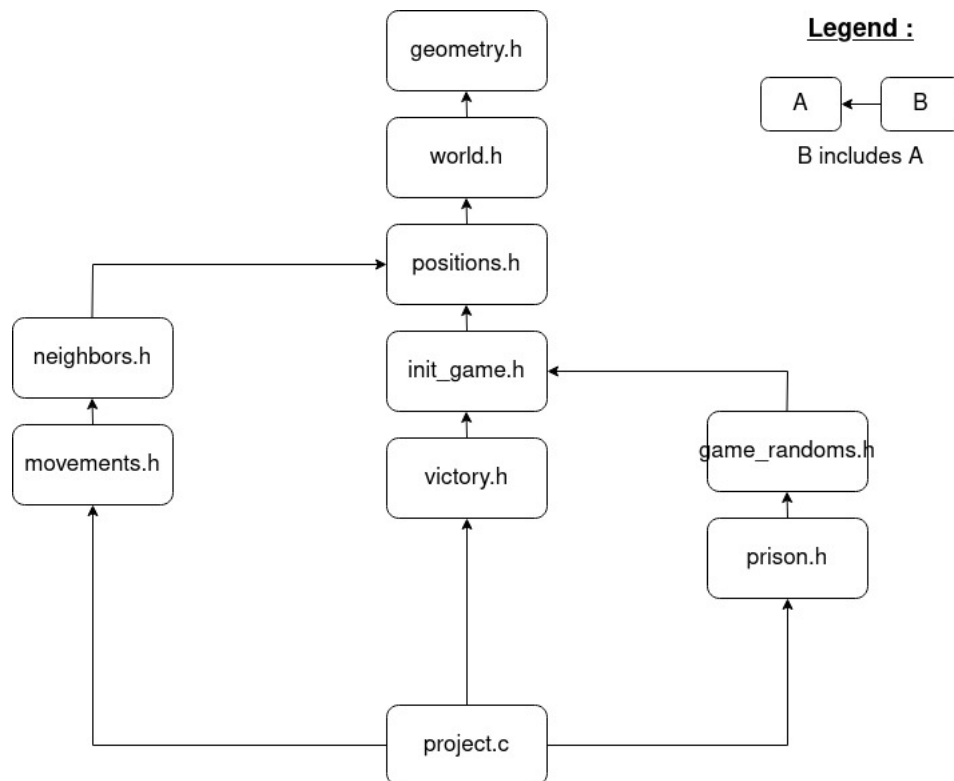


FIGURE 1 – DIAGRAMME DES DÉPENDANCES

Présentation des différents blocs :

- `geometry.h` : Contient les *enums* relatifs aux couleurs et au type des pièces occupant les cases.
- `world.h` : Dans ce fichier sont définis, la structure *struct world\_t* descriptive du plateau de jeu, ainsi que les fonctions relatives à l'accès aux champs de la structure et à leur modifications.
- `positions.h` : Contient la définition de la structure *set*, représentant l'ensemble des positions des pièces de chaque joueur ainsi qu'une fonction d'affichage du plateau de jeu.
- `neighbors.h` : C'est le fichier qui contient les structures de *vector\_t* et *neighbors\_t*, ainsi que le prototype des fonctions qui, à partir de ces structures, calculent les cases "voisines" d'une case donnée.
- `movements.h` : Contient toutes les fonctions qui calculent la destination d'un mouvement pour chaque type de pièces.
- `init_game.h` : Initialise toutes les structures dans leur état de début de partie.

- victory.h : Est relatif aux fonction qui définissent les différents types de victoire et testent à chaque tour si l'un des deux joueurs à gagné.
- game\_randoms.h : Contient toutes les fonctions permettant d'effectuer les tirages aléatoires.
- prison.h : Contient la structure *prison* et ses fonctions de modification, relatives à l'achèvement 3.
- project.c : Ce fichier contient la boucle de jeu, et engendre l'exécutable final du projet.

Chaque fichier .h est associé à un fichier .c de même nom, dans lequel sont codées les différentes structures et fonctions. Ces fichiers .c n'incluent que leur .h associé.

## 4.2 Les structures du projet

### 4.2.1 Gestion du monde

Le monde est représenté grâce à une structure définie de manière abstraite dans le fichier *world.c* : *struct world\_t*.

```
struct world_t
{
    enum color_t color[WORLD_SIZE];
    enum sort_t sort[WORLD_SIZE];
};
```

FIGURE 2 – STRUCTURE POUR LE MONDE *set*

Comme nous l'avons expliqué dans la présentation du projet, le monde se caractérise par une couleur et un type. Pour définir ces deux caractéristiques, nous avons utilisé des énumérations définies dans *geometry.h*.

```
/** Enum defining the possible colors in the world */
enum color_t {
    NO_COLOR = 0, // Default color, used to initialize worlds
    BLACK = 1,
    WHITE = 2,
    MAX_COLOR = 3, // Total number of different colors
};

/** Enum defining the possible sorts of pieces in the world */
enum sort_t {
    NO_SORT = 0, // Default sort (i.e nothing)
    PAWN = 1,
    TOWER = 2,
    ELEPHANT = 3,
    MAX_SORT = 4, // Total number of different sorts
};
```

FIGURE 3 – ENUMÉRATIONS RELATIVES AU MONDE

Un unique monde est créé en début de jeu, et est modifié tout au long de la partie. Il est d’abord initialisé avec que des case vides, puis la fonction *world\_depart* (fichier *init\_game.c*) modifie les bonnes cases pour disposer les pièces des joueurs à leur position de départ en fonction du plateau de jeu, donc en fonction de la racine (*seed* en anglais) présentée juste après.

### 4.2.2 Création de plateaux de jeux

La racine (seed) permet dans notre travail de déterminer quel type de plateau de jeu sera utilisé pour les tours à venir. Nous avons choisi de créer 3 plateaux de jeu distincts :

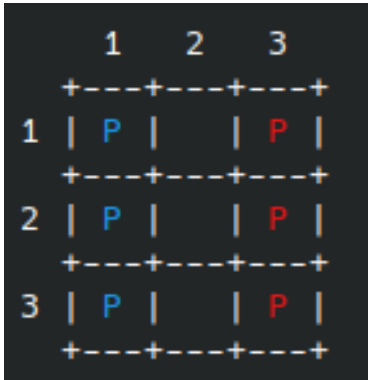


FIGURE 4 – PLATEAU RECTANGULAIRE

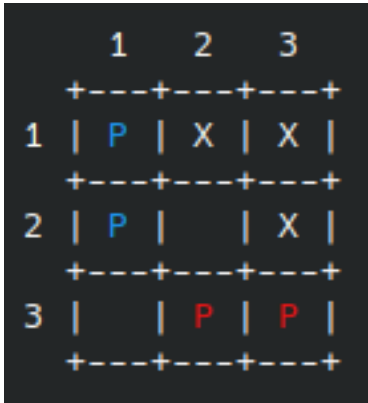


FIGURE 6 – PLATEAU TRIANGULAIRE

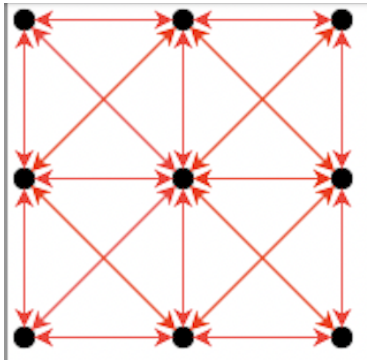


FIGURE 5 – RELATIONS OMNIDIRECTIONNELLES

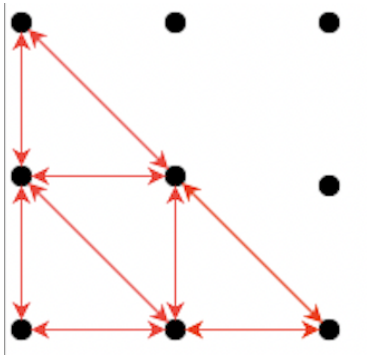


FIGURE 7 – RELATIONS TRIANGULAIRES

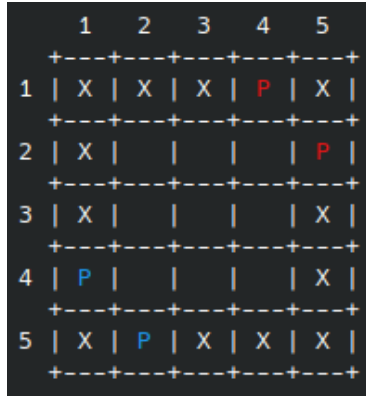


FIGURE 8 – PLATEAU ÉTOILÉ

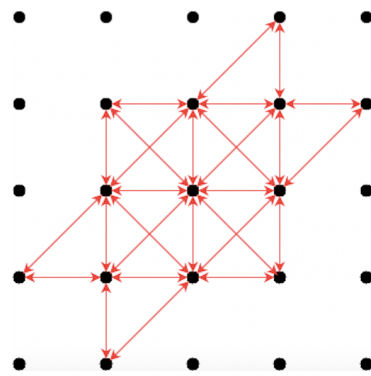


FIGURE 9 – RELATIONS DU PLATEAU ÉTOILÉ

Nous sommes partis du monde (*World*), structure carrée de  $HEIGHT * WIDTH$  points (figure 10), pour définir des relations complexes entre ces points et ainsi former les différents plateaux de jeu.

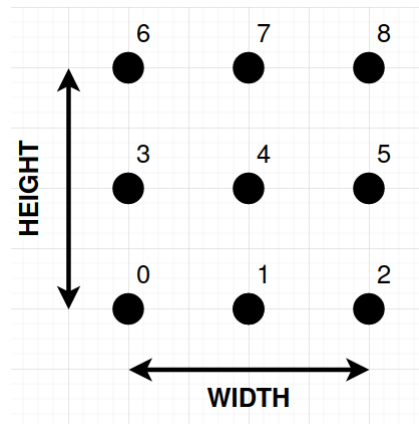


FIGURE 10 – NUMÉROTATION DU MONDE

Nous avons travaillé ces relations dans la fonction *get\_neighbor* (définie dans *neighbors.c*). Son objectif est d'obtenir le voisin d'une case dans une direction donnée. Nous avons implémenté une disjonction de cas sur les différents types de plateaux (selon la racine *seed*).

Pour le plateau rectangulaire, chaque point est en relation avec les 8 points qui l'entourent, à l'exception bien-sûr des bords du plateau (figure 4 et figure 5). Ainsi, dans la fonction, nous effectuons une série de boucles conditionnelles (if, elsif, else) permettant de renvoyer `UINT_MAX` dans les cas où la pièce n'a pas de voisin dans la direction donnée (par exemple une case sur le bord gauche n'a pas de voisin à sa gauche) ou de renvoyer l'identifiant de la case voisine sinon, en fonction toujours de la direction.

Pour le plateau triangulaire (figure 6 et figure 7), nous avons procédé d'une manière un similaire. Nous avons d'abord défini la taille du côté de notre plateau ayant la forme d'un triangle rectangle



isocèle (afin de pouvoir rentrer le plus grand plateau possible dans le monde rectangulaire de taille  $WORLD\_SIZE = HEIGHT * WIDTH$ ).

```
// on détermine la longueur des cotés du plateau triangulaire  
len = WIDTH > HEIGHT ? HEIGHT : WIDTH;
```

FIGURE 11 – PLATEAU TRIANGULAIRE : DÉFINITION DE LA TAILLE

Comme tous les points du monde ne sont pas accessibles dans ce cas de plateau, nous avons dû créer une liste (structure comprenant un tableau et sa taille) des positions possibles des joueurs (si une pièce se retrouve sur une position hors de cette liste alors elle est hors du plateau et ne peut donc pas bouger). Avec cette liste nous avons ensuite procédé de la même manière que pour le plateau rectangulaire pour déterminer les voisins de chaque case en testant en plus si cette case est dans le plateau ou non à chaque fois.

Enfin, pour le plateau étoilé (figure 8 et figure 9), nous avons procédé de la même manière que pour le plateau triangulaire. Nous avons écrit une fonction auxiliaire permettant de créer un pointeur redirigeant vers une liste de positions possibles. La recherche du voisin se fait ensuite de la même façon que pour les plateaux rectangulaire et triangulaire. La difficulté que nous avons rencontrée dans l'implémentation de ce plateau est due à sa forme particulière, étoilée. Ainsi la recherche des relations mathématiques permettant de déterminer quelles positions du monde seront accessibles ou non, nous a pris pas mal de temps.

La fonction *get\_neighbor* présente une complexité cyclomatique assez désastreuse que nous n'avons malheureusement pas su améliorer. En effet, elle possède une structure qui imbrique plusieurs boucles conditionnelles, ce qui rend le code un peu confus et lui confère une mauvaise complexité.

Créer une fonction d'affichage du plateau de jeu dans le terminal nous a pris du temps mais cela s'est révélé très utile pour avoir un visuel du projet. Nous pouvons ainsi suivre la partie jouée tour par tour, vérifier les positions et les déplacements des pièces des différents joueurs.

### 4.2.3 Gestion des pièces des joueurs

Nous avons décidé de pouvoir faire jouer exactement 2 joueurs. Selon le type de plateau choisi aléatoirement au début de la partie, les positions initiales de leurs pièces respectives diffèrent.

	1	2	3	4	5
1	T				T
2	E				E
3	P				P
4	E				E
5	T				T

FIGURE 12 – POSITIONS INITIALES PLATEAU RECTANGULAIRE

	1	2	3	4	5
1	T	X	X	X	X
2	E		X	X	X
3	E			X	X
4	T				X
5		T	E	E	T

FIGURE 13 – POSITIONS INITIALES PLATEAU TRIANGULAIRE

	1	2	3	4	5	6	7	8	9
1	X	X	X	X	X	X	T	X	X
2	X	X	X	X	X	E	E	X	X
3	X	X						E	T
4	X	X						E	X
5	X	X						X	X
6	X	E						X	X
7	T	E						X	X
8	X	X	E	E	X	X	X	X	X
9	X	X	T	X	X	X	X	X	X

FIGURE 14 – POSITIONS INITIALES PLATEAU ÉTOILÉ

La façon dont nous avons choisi de gérer les pièces des joueurs nous a permis de nous familiariser avec de nouvelles structures comme les listes.

En effet, pour repérer les positions des pièces de chacun des joueurs tout au long de la partie, nous avons créé une structure de liste appelée *set* (figure 15). Chaque joueur possède son *set*, liste de taille maximale `MAX_SET_SIZE` contenant les nombres *idx* (entiers non signés) des cases qu'il occupe. Ainsi, à chaque mouvement d'une pièce, la liste associée au bon joueur est mise à jour : le dernier élément de la liste remplace l'élément correspondant à la position initiale de la pièce grâce à la fonction *remove\_pos\_in\_set*, et sa position d'arrivée est ajoutée en fin de liste grâce à la fonction *add\_pos\_in\_set*.

C'est à partir de ces listes de positions que l'on choisit, de manière aléatoire, la pièce qui sera jouée au prochain tour.

```
#define MAX_SET_SIZE 16

struct set
{
    unsigned int positions[MAX_SET_SIZE];
    unsigned int size;
};
```

FIGURE 15 – STRUCTURE DE LISTE *set*

De même, des structures ont été mises en places pour gérer l'emprisonnement de pièces. La structure *prison* est une liste d'éléments de type *prisonnier*. Ce dernier permet de stocker trois informations nécessaires sur le prisonnier en question : le type de la pièce, le joueur à qui il appartient, le numéro de la case sur laquelle il a été capturé et pourra peut-être être relâché.

```
struct prisonnier {
    unsigned int type_piece;
    unsigned int equipe;
    unsigned int idx_capture;
};

struct prison {
    struct prisonnier cellule[2*MAX_SET_SIZE];
    unsigned int size;
};
```

FIGURE 16 – STRUCTURES POUR L'EMPRISONNEMENT

La gestion de la prison durant la partie se fait à l'aide d'un pointeur renvoyant vers un objet de ce type *struct prison*.

### 4.3 Implémentation des mouvements des pièces

Nous avons défini trois types de pièces différents, chacune ayant un comportement différent sur le plateau de jeu (mouvements différents) : le pion, la tour, et l'éléphant.

La plus grande difficulté que nous avons rencontrée à ce niveau du code a été de permettre aux pions (*pawn*) d'effectuer des sauts multiples au-dessus des pièces adverses. Nous n'avons pas su implémenter les fonctions gérant ces mouvements de manière récursive, et avons donc préféré utiliser une boucle *while*. Le code est donc assez compliqué, confus et peu optimisé au niveau des complexités temporelle et spatiales. Les fonctions *test* nous ont permis de vérifier leur bon fonctionnement.

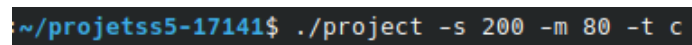
### 4.4 Déroulement d'une partie

Le déroulement d'une partie est implémentée dans la fonction *game* dans le fichier *project.c*. Cette fonction fait appel à toutes les principales fonctions définies dans les fichiers *.h*. Elle fait d'abord appel aux fonctions d'initialisation puis engendre le déroulement du jeu avec une boucle *while* qui ne s'arrête que lorsqu'un joueur a rempli la condition de victoire ou si le nombre de tour maximale a été atteint. La partie se déroule en autonomie, sans l'intervention de l'utilisateur. Chaque choix à effectuer durant le jeu, est effectué aléatoirement (changement de la racine pour changer le type de plateau à intervalles de tours réguliers, choix de la pièce à bouger, choix du mouvement de la pièce, ...).

#### 4.4.1 Compilation et choix de l'utilisateur

Pour effectuer la compilation du projet et des tests, nous avons dû apprendre à utiliser un *Makefile*. Ce document nous permet de compiler facilement grâce aux commandes *make* et *make project* pour le projet en lui-même, ou encore *make test* pour le code contenant les fonctions *test*.

Une fois l'exécutable produit, nous pouvons l'exécuter directement. Cependant il est aussi possible d'ajouter à la ligne de commande quelques paramètres en plus tels que le nombre de tours maximal à effectuer (-m), le type de victoire attendu (-t), ou encore le paramètre concernant l'initialisation du générateur aléatoire (-s). Pour cela, nous avons utilisé la bibliothèque *getopt* qui permet d'analyser cette ligne de commande.



```
~/projetss5-17141$ ./project -s 200 -m 80 -t c
```

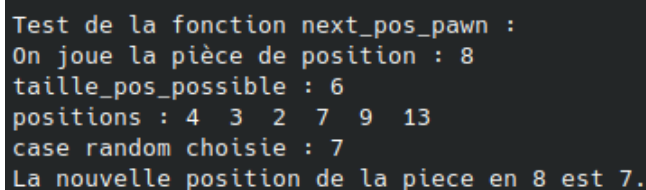
FIGURE 17 – EXEMPLE DE LIGNE DE COMMANDE

La fonction *getopt()* analyse les arguments facultatifs dans la ligne de commande, soit les arguments qui suivent de tirets. Dans notre cas, 't', 's' et 'm' sont des arguments qui seront pris en compte s'ils sont rencontrés. La fonction renvoie un pointeur vers le texte qui suit dans la ligne de commande dans *optarg*. Il est alors possible d'utiliser l'élément pointé par *optarg* pour modifier dans notre cas des constantes utiles au bon déroulement du jeu.

## 5 Tests de validation

Nous avons créé un fichier `test.c` permettant d'implémenter des fonctions testant si nos fonctions fonctionnent correctement. La plupart de ces tests consistent à appeler la fonction à tester avec les initialisations et paramètres nécessaires, et de vérifier qu'elle a le bon comportement en effectuant un affichage dans le terminal.

Par exemple, pour tester la fonction `next_pos_pawn`, fonction qui renvoie une position d'arrivée possible pour un pion à une case de départ donnée, nous avons d'abord initialisé les sets des joueurs avant d'appeler la fonction étudiée. Plusieurs affichages sont faits : la position de la pièce à bouger, l'ensemble des positions accessibles par cette pièce en un mouvement sur le plateau souhaité, et la position renvoyée par `next_pos_pawn`.



```
Test de la fonction next_pos_pawn :  
On joue la pièce de position : 8  
taille_pos_possible : 6  
positions : 4 3 2 7 9 13  
case random choisie : 7  
La nouvelle position de la piece en 8 est 7.
```

FIGURE 18 – AFFICHAGE DU TEST DE `next_pos_pawn`

## 6 Conclusion

Ce projet nous a permis de développer de nouvelles compétences en programmation. Nous avons pu mettre en pratique ce que nous avons appris en cours de programmation impérative au début de l'année, ainsi que nous familiariser avec le langage C. L'utilisation de pointeurs ou la création de structures nous paraissaient assez difficiles au départ mais nous avons vite appris à les manier et avons pu constater leurs avantages.

Nous avons également découvert l'existence et le fonctionnement de git ainsi que du makefile. Ces deux outils se sont révélés capitales pour la réalisation de projets comme celui-ci.

Le travail en binôme nous a permis d'apprendre à bien s'organiser pour que chacun puisse faire sa part du travail et que l'autre puisse comprendre et assimiler le travail de l'autre de manière efficace.

Enfin, le format évolutif du projet nous a appris l'importance d'avoir un code bien écrit, bien structuré, et facilement modulable pour remplir les nouvelles exigences requises au fur et à mesure de notre avancée. C'est pour nous l'un des points les plus difficiles de ce projet. Il sera important pour nous lors de prochains projets, de nous améliorer sur l'écriture de fonctions de manière optimale. La complexité de nos algorithmes est parfois assez mauvaise et pourrait sûrement être moindre si on avait codé différemment.