

# Projet C : FourmiDo

Explications du programme.

Sabri Hani Belli  
Akrim Anass

# Introduction:

- Nous allons expliquer, dans l'ordre, chaque fonction que l'on a utilisé lors de notre projet.
- Nous allons ainsi expliquer leur fonctionnement, pourquoi nous avons eu recours à celles-ci, leur utilité et leur complexité, et quels sont les avantages/inconvénients de chaque fonction (et éventuellement les problèmes rencontrés).

# Structures

- Tout d'abord, nous avons fait appel à 3 structures :
- Structure fourni, qui corres

# Les fonctions Read et Clean:

- Ces trois fonctions sont chargés de la sécurisation des entrées au clavier.
- **int Read(char \*chain, int length)**  
La fonction lire va récupérer au clavier au maximum length caractères avec la fonction *fgets* ou n caractères jusqu'au retour à la ligne.
- **void CleanBuffer()**  
La fonction *CleanBuffer* va vider le buffer à la suite d'un appel à la fonction lire.
- **long ReadLong ();**  
Dans notre programme nous récupérerons des chaines de caractères ainsi que des entiers, pour cela nous appelons la fonction *ReadLong* avec un changement de type (int). Cette fonction appel *Read ()* et converti la chaine de caractère en un nombre du type long.

# La fonction Print\_board

- Cette fonction prend en argument le plateau c'est-à-dire un tableau de dimension 2 et va l'afficher.
- Pour cela on utilise une double boucle for pour parcourir les lignes puis les colonnes en affichant à chaque fois l'élément présent dans la case. A la fin du parcours d'une ligne on retourne à la ligne (on passe à la seconde ligne pour la parcourir).
- Pour plus de clarté, nous avons ajouté une ligne et une colonne qui correspondent aux numéros de lignes/colonnes.
- Complexité de l'algorithme :  $O(\text{SIZE}^2)$  en raison des deux boucles for imbriquées.

# La fonction Fight

- Pour gérer les combats, nous avons créé une fonction Fight qui prend en argument la structure World, et deux pointeurs vers des fourmis.
- Il est important de noter *qu'ant1* est la fourmi qui se déplace et *ant2* la fourmi qui reste immobile. Pour gérer l'issue des batailles, nous utilisons la commande *srand()%(3)+1* qui nous permet de tirer aléatoirement un entier entre 1 et 4.
- Dans cette fonction on traite deux cas :

Les agents qui se battent sont soit 2 soldats soit des non-soldats : On estime donc que chacun a autant de chances que l'autre de gagner (1chance/2) ;

Un seul des deux agents est un soldat : On estime alors que le soldat a 75% de chances de gagner (donc 3chances/4).

- A l'issue du tirage aléatoire et en fonction du cas traité (1 ou 2) nous devons effacer le maillon correspondant à l'agent vaincu de la liste chaînée de sa fourmilière.

Ici aussi deux cas s'offrent à nous :

-L'agent n'est pas une fourmilière, il suffit donc de supprimer son maillon de la liste chaînée.

-L'agent est une fourmilière, la tâche est plus complexe. En effet il faut d'abord détacher cette fourmilière de ses éventuels voisins, puis il fait parcourir toute la liste chaînée correspondant aux agents qui lui sont liées en les supprimant un par un. Ici aussi, nous avons deux cas à traiter :

- L'agent ant2 n'est pas une Ouvrière (ant2->type !=3), on peut donc supprimer le maillon lui correspondant.
- L'agent ant2 est une Ouvrière (ant2->type ==3), il faut alors supprimer cet agent, et en créer un autre de même type, à la même position mais appartenant au camp ennemi. Pour cela on stocke sa position et on fait appel à la fonction *NewWorker* sur une fourmi dont on vient d'allouer l'espace mémoire, on la place sur le plateau à la bonne position et on lie cet agent à la liste chaînée de la fourmi ant1.

Pour la gestion des cas où il s'agit d'une fourmilière nous avons créé une deuxième fonction :

# La fonction FightAnt

- Nous avons décidé de passer par une seconde fonction pour plus de clarté dans le code et pour ne pas surcharger la première fonction Fight. Cela nous a également beaucoup aidé lorsque nous devions debugger, ceci nous a permis de repérer le bug qui nous causait des soucis.
- Sa complexité ne dépend que de la longueur ( $L$ ) de la liste chaînée parcourue pour la suppression des agents, elle est donc au maximum en  $O(L)$ . De plus comme la fonction Fight appelle cette fonction, la complexité de la fonction précédente Fight est également de l'ordre de  $O(L)$  (au pire des cas) car toutes les autres opérations effectuées par Fight sont en  $O(1)$ .



# La fonction Ordre\_ant :

Cette fonction joue un rôle clé dans le programme.

En effet elle est chargée de gérer tous les ordres donnés par l'utilisateur. On a trois ordres majeurs :

- **1)** L'utilisateur veut déplacer la fourmi se trouvant à la position (linedep, columndep) vers la position (linearr, columnarr), donc \*ordre= « bouger ». Avant toute chose, nous vérifions si nous pouvons déplacer l'agent en question, c'est-à-dire s'il ne s'agit pas d'une ouvrière immobilisé (type==3 et immo==1), ou encore si linedep==linearr et columndep==columnarr, c'est a dire si la case de destination correspond à la case actuelle de l'agent.
- Si l'agent peut se déplacer nous pouvons continuer. Pour cela nous avons répertorié tous les cas de déplacement possibles :

F				A	
	X			B	
	D			C	
E					

- Imaginons que nous voulons déplacer la fourmi X sur ce plateau, nous travaillons alors en fonction de ligne X et colonne X,
- 
- a) Déplacement en A, on a  $\text{ligneA} < \text{ligneX}$  et  $\text{colonneA} > \text{colonneX}$ , on va donc incrémenter ligneX et ligneY ( $\text{linedep}++$  et  $\text{columndep}++$ ). On modifie donc  $\text{X} \rightarrow \text{pos}[0]$  et  $\text{X} \rightarrow \text{pos}[1]$ . X se déplace donc d'une case en
- 
- b) Déplacement en B, on a  $\text{ligneB} == \text{ligneX}$  et  $\text{colonneB} > \text{colonneX}$ , on ne va incrémenter que colonneX.
- c) Déplacement en C, on a  $\text{ligneC} > \text{ligneX}$  et  $\text{colonneC} > \text{colonneX}$ , on va donc décrémenter ligneX et incrémenter ligneY ( $\text{linedep}--$  et  $\text{columndep}++$ ). On modifie donc  $\text{X} \rightarrow \text{pos}[0]$  et  $\text{X} \rightarrow \text{pos}[1]$ . X se déplace donc d'une case en
- d) Déplacement en D, on a  $\text{ligneD} > \text{ligneX}$  et  $\text{colonneD} == \text{colonneX}$ , on ne va incrémenter que ligneX.
- e) Déplacement en E, on a  $\text{ligneE} > \text{ligneX}$  et  $\text{colonneE} < \text{colonneX}$ , on va donc incrémenter ligneX et décrémenter colonneX ( $\text{lignedep}++$  et  $\text{colonnededep}--$ ). On modifie donc  $\text{X} \rightarrow \text{pos}[0]$  et  $\text{X} \rightarrow \text{pos}[1]$ . X se déplace donc d'une case en
- f) Déplacement en F, on a  $\text{ligneF} < \text{ligneX}$  et  $\text{colonneF} < \text{colonneX}$ , on va donc incrémenter ligneX et décrémenter ligneX et colonneX ( $\text{lignedep}++$  et  $\text{colonnededep}--$ ). On modifie donc  $\text{X} \rightarrow \text{pos}[0]$  et  $\text{X} \rightarrow \text{pos}[1]$ . X se déplace donc d'une case en.
-

- Il est important ici de noter que la fonction ne fait bouger l'agent X que d'une position, il va donc falloir la rappeler tant que `fourmi->pos[0] != fourmi->dest[0]` ou que `fourmi->pos[1] != fourmi->dest[1]`.
- 2) L'utilisateur veut immobiliser l'agent, ici il y'a deux cas auxquels nous devons prêter attention :

a) L'agent en question n'est pas une ouvrière, on passe le tour de l'agent sans exécuter d'instruction particulière.

b) L'agent en question est une ouvrière, on doit donc l'immobiliser pour le reste du jeu, alors :

`World->plateau[lignedep][colonedep].fourmi->immo=1.`

3) Enfin, si l'utilisateur veut détruire l'agent en question, pour ce faire, nous retirons le maillon correspondant à cet agent de la liste chaînée et le supprimons.

- → La complexité ici est en  $O(1)$ , dans le pire des cas, on veut bouger une fourmi et la fonction effectue 6 comparaisons de cas pour trouver celui qui correspond à notre situation et fait un échange sur le plateau.
- → Cette fonction n'est pas très 'esthétique', mais c'est la première fonction à laquelle nous avons pensé en réalisant ce projet et, au final, le fait de lister les cas nous a beaucoup aidé.

En effet lors de nos premiers essais nous avons des bugs liés aux déplacements (déplacements en haut alors que la fourmi devait aller en bas, etc ..), ainsi lorsque nous observions un déplacement inhabituel, nous savions directement de quel cas il s'agissait et nous étions en mesure de corriger nos erreurs assez rapidement.

# Les fonctions NewAnthill, NewQueen, NewSoldier et NewWorker

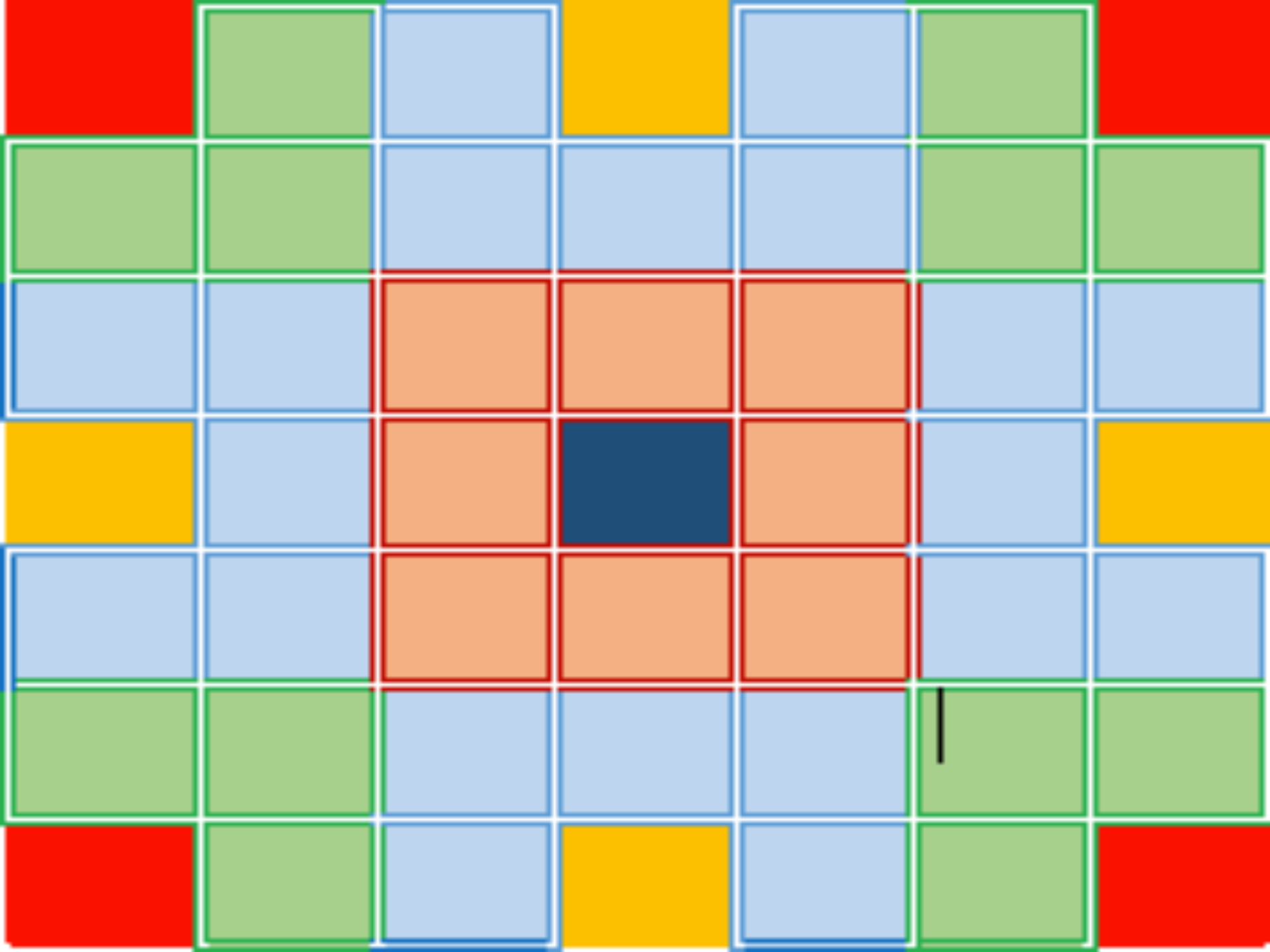
- Ces quatre fonctions sont similaires, elles ont toutes les quatre pour but d'initialiser les champs de la structures Fourmi pour créer des fourmilières, Reines, Ouvrières ou encore des Soldats. Elles diffèrent cependant au niveau d'une instruction.
- Evidemment toutes les fourmis ne sont pas du même type : les fourmilières sont du type (1), les Reines du type (2), les Ouvrières du type (3) et les Soldats du type (4). Les valeurs de champ *ant->type* diffèrent donc selon la fonction appelée.
- Pour le reste les quatre fonctions vont faire la même chose. Le champ *ant->color* va prendre la valeur du paramètre formel *color*, et les champs *ant->pos[0]* et *ant->pos[1]* vont respectivement prendre les valeurs ligne et colonne passées en paramètre. Tous les champs correspondants aux pointeurs *suiv*, *prec*, *next* et *prev* sont quant à eux initialisés à NULL.

- Ici le nombre d'opérations ne dépend pas des paramètres en entrée, la complexité est donc en  $O(1)$ .
- Nous n'avons pas rencontré de difficulté particulière avec ces fonctions, nous avons cependant dû les adapter dès le niveau 2 en ajoutant l'initialisation de deux nouveaux pointeurs par classe de fourmis. Nous n'avons pas remarqué que cela faisait « bugger » notre programme jusqu'à ce que nous le testions sur les machines de l'université.
- Nous avons perdu un petit peu de temps car nous ne savions pas que le bug provenait de là.

# La fonction free

- Cette fonction joue un rôle important lors de la production d'un nouvel agent par une fourmilière.
- En effet après appel, les paramètres  $i$  et  $j$  prennent la valeurs de la ligne et de la colonne d'une case voisine de la fourmilière. Si une case est trouvée elle renvoie 1, sinon elle renvoie 0. Les paramètres ligne et colonne eux correspondent à la position de la fourmilière qui tente de produire un agent.
- Pour cette fonction nous avons traité 8 cas (beaucoup de lignes de code ce qui n'est pas très esthétique, mais ceci est la première idée qui nous soit venue, elle nous assure également que nous traitons tous les cas). Nous n'avons pas créé une fonction qui parcourt plusieurs 'couches' de cases autour de la fourmilière, en effet une fourmilière doit créer ses agents et les placer sur une de ses cases voisines, donc nous ne travaillons que sur les cases qui lui sont mitoyennes.
- Cependant le mode de fonctionnement de la fonction est assez simple, il repose sur l'évaluation de chacun de ces 8 cas. Pour chacun de ces cas, nous allons faire des opérations bien particulières qui nous assureront de trouver une case libre autour d'une fourmilière si bien sûr il y'en a une.





- Considérons ce plateau,

Notre fonctions va d'abord traiter les cas « critiques » c'est dire les quatre coins en rouge.

En effet en fonction du coin considéré ((0,0), (0, SIZE-1), (SIZE-1, SIZE-1) ou (SIZE-1, 0)) donc on fonction des valeurs des paramètres ligne et colonne, nous allons parcourir les 3 cases voisines correspondantes (cases aux bordures vertes).

- Puis nous traitons le cas des bords. Ici aussi en fonction du bord considéré (droit : (X, 0), haut : (0,X), gauche : (X,SIZE-1), ou bas : (SIZE-1, X) avec  $X \in ]\text{SIZE-1}, \text{SIZE-1}[$ ), en fonction des valeurs des paramètres ligne et colonne, comme pour le cas des coins, nous savons donc parfaitement quelles sont les 5 cases à parcourir(cases aux bordures bleues).

- Enfin tous les autres cas se traitent de la même manière, nous n'avons plus a nous soucier de le position de la fourmilière.

En effet il ne nous reste plus qu'a parcourir les 8 cases autour de la fourmilière (exemple des cases aux bordures rouges).

- Dans tous les cas, nous n'utilisons aucune boucle.

- Comme pour la fonction `ordre_Ant`, le fait d'avoir séparé tous les cas ici nous a pris du temps certes, mais nous avons au final pu retrouver les sources de certains bugs que nous avons remarqué, en effet parfois même s'il y'avait une case libre autour de la fourmilière le programme nous disait le contraire, et connaissant l'emplacement de la fourmilière sur le plateau, nous avons pu retrouver la partie de la fonction qui nous posait problème.

# La fonction productant :

- Voici la fonction en charge de la production d'un agent par une fourmilière, ses paramètres formels correspondent à:
  - Un pointeur sur le World,
  - Deux entiers, ligne et colonne qui représentent la position de la fourmilière sur le plateau
  - Un entier, code qui correspond à la classe de la fourmi que l'on veut créer,
  - Un caractère color pour la couleur de l'agent (rouge ou noir),
  - Un pointeur vers la future fourmi que l'on va créer.
- On commence d'abord par vérifier que la fourmi qui se trouve à la position (ligne, colonne) est bien une fourmi (*plat->plateau[ligne][colonne].fourmi->type==1*) et aussi qu'il y'a une case libre autour d'elle (*Libre(plat, ligne, colonne, &i, &j) !=0* avec i et j deux entiers déclarés auparavant).
- Puis en fonction de la valeur de *code* nous créons une nouvelle Reine, Ouvrière ou un nouveau Soldat grâce aux fonctions New (pour une Reine par exemple : *NewQueen(ant, color, i, j)* où ant est une fourmi à laquelle nous avons déclaré l'espace mémoire nécessaire).
- Il nous faut ensuite placer l'agent fraîchement créé sur le plateau, à la position *plat->plateau[i][j]*.

- Enfin, il faut ajouter cet agent à la liste doublement chaînée des agents de la fourmilière, nous avons décidé d'insérer le maillon du nouvel agent au début de la liste doublement chaînée (c'est-à-dire juste après le maillon de la fourmilière) pour plus de facilité.
- Lorsque l'agent est finalement produit et installé nous prévenons le joueur avec un message à l'écran que l'agent a été produit.
- Si nous n'arrivons pas à créer un agent parce qu'il n'y a pas de case libre autour de la fourmilière, nous prévenons le joueur en lui disant que la production est en attente et qu'il doit libérer une case pour l'agent soit produit.
- Ici on retombe sur une complexité en  $O(1)$  car il n'y a pas de boucles, pas de récurrences, le nombre d'opérations dépend certes des entrées mais pas de leur taille.
- Nous avons créer cette fonction assez naturellement, elle ne nous a pas poser trop de difficulté étant donné qu'elle fait appel à des fonctions déjà créées. Le seul points qui nous a donné un peu de mal est la gestion de la liste doublement chaînée, en effet au début nous avons presque toujours des *segmentation fault*. Cependant après un certains temps, nous avons réussi à contourner cette difficulté.

# La fonction Save:

- Cette fonction va nous permettre de sauvegarder la partie. On peut faire appel à cette fonction au début de chaque tour, c'est à dire lorsque le joueur choisit de sauvegarder la partie au début de son tour.
- Elle prend comme argument :
  - Les listes de fourmis rouges et noires ;
  - Le nombre de pièces de chaque joueur(on prend seulement les valeurs, les variables ne seront pas modifiées).
  - La variable round, soit quel tour de la partie ;
  - Dans le main, roundPlayer=0 si le joueur 1 est en train de jouer, 1 si c'est le 2<sup>e</sup> joueur qui joue. Cette variable nous permet de savoir quel joueur a sauvegarde, afin de retomber directement sur le tour de ce joueur lorsque l'on charge la partie.
- Dans cette fonction save, on initialise 4 pointeurs par malloc en leur allouant l'espace d'une structure fourmi :
  - On commence par créer un fichier f, puis on écrit dessus la première ligne le nombre de pièces de chacun, la valeur de round et roundPlayer ;
  - Ensuite, on affecte aux pointeurs p et pp la liste de fourmis rouges.

- Premièrement, p et pp vont pointer sur le premier élément de la liste, soit la fourmilière.

On dispose alors de deux boucles itératives :

Tant que pp différent de NULL (soit la principale) :

Au début p=pp, soit la fourmilière, ensuite la deuxième boucle :

Tant que p différent de NULL :

P pointe sur un élément, non nul. On va alors transcrire sur le fichier les informations de l'élément sur le fichier, selon le type :

+ Si fourmilière (type 1) , alors la couleur, le type, la position, le type et temps restant de la production, ainsi que temps d'immobilisation (celui ci inutile, il servira seulement à ne pas laisser une case vide dans un tableau pour plus tard)

+Sinon, alors la couleur, type, position, case en destination, temps d'immobilisation.

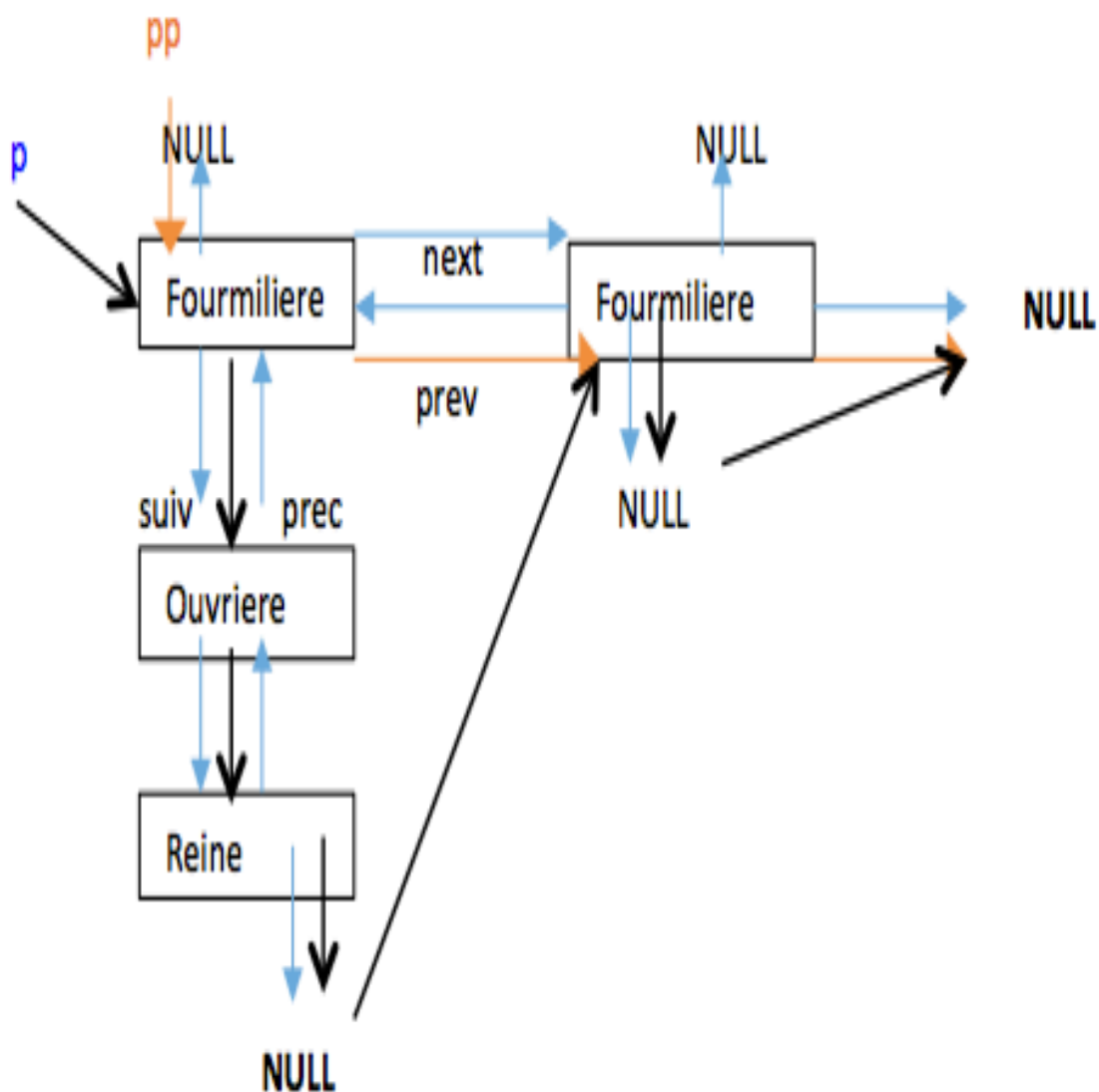
Lorsque p=NULL, soit fin tant que,

Alors on affecte a pp= pp->next, soit la prochaine fourmilière si existe.

Fin tant que

- Si elle existe, alors p=pp, la 2<sup>e</sup> fourmilière, et on réitère ensuite notre procédé (représenté par le schéma suivant).


(p pointe avec les flèches en noir).



pp va parcourir chaque  
liste chainee (fleches en  
orange).

P va parcourir chaque  
element de la liste des  
fourmis rouges.

- Meme procédé pour la liste de fourmis noires, puis on termine en fermant le fichier et libérant l'espace.
- Au tour n°1 et lorsque le joueur 1 joue son tour , on obtient un fichier comme ceci:



	50	50	0	1			
	r	1	0	0	0	0	0
	r	3	0	1	0	1	0
	r	2	1	0	8	0	0
	n	1	9	9	0	0	0
	n	3	9	8	9	8	0
	n	2	8	9	8	9	0
	l						

Couleur

Type

Position  
(ligne et colonne)

Destination si type  
different de 1,  
Type de production et  
temps restant si type de la  
fourmi concernee=1.

Temps  
d'immobilisation.



- Complexité= dépend de la longueur de la liste du monde de fourmis= $O(1)$  car il y a au plus 100 fourmis,
- Nous avons choisi d'établir la fonction save comme ceci afin de parcourir tout la liste et de noter toutes les informations dans un fichier texte ensuite.
- Plus tard, si l'on souhaite charger l'ancienne partie, on aura recours à ce fichier texte afin de récupérer toutes les informations.

# La fonction init\_world :

- Il s'agit ici d'initialiser le plateau en début de partie.

Cette fonction prend donc en argument :

- Un pointeur sur le Monde
- Six pointeurs sur des Fourmis qui correspondent aux 6 agents présents lors du lancement d'une nouvelle partie.

Notre fonction fait appel aux fonctions New pour créer tous les agents nécessaires (Fourmilière Rouge, Ouvrière et une Reine pour chacune des deux couleurs).

Une fois les six agents créés, elle les lie dans deux listes doublement chaînées ayant pour premier élément les deux fourmilières, puis elle fait pointer les deux pointeurs Frouge et Fnoir de la structure Monde vers ces deux fourmilière (On a donc pour les camp noir la liste chaînée suivante : World>Fnoir=FourmilereNoire->OuvriereNoire->ReineNoire).

# La fonction Load:

- Cette fonction est similaire à la précédente (init\_world). Au départ, l'environnement n'est pas initialisé. Si l'on choisit une nouvelle partie, alors on fait appel à init\_world, ce qui nous permet de tout initialiser (l'environnement du jeu).
- Sinon, alors on fait appel à load, celle-ci prend pour arguments :
  - Les mêmes pointeurs utilisés précédemment dans init\_world
  - Deux pointeurs de plus (ici appelés Sr et Sn pour soldats rouges et noirs si existants).
  - Les pointeurs sur gold1 et gold2 pour modifier la valeur du nombre de pièces.
  - Pointeurs pour modifier la valeur de la variable tourJoueur (c'est à dire pour revenir sur le tour du joueur précisé sur le fichier : si dans le fichier, tourJoueur=1 alors on passe directement au joueur 2, sinon on reste sur le 1), et la valeur de la variable tour.

On initialise 4 pointeurs p, pp, q, et qq à null.

On commence par créer le plateau à l'aide de deux boucles for (comme dans init\_world).

Ensuite on fait appel au fichier texte de sauvegarde que l'on va lire :

- On va commencer par lire la première ligne, contenant 4 entiers que l'on va stocker dans le tableau vars, et l'on va affecter les valeurs collectées à, respectivement, gold1, gold2, tourJoueur, tour. Les variables sont alors modifiées.
- -Sachant que, dans tous les cas, on ne peut sauvegarder une partie si un joueur ne dispose plus aucune fourmi. Alors on peut être sûr qu'au pire des cas, chacun dispose une fourmière. (car si un joueur ne possède AUCUNE fourmière, celui-ci a perdu la partie.)
- De ce fait, sachant aussi qu'une fourmière est le premier élément d'une liste et que la première du joueur 1 est rouge, alors on va commencer par lire la seconde ligne et ensuite créer une fourmière (d'argument le pointeur Fr initialisé précédemment dans le main) en faisant appel à la fonction concernée, et avec les valeurs collectées (dans les tableaux coul et characs). On place ensuite la fourmière dans le plateau de jeu.
- On alloue à pp l'espace d'une structure fourmi, lui affecte Fr, puis on affecte pp à p. Ensuite on initialise un entier n à 0 que l'on va utiliser plus tard.

Ici, on lance une boucle while, tant que fscanf différent de null, soit tant qu'il reste une ligne à lire.

- Au début de chaque itération, on va collecter les informations de la ligne prise en compte,
- Si la couleur est rouge, soit un caractère 'r' perçu, alors on fait appel à une boucle switch, de paramètre charac[0], soit le type :
- Si type= 2 ou 3 ou 4, alors on crée un agent comme ce que l'on a fait pour la fourmière précédente, sauf que l'on va allouer à p->suiv l'espace d'une structure fourmi, que l'on va remplir par un la case d'un agent créé, pour ensuite faire avancer le pointeur p vers cette case. P pointe alors sur cet agent créé, et on termine par p->suiv=NULL.

- Sauf que si type= 1, soit une fourmiliere, alors on va recréer une fourmiliere comme ce que l'on a fait pour la premiere. Or, sachant que pp pointe toujours sur cette premiere fourmiliere, alors on va affecter pp->next à p. On lance ensuite une boucle itérative tant que p different null, alors on fait avancer p(next). Lorsque p pointera sur une case vide, alors on va lui affecter cet agent crée en lui ayant alloué au préalable l'espace d'une structure fourmi. P va alors pointer sur cette nouvelle fourmiliere, et les prochains agents que l'on va créer seront relies à cette nouvelle fourmiliere.
- Lorsque l'on aura complété cette liste de fourmis rouges (lorsqu'il n'y a plus de caractere 'r'), alors le premier élément que l'on va trouver est une fourmiliere noire. Pour ce faire, entrer dans la boucle else, puis ensuite dans la boucle if comprise (si n=0) dans celle ci, et on va créer une fourmiliere noire comme procédé effectué pour la liste rouge, sauf que l'on va utiliser q et qq à la place de p et pp. On va ensuite modifier n pour ne plus revenir dans cette boucle if, et on fait appel à continue pour scanner la prochaine ligne et sauter les prochaines commandes.
- Cette procedure impliquant la variable n (initialisée à 0) va nous permettre de garder qq comme un pointeur sur la premiere fourmiliere noir.
- Ensuite, on va effectuer le meme procede effectue pour la liste rouge de fourmis.
- 
- A la fin de la boucle while, on accorde a World->Frouge le pointeur pp, soit ici la liste de fourmis rouges prises en arguments va pointer sur la premiere fourmiliere rouge. De meme pour World->Fnoir et qq.
- On finit par liberer l'espace utilise.
- Complexite identique à la fonction save.

# Fonction main

- C'est dans le main que nous avons décidé de mettre toutes les instructions concernant l'interface de jeux. Ce n'est pas la meilleure methode, en effet au final nous nous retrouvons avec une fonction main assez longue, ce qui rend le code peu esthetique.
- Le main comporte deux parties principales : une pour chaque joueur. Comme les deux parties sont assez similaires, nous n'allons en expliquer qu'une seule.

- Au début de la fonction, nous avons commencé par déclarer toutes les variables dont nous allons avoir besoin lors de nos appels aux fonctions. Avant de faire jouer le premier joueur, nous affichons d'abord un menu avec 3 Options :
  - 1-New Game (On démarrer une nouvelle partie : appel à la fonction `init_World`)
  - 2-Load Previous Game (On charge une ancienne partie : appel à la fonction `Load`)
  - 3-Show the 10 best scores (On fait appel à la fonction `top_players` pour afficher les 10 meilleurs scores stockés dans un fichier `payers.txt`).
- L'utilisateur doit alors entrer un entier compris entre 1 et 3 en fonction de ce qu'il veut faire. Tant qu'il n'a pas entré une valeur correcte (entre 1 et 3) le programme continue d'attendre une entrée.
- Pour le niveau 1 nous n'avons utilisé que deux pointeurs rouges et noirs pour pointer vers les fourmilières pointées par Frouges et Fnoire. Puis à partir du niveau 2 nous avons ajouté 2 nouveaux pointeurs pour pointer vers les autres éventuelles fourmillières.

- Nous avons tout d'abord utiliser une boucle while qui porte sur la valeur de l'entier *choix* s'il vaut 4 nous quittons la boucle et par consequent le jeux. En effet à chaque debut de tour le joueur se voit offrir 4 options :

1-Play.

2-Skip.

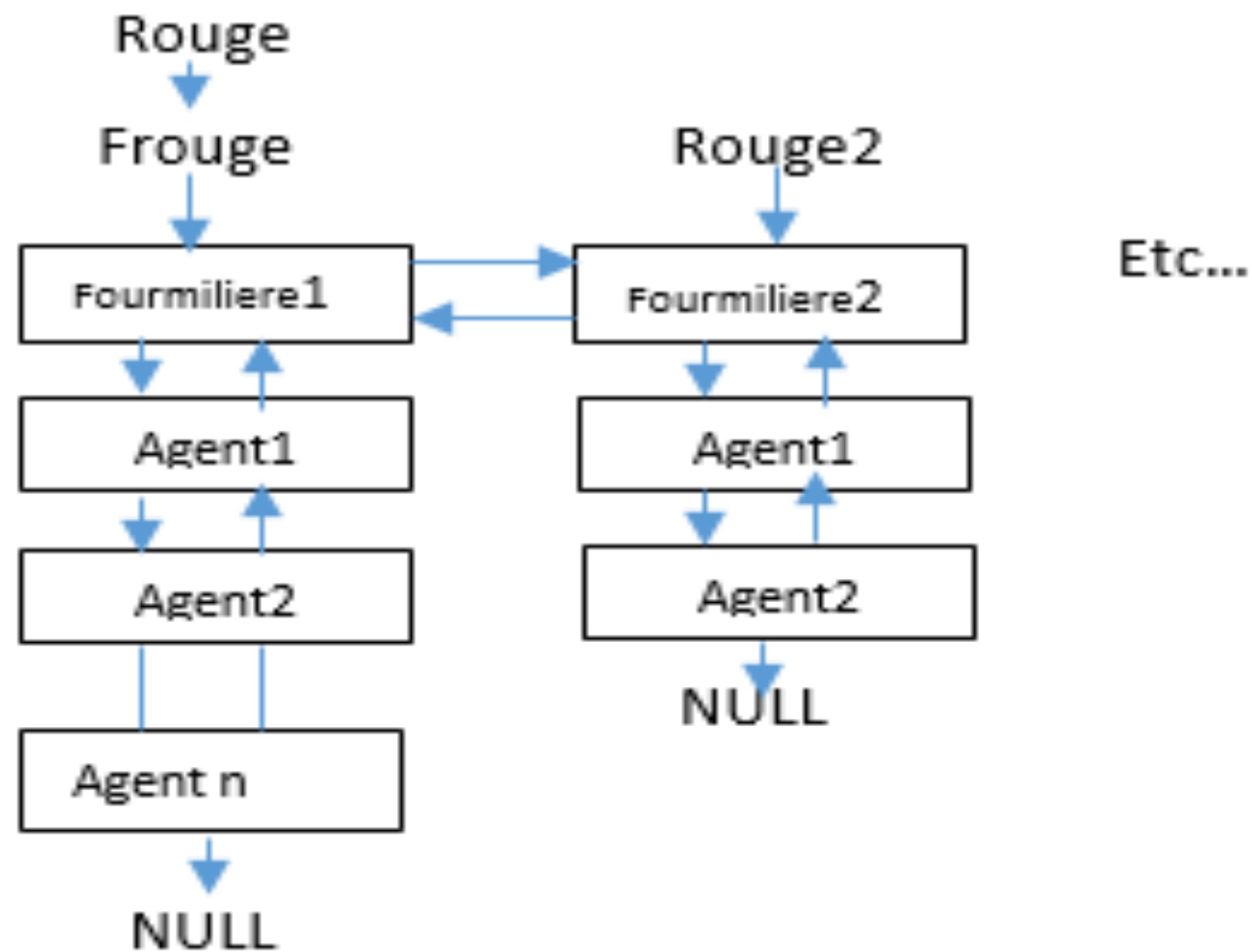
3-Save

4-Quit.

- En fonction de ce qu'il entrera au clavier, le programme se comportera differemment.



- Grace à deux boucles while imbriquées et portant sur la valeur des pointeurs rouge et rouge2, nous parcourons toute les listes chaînées des agents.
- Dans un premier temps, nous parcourons la première liste tant que rouge != NULL (à chaque agent traité on fait pointer rouge sur le suivant).
- Puis quand rouge == NULL, nous faisons rouge = rouge2 pour faire pointer rouge sur la deuxième fourmilière et rouge2 = rouge2->next pour faire pointer rouge2 sur la troisième et ainsi de suite jusqu'à ce que rouge2 == NULL .



- Pour chaque agent pointé par rouge, nous évaluons son type (rouge->type).
- Nous pouvons alors proposer au joueur un menu de jeux et des options adaptées à la classe de la fourmi dont il est question.  
Par exemple pour une fourmilières, il y a deux options : produire ou ne pas produire,  
tandis que pour une Reine il y'a plus d'option : se déplacer(move), immobiliser(freeze), detruire(suicide) ou transformer en fourmilière.
- Dès que nous attendons une entree au clavier, nous évaluons celle ci à l'aide d'un boucle while, ainsi si l'utilisateur entre un mot diffèrent de ceux qui sont attendus (par exemple pour une reine, un mot diffèrent de « move », « suicide », « freeze » ou « transform »), nous sommes en mesure d'afficher un message lui demandant de recommencer la saisie.
- Pour la recuperation des elements au clavier, nous utilisons les fonction de saisie sécurisée.

- Pour chacune des actions souhaitée nous utilisons une des fonctions créées :
- Si une fourmiere veut produire un agent, nous vérifions que le joueur possède les fonds suffisants, puis nous initialisons son compteur de production et son champs type de production aux bonnes valeurs en fonction de l'agent à produire. Puis nous décrémentons son compteur de production, lorsqu'il vaut 0, nous appelons la fonction `product_ant` et le nouvel agent est créé.
- Si le joueur n'as pas assez d'argent nous affichons un message à l'ecran pour le lui faire remarquer .
- Si un agent veut se deplacer, nous demandons au joueur de saisir la destination (coordonnées de la case), nous initialisons les champs 'dest' de l'agent en question et nous appelons la fonction `ordre_ant` avec « move » (mot saisi au clavier par le joueur) comme parametre d'entree. Ainsi la fourmi avance d'un pas par tour vers sa destination.
- Si un agent veut se suicider, nous appelons la fonctions `ordre_ant` avec « suicide » (mot saisi au clavier par le joueur).
- Si un agent veut s'immobiliser, nous appelons la fonctions `ordre_ant` avec « freeze » (mot saisi au clavier par le joueur).
- Si une Reine veut se transformer, nous verifions que le joueur dispose d'assez d'argent, puis nous appelons les fonctions `NewQueen` et `product_ant`(avec comme argument « suicide ») pour détruire la Reine et la remplacer par une fourmiere.

- Cependant, il se peut que le pointeur rouge pointe sur un agent mais que le joueur ne puisse pas lui donner d'ordre.
- Alors si l'agent est une Fourmiliere, nous verifions qu'elle n'est pas en production. Si c'est la cas nous continuons la production (decrementer le compteur de production ou produire si ce dernier est nul)
- Si c'est une Ouvriere, nous verifions qu'elle n'est pas immobilisée. Et s'il c'est le cas nous ajoutons aux fonds du joueur 1 piece.
- Sinon nous verifions que l'agent n'est pas en mouvement. Si c'est le cas, nous le faisons avancer d'un pas vers sa destination.
- Toutes ses actions se produisent dans le cas ou le joueur decide de jouer (1).

- Si le joueur decide de passer son tour (2), nous parcourons tout de meme la liste chainée de ces agents pour effectuer les 3 taches ci-dessus.
- Si le joueur décide de sauvgarder la partie(3), nous faisons appel à la fonction Save, et passons son tour, ainsi lors du chargement de cette partie, nous allons redonner la main à ce joueur.
- Enfin si le joueur decide de quitter le jeu(4), nous liberons la mémoire allouée par des malloc à l'aide de free(), puis nous utilisons un break pour quitter la boucle while qui porte sur la valeur du choix.
- Nous faisons également des appels réguliers la fonction affiche Print\_board pour afficher le plateau.
- Nous avons mis en place un système de points pour les joueurs, ainsi lorsqu'ils produisent un agent ils gagnent un certain nombre de point (en fonction de l'agent crée), aussi lorsqu'ils gagnent des batailles ils reçoivent des points (ici aussi le nombre de points rapportés dépend du type de la fourmi vaincue).
- Nous faisons toutes ces opérations pour chacun des deux joueurs, ce qui au final nous donne une fonction main assez conséquente.

- La complexité de cet algorithme est plus importante que pour celles des autres, en effet du fait des boucles while() imbriquée, nous tombons sur une complexité dépendant des longueurs des listes doublement chaînées des deux joueurs. De plus du fait des nombreux appels aux fonctions, la complexité augmente considérablement. Il est assez difficile de l'explicité, mais on peut remarquer qu'un tel algorithme n'est pas le plus optimisé.
- Avoir une fonction main de cette longueur n'est, une fois encore, pas très esthétique, mais cela n'est pas très gênant dans la mesure où les parties correspondant aux deux joueurs sont assez similaires. Bien que ce ne soit pas particulièrement lié à cette partie du code, nous avons rencontré beaucoup de difficultés avec les allocations mémoire, notamment avec la libération mémoire (et cela dans toute les fonctions).