Rapport projet :
Puissance N

Projet réalisé par :

Pereira Hugo, Le Moing Lucas, Paolini Lucien

Dans le cadre de l'UV :

IFB

Gérant de l'UV :

Alexandre Lombard

Professeur de TD

Nathan Crombez

utbm
université de technologie
Belfort-Montbéliard

Table des matières

Introduction

Sujet du projet

Dans le cadre de l'UV IFB2, nous avons été amenés à réaliser un Puissance N en langage C, à l'aide de l'IDE (integrated development environment) CLion, et du compilateur MinGW.
La consigne étant de réaliser un Puissance N avec les règles suivantes : au lancement du programme, l'utilisateur a trois options : créer une partie, charger une partie existante ou quitter le programme. Si le joueur choisit une nouvelle partie, une partie est créée, et il décide s'il veut jouer contre une personne, ou contre une intelligence artificielle. Ensuite, chacun joue à son tour avec deux options : ajouter ou retirer un jeton. Le jeu s'arrête lorsqu'un des deux joueurs a N jetons alignés (que ce soit à la verticale, à l'horizontale, ou de travers). Si l'utilisateur décide de charger une partie existante, il pourra continuer à jouer une partie déjà entamée (grâce à la fonction sauvegarder et quitter). Enfin, si l'utilisateur fait le choix de quitter, le programme s'arrête.

Des consignes nous étaient données et étaient obligatoires pour la construction de notre projet telle que les options **Démarrer une nouvelle partie**, **Charger une partie** et **Quitter**. Nous devions également respecter le fait qu'un ordinateur devait jouer à la place d'un potentiel 2e joueur si celui-ci ne jouait pas. Et nous devions faire apparaître les options suivantes durant les parties : **Retirer un jeton**, **Ajouter un jeton**, **Sauvegarder et quitter**.

Enfin, des instructions supplémentaires nous étaient imposées pour la réalisation du Puissance N. Nous devions donc définir les fonctions suivantes : **show_grid**, **add_token**, **remove_token**, **check_winner** ainsi que la structure **Grid**. Nous expliquerons leurs spécificités et comment nous les avons codées durant la suite du rapport. Mais ces fonctions ne sont pas suffisantes pour faire fonctionner le jeu, nous avons donc été amenés à créer d'autres fonctions.

Organisation du travail

Étant en distanciel nous nous sommes rassemblés sur des plateformes vocales pour effectuer le projet et ajouter la dernière version du projet dans un groupe privé.
Nous avons réalisé le rapport sur Google Doc pour pouvoir chacun de son côté ajouter des éléments de réponses en fonction de ce qu'il a codé. Nous nous rassemblions donc quand on pouvait, ou bien nous travaillions chacun de notre côté quand on avait nos propres disponibilités.

Nous allons donc vous présenter en détails comment fonctionne notre projet, les différentes fonctions, avec quoi nous les avons codées et pourquoi dans l'ordre chronologique du code C.

Explication du programme

Base du programme

La structure Grid

Avant de pouvoir commencer à expliquer les différentes fonctions utilisées, il faut d'abord expliquer la structure utilisée qui nous a servi dans de multiples parties de notre projet.
La structure **Grid** nous a été imposée pour la réalisation du projet est nous permet notamment de créer notre tableau où le jeu se fera.
Nous l'avons définie dans une bibliothèque que nous avons appelée **Jeu.h** et que nous avons appelée dans le **main** du projet et dans le **Jeu.c** là où figurent toutes nos fonctions.
Elle est définie avec deux variables avec des entiers et une variable caractère.
Une variable correspondant à la longueur du tableau, une à la largeur et l'autre à un tableau à deux dimensions permettant de visualiser celui-ci appelé **grille**.
Dans la suite du programme, nous appelons régulièrement une variable **plateau** définie avec la structure **Grid** pour pouvoir l'associer à un tableau à deux dimensions qui correspondra au plateau de jeu.

Voici l'algorithme correspondant à la structure créée :

Structure Grid *entier* : hauteur, largeur
*tableau de hauteur * largeur caractères* : grille

Les choix (le main)

L'utilisation des **switch** dans le **main** du projet, et dans la fonction **jouer** a été préférée à des boucles de **if** car cela nous a permis d'alléger et simplifier le code, l'utilisation des commandes **if** aurait été beaucoup plus brouillon dans le code car il aurait fallu les imbriquer. De plus, la structure à multiple choix est utilisée dans le cas où il n'y a qu'un seul type de variable, ici des entiers en l'occurrence.
Le code est donc beaucoup plus structuré et les conditions pour utiliser ce type de commande (**switch)** ont été respectées.
Dans le **main**, la valeur qui va définir la case du **switch** est la valeur renvoyée par la fonction **Option**.

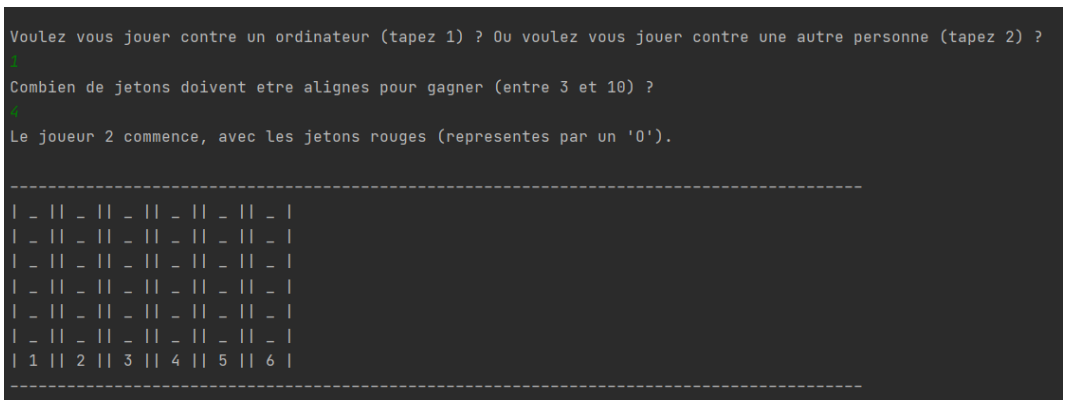
Paramétrage et affichage

La fonction paramètre

La fonction **paramètre** regroupe les informations importantes avant de lancer une partie.
Cette fonction demande à l'utilisateur le nombre de joueurs, le nombre de jetons qu'il faut aligner pour gagner la partie et enfin définir l'ordre de jeu.
Tous les paramètres de la fonction ont été initié par adresse avec les " ***** ". Le premier **printf** qui demande le nombre de joueurs va donc modifier la variable **Nbr_joueur**.
Une boucle **while** est utilisée comme message d'erreur. La boucle **while** est la plus optimisée ici car lorsque le nombre de joueurs est supérieur à 2 ou inférieur à 1, la boucle se répétera. De plus, l'utilisation de **while** au lieu de **do while** permet de directement afficher le message d'erreur.
Dans un second temps, la fonction demande combien de jetons le joueur doit aligner afin de gagner. La variable **N** servira alors pour la fonction **show_grid,** car dans la fonction paramètre on initialise la largeur du tableau et la hauteur de celui-ci en ajoutant +2 à **N**.
A l'aide d'une fonction **malloc** on peut créer un tableau dynamique à double dimensions en utilisant deux boucles **for**, une pour la hauteur du tableau et une pour la largeur de celui-ci. La fonction **for** est utilisée car la hauteur et la largeur sont connues grâce à la variable **N**.
Enfin la fonction définie qui commence entre les deux joueurs. L'utilisation de **rand()%** est donc utilisée pour laisser le hasard choisir.
La fonction **if** qui suit permet juste d'afficher qui commence avec la forme des jetons associés aux joueurs.
Comme il a été dit précédemment, tous les paramètres ont été initialisés par adresse pour que les fonctions qui suivent gardent ces mêmes paramètres.

La fonction show_grid

Afin de pouvoir afficher le tableau permettant de jouer au jeu, nous avons d'abord demandé à l'utilisateur de choisir le nombre de jetons qu'il voulait aligner pour pouvoir créer le tableau en fonction de ça. Nous avons donc utilisé deux boucles **for** pour pouvoir afficher en même temps la hauteur et la largeur du tableau. La boucle **for** ici est la plus utile car nous connaissons déjà le nombre de colonnes et de lignes correspondantes.
L'utilisation du caractère " **|** " pour délimiter les cases. Voici un exemple pour un tableau de 6 colonnes (donc un puissance 4) :



Nous avons bien ici 6 colonnes délimitées chacune par des caractères " **|** ", tout ça grâce à l'utilisation de deux simples boucles **for**.
L'utilisation d'une troisième boucle **for** permet d'afficher les numéros en bas du tableau, **k+1** correspondant à la i-ème colonne du plateau de jeu. Donc pour chaque colonne, le nom de la colonne est affiché. Ci-dessous l'algorithme de cette fonction :

```
Vide Show_grid (Grid plateau)
Variables   entier : i, j, k
Début
        i ← 0
   Pour i de 0 à i < plateau.largeur de 1
        Pour j de 0 à j < plateau.hauteur de 1
            Ecrire ( |plateau.grille [i][j]| )
        FinPour
        Ecrire ()
   FinPour
   Pour k de 0 à k < plateau.largeur de 1
        Ecrire ( |k+1| )
   FinPour
   Ecrire (----------)
Fin
```

Les fonctions option et option_play

Ces deux fonctions sont relativement similaires. Elles sont constituées d'un **printf** et d'un **scanf** permettant de modifier la variable principale.
A la fin, la commande : **return** "la variable", permet d'associer la valeur obtenue à la fonction **switch** utilisée dans le **main** du projet et dans la fonction **Jouer** qui sera expliquée plus tard dans le rapport. Cela permet de savoir le choix fait par l'utilisateur et de les valider dans les **switchs**. Car étant donné que les variables sont des entiers entre 1 et 3, les **cases** qui correspondent aux **switchs** sont directement associés à ces deux fonctions.
C'est pour cela que dans le **main** et dans la fonction **Jouer**, la commande **switch** appelle directement la fonction **option** ou **option_play**, pour que la valeur retournée corresponde aux différents choix possibles.

Jouabilité

La fonction jouer

La fonction **Jouer** est le pont entre le **main.c** et toutes les autres fonctions qui composent notre jeu.
Cette fonction définit le déroulement de la partie c'est-à-dire qu'elle permet d'ajouter un jeton, en retirer un ou sauvegarder la partie.
Tout d'abord la fonction va continuer à boucler tant qu'il n'y a pas de gagnant à la partie, vérifier avec la fonction **Check_winner,** ou si la fonction **Vérification** renvoie -1 (ceci sera expliqué plus tard), ce qui signifie que le plateau de jeu est complet, il y aura donc égalité. Toutes ces conditions sont permises par une boucle **do while** qui est plus optimisée que les autres boucles car le programme va au moins la parcourir une fois.
Dans un second temps l'utilisation d'un **switch** a été employé car cette fonction utilise ce que la fonction **Option_play** a renvoyé, une valeur entière entre 1,2 et 3. Le switch contient donc 3 cas possibles qui ont déjà été énumérés précédemment.
Pour ajouter ou retirer un jeton, la fonction **Jouer** fait appel aux fonctions **Add_token** et **Remove_token** en tant que conditions de **do while**. Ces conditions sont utilisées car les fonctions énoncées précédemment retourne à la fin de leur programme une valeur entre 0 et 1, c'est donc cette valeur que nous utilisons dans ces boucles **do while**.
Lorsqu'on appelle les fonctions **Add_token** et **Remove_token** cela demande des spécificités. Par exemple, lorsque la fonction **Add_token** est appelée, on écrit en paramètre le **plateau** qui a été défini avec la structure **Grid** puis on écrit la variable **colonne** (correspond au nombre de colonnes) à laquelle on retire 1 car les colonnes commencent au chiffre 0. Puis on effectue un modulo à la variable **num_j** car si on ne fait pas, cette variable prendra la valeur pour le joueur 1, mais le joueur 2 continuera jusqu'à l'infini, c'est pour ça que le modulo 2 est nécessaire pour éviter tout types de bugs.
Si la variable ne correspond ni à 1 et 2 alors la partie est sauvegardée, la fonction **Jouer** fait tout simplement appel à la fonction **Save** et **return EXIT_SUCCESS** car sinon l'utilisation d'un **break** à la fin du **switch** n'aurait pas arrêté le programme. La fonction **Jouer** fait aussi intervenir le bot si l'utilisateur veut jouer tout seul à l'aide d'une fonction **if** qui a pour conditions le **Nbr_joueur** et **num_j** qui doivent valoir 1 et 2, pour que la fonction laisse la fonction **Bot_token** se déclencher.

Ci-dessous l'algorithme de cette fonction :

```
Entier jouer (Grid plateau, entier : num_j, *blocage, Nbr_joueur )
Variables   entier : colonne
Début
  Répéter
        Show_grid(plateau) ;
        Si num_j = 1 [2] et Nbr_joueur = 1 alors
              Bot_token(&plateau, num_j[2], blocage)
        Sinon
              Choisir Option_play parmi
                    Constante 1
                          Répéter
                                Ecrire (Dans quelle colonne voulez vous ajouter votre jeton ?)
                                Lire colonne
                                Tant que colonne < 1 ou colonne > plateau.hauteur
                                      Ecrire (Erreur ! Dans quelle colonne voulez vous ajouter votre jeton ?)
                                      Lire colonne
```

```
                    FinTantQue
                Tant que add_token(plateau, colonne-1, num_j[2], blocage) = 0
            Constante 2
                Répéter
                    Ecrire (Dans quelle colonne voulez vous ajouter votre jeton ?)
                    Lire colonne
                    Tant que colonne < 1 ou colonne > plateau.hauteur
                        Ecrire (Erreur! Dans quelle colonne voulez vous enlever votre jeton ?)
                        Lire colonne
                    FintTantQue
                Tant que remove_token(plateau, colonne-1, blocage) = 0
            Sinon
                Save(plateau, num_j[2])
                Ecrire (Merci d'avoir joué !)
                Retourner EXIT_SUCCESS
        Fin
    FinSi
    Num_j ← num_j + 1
Tant que check_winner(&plateau) = -1 et  verification(&plateau) ≠ -1
Si check_winner(&plateau) = 1 alors
    Ecrire (Bravo au joueur 1 ! Représenté par un 'X')
Sinon si check_winner(&plateau) = 0 alors
    Ecrire (Bravo au joueur 2 ! Représenté par un 'O')
Sinon si verification(&plateau) = -1 alors
    Ecrire (Egalité !)

FinSi
Fin
```

La fonction add_token

Cette fonction permet d'ajouter un jeton au plateau de jeu.
Dans un premier cas, on utilise une première condition qui vérifie si la colonne que le joueur a choisie possède une case vide, ou bien si la colonne demandée est différente à la variable **blocage,** déjà expliquée dans la fonction **Remove_token.**
Si cela n'est pas le cas, alors la fonction va retourner 0, cette valeur sera après utilisée dans la fonction **Jouer.**
Si le joueur peut jouer, alors la fonction va ajouter un jeton dans la case qui correspond à celle du tableau à deux dimensions de la variable **plateau** également appelé dans cette fonction en tant que paramètre (la variable **grille**), avec en ligne **i -1**, une variable appelée dans la fonction qui correspond également à la ligne du plateau et qui a parcouru les lignes avec l'utilisation d'une boucle **while,** comme nous avons pu le faire dans la fonction **Remove_token**. Nous lui avons retiré 1 car la dernière valeur de **i** correspond à une case déjà remplie ou inexistante quand aucun jeton n'a été ajouté dans une colonne, il faut donc revenir en arrière de 1 pour accéder à une case vide. A la fin, nous avons assimilé -1 à la variable **blocage** pour que toutes les colonnes puissent être utilisées. Ci-dessous l'algorithme de cette fonction :

```
Entier Add_token (Grid plateau, entier : colonne, num_j, *blocage)

Variables   entier : i, colonne, num_j
Début
        i ← 0
  Si plateau.grille [0][colonne] = '_' et colonne ≠ *blocage alors
            Tant Que i ≠ plateau.largeur et plateau.grille [i][colonne] = '_'
                  i ← i + 1
            Fin Tant Que
            Si num_j = 1 alors
                  Plateau.grille [i-1][colonne] ← 'X'
            Sinon si num_j = 0 alors
                  Plateau.grille [i-1][colonne] ← 'O'
            FinSi
      *blocage ← -1
       Retourner 1
   Sinon
       Ecrire (Impossible !)
       Retourner 0
   FinSi

Fin
```

La fonction Remove_token

Ici on cherche à retirer un jeton avec la condition suivante : lorsqu'un joueur enlève un jeton, au prochain tour il sera impossible d'utiliser cette colonne.
Dans un premier temps, on utilise une fonction **while** qui va permettre d'assimiler à **i,** une variable de nombre entier initialisé à 0. Le **while** va permettre de faire descendre la variable **i** dans le plateau jusqu'à arrivé à un jeton, si c'est le cas alors **i-1** correspondra à la ligne où le joueur va pouvoir retirer son jeton sinon cela signifie que **i** est arrivé à la dernière case du plateau sans trouver de jeton.
Ensuite, on utilise un **if** avec deux possibilités : dans la première possibilité la fonction va retourner comme valeur 0 et l'utilisateur ne pourra pas retirer de jetons car la variable **i** aura atteint le première ligne du plateau en partant du bas, ce qui signifierait qu'il n'y a tout simplement pas de jetons dans la colonne, et la deuxième possibilité enlève le jeton et assimilé à la variable **blocage** la valeur de la colonne, et la valeur retourné est 1. Nous avons initialisé la variable **blocage** en tant que pointeur et paramètre de la fonction car nous en aurons besoin dans les fonctions **Bot_token**, **Add_token** et **Jouer**, elle va permettre de bloquer une colonne pendant un tour si un joueur a retiré un jeton, comme nous l'avons énoncé précédemment.
Ci-dessous l'algorithme de cette fonction :

```
Entier Remove_token (Grid plateau, entier colonne, *blocage)

Variables   entier : i, colonne, *blocage
Début
        i ← 0
    Tant que i ≠ plateau.largeur et plateau.grille [i][colonne] = '_'
        i ← i+1
    Fin Tant Que
    Si i = plateau.largeur alors
        Ecrire (Impossible !)
        Retourner (0)
    Sinon
        Plateau.grille [i][c] = '_'
        *blocage ← colonne
        Retourner (1)
    FinSi
Fin
```

La fonction Bot_token

Cette fonction va permettre à l'ordinateur de faire ses choix aléatoirement dans le programme. Pour les choix aléatoires nous avons initialisé une variable **choix** qui est égale à un nombre aléatoire entre 1 et 2.
Cette fonction, tout comme la fonction **Jouer**, appelle les fonctions **Remove_token** et **Add_token** à l'aide de boucles **while** qui prend en compte leur valeur retourné dans chaque fonction.
La spécificité de cette fonction est donc une boucle **if** qui traduit la chose suivante : si la fonction **Vérification** a bien retourné 1 et la valeur de la variable **choix.** En fonction de la valeur de cette variable, la fonction va donc ajouter un jeton et en retirer un avec des boucles **do while**. Ces boucles vont aléatoirement choisir la colonne où le bot va ajouter ou retirer son jeton tant que les fonctions **Remove_token** et **Add_token** renvoient 0, c'est-à-dire jusqu'à ce que le bot trouve une colonne valide pour faire fonctionner les fonctions.
A la suite des **do while** nous avons initialisé la variable **blocage,** qui soit bloque la colonne choisie par le bot dans le cas ou il choisit de retirer un jeton, ou soit il débloque toutes les colonnes quand il ajoute un jeton, pour cela la variable **blocage** voudra -1, car -1 correspondra jamais à une colonne du plateau de jeu.

La fonction vérification

Nous avons créé cette fonction pour éviter tout bug avec la fonction **Bot_token**, car celle-ci permet de vérifier qu'il y ait bien un jeton dans le plateau pour éviter que l'ordinateur n'en retire un là où il n'y en a avec l'utilisation d'une boucle **while**. Si la ligne dernière ligne du plateau est vide alors la fonction retourne 1, sinon 0. Une deuxième boucle **while** a également été inséré pour vérifier si la première ligne du plateau en partant du haut est pleine, si c'est le cas, un **esle if** va retourner la valeur -1

Les fonctions check_winner et check_direction

Ces deux fonctions permettent de vérifier s'il y a un gagnant, le définir et si oui d'arrêter la partie. Une permet de vérifier si N mêmes jetons sont alignés et l'autre permet de d'abord définir un jeton puis fais varier la direction pour définir un gagnant. Les deux fonctions sont donc complémentaires.
Dans un premier temps la fonction **Chek_winner** parcours le tableau grâce aux deux boucles **while** imbriquées jusqu'à trouver un jeton, que ce soit du joueur 1 ou 2. Dès lors un **if** composé de deux boucles **for** imbriquées, va déterminer un couple de variables dirX et dirY qui va former un vecteur direction. Dans ces boucles **for**, ces deux variables varient grâces à des valeurs qui vont de -1 à 1, d'où l'utilisation de boucle **for** car les valeurs sont connues, et chacune de ces valeurs sont assimilées à une direction.
C'est ici que la fonction **check_direction** entre en action car après avoir définie la direction, la fonction **chek_winner** fait appel à celle-ci pour pouvoir vérifier qu'il y ait assez de jetons alignés pour gagner la partie. Tout d'abord, une commande **if** a été utilisé pour vérifier que les deux variables de directions ne soient pas égales à 0 en même temps, si ce n'est pas le cas, alors une boucle **do while** a été utilisée pour pouvoir avancer dans la direction choisie, tout en ne sortant du tableau et tant qu'un jeton similaire au précédent est aligné (ceci a été définie dans les conditions de la boucle). Deux choix sont alors possibles, N jetons ne sont pas alignés, et alors la fonction retourne 0, dans le cas où N jetons sont alignés, la fonction va alors retourner 1, ceci va avoir un impact dans **Check_winner**.
On retourne alors dans le **if** de **Chek_winner**, dans le cas où la valeur retournée par **check_direction** est 1, un **if** va déterminer qui a gagné en observant quels types de jetons est à la même position que le jeton observé dans **chek_direction**, pour savoir quel joueur a donc gagné, la fonction retourne donc 0 ou 1 en fonction du type de jetons.
Si personne n'a gagné, alors **Chek_winner** va encore faire appel à **check_direction** en faisant varier le couple de vecteurs pour une nouvelle fois vérifier s'il y a un gagnant. Si à chaque direction, la valeur retournée par **check_direction** est 0, alors, grâce au double **while**, la fonction **Chek_winner** va chercher un autre jeton pour réitérer la même tâche.
S'il n'y a toujours pas suffisamment de jetons alignés pour gagner, alors **Check_winner** va retourner -1, qui sera donc utilisé dans notre fonction Jouer déjà expliquée précédemment.
Ci-dessous les algorithmes de ces deux fonctions :

Entier check_direction (*Grid* *plateau, *entier* ligne, colonne, dirX, dirY, *caractère* jeton)

Variables *entier* : Row, Col, token, ligne, colonne, dirX, dirY

Caractère : jeton

Début

Row ← ligne

Col ← colonne

token ← 0

Si dirY = 0 et dirX = 0 alors

Retourner (0)

FinSi

Répéter

token ← token + 1

Row ← Row + 1 * dirY

Col ← Col + 1 * dirX

Tant que Col < plateau->hauteur et Row < plateau->largeur et Col ≥ 0 et
Row ≥ 0 et plateau->grille[Row][Col] = jeton et token < plateau->hauteur-2

Si token < plateau->hauteur-2 alors

Retourner 0

Sinon

Retourner 1

FinSi

Fin

Entier check_winner (*Grid* *plateau)

Variables *entier*: ligne, colonne, i, j

Début

ligne ← 0

colonne ← 0

Tant que ligne < plateau->largeur

Tant que colonne < plateau->hauteur

Si plateau->grille [ligne][colonne] ≠ '_' alors

Pour i de -1 à i ≤ 1 de 1

Pour j de -1 à j ≤ 1 de 1

Si check_direction(plateau, ligne, colonne, i, j,
plateau-> [ligne][colonne] = 1) alors

Si plateau->grille [ligne][colonne] = 'O' alors

Retourner 0

Sinon si plateau->grille [ligne][colonne] = 'X' alors

Retourner 1

FinSi

FinSi

FinPour

FinPour

FinSi

colonne ← colonne + 1

FinTantQue

colonne ← 0

ligne ← ligne + 1

FinTantQue

Retourner -1

Fin

Les sauvegardes

La fonction save

Cette fonction a pour but d'établir une sauvegarde au cours d'une partie pour pouvoir la lancer plus tard en quittant le projet.
La fonction **Save** fonctionne à l'aide d'un fichier texte.
Le fichier se crée tout seul car on l'appelle avec la commande "**w**" ce qui signifie que si le fichier de base est inexistant, il sera créé après l'utilisation de la fonction **Save**. On pourra donc écrire dans le fichier mais pas lire son contenu.
Dans ce fichier, le contenu suivant est sauvegardé : le nombre de joueurs, la colonne bloquée par la fonction **Remove_token**, la largeur du tableau, le plateau de jeu, et par conséquent l'emplacement des jetons. Les jetons sont conservés grâce à l'utilisation de deux boucles **for** imbriqués. La première correspond à garder la taille du tableau, et la deuxième boucle va utiliser un **fprintf** qui va parcourir tout le tableau pour enregistrer les jetons, comme nous l'avons énoncé précédemment.
Ci-dessous l'algorithme de cette fonction :

```
Vide Save (Grid plateau, entier num_j, blocage)
Variables   entier : num_j, k, i, j
Début
    FILE* f = fopen("Sauvegarde.txt , " w")

  Si f ≠ 0 alors

      frintf(f, "%d", num_j)
      frintf(f, "%d", blocage)
      frintf(f, "%d", plateau.largeur)

      Pour i de 0 à i < plateau.largeur de 1
          Pour j de 0 à j < plateau.hauteur de 1
              fprintf(f, "%c", plateau.grille [i][j])
          FinPour
      FinPour
  FinSi

  Fermer(f)
Fin
```

La fonction load

La fonction **Load** est la suite logique de la fonction **save**. Elle permet de relancer une partie déjà sauvegardée.
Dans cette fonction nous appelons le fichier texte déjà sauvegardé en l'appelant avec le même nom que dans la fonction **Save**, et en utilisant "**r**" qui permet de lire le contenu d'un fichier sans pour autant pouvoir écrire dedans.
La première fonction **if** permet de vérifier s'il y a bien un fichier de sauvegarde déjà existant, cela nous permet d'éviter d'éventuels problèmes. La fonction **while** permet de parcourir tout le fichier avec l'aide d'un tableau de caractères qui parcourt tout le fichier. On initialise notre variable **taille** grâce à **Strlen** ; cette variable va nous permettre de supprimer le **"\n"** à la fin de chaque ligne du fichier. La deuxième fonction **if** permet de savoir quel joueur va commencer à jouer. Pour expliquer, dans le fichier de sauvegarde, la première valeur sauvegardée est le

numéro du joueur qui commence. Donc on assimile la variable **player** à ce chiffre. Etant donné que notre tableau est un tableau de caractères, celui-ci renvoie par exemple à 49 si le caractère est ‘1’, on a donc soustrait le caractère ‘0’ qui correspond à 48 à cette valeur (référence à la table ascii). Donc si le caractère est égal à deux, ce code renverra l’entier deux, et de même si la valeur est égale à un. On réitère ce programme sur la deuxième ligne du fichier pour définir la colonne bloquée lors de la précédente partie s’il y en a une.

La suite de la fonction est plus simple. La deuxième ligne de la sauvegarde correspond à la largeur du tableau, ce qui permet à la fonction **Load** de réinitialiser ce tableau avec une fonction **malloc** et deux boucles **for** imbriquées. Le dernier **else** ne prend pas en compte les deux premières lignes et remet en place les jetons avec une boucle **for** (car ici le nombre de colonnes est connu à l’aide de la deuxième ligne de sauvegarde).

La fin de la fonction ferme le fichier et retourne la variable **Player** qui sera utilisé dans le **main** à la place de la variable **num_j**, qui correspond donc au Joueur 1 ou 2.

Conclusion

Pour conclure, cela a été très formateur pour l'apprentissage du C et nous sommes fiers de ce que nous avons réalisé.
Il nous a permis également de mieux travailler en groupe et surtout en distanciel. Cela n'a pas été évident pour nous, nous nous sommes organisés de sorte que personne ne se gêne durant la programmation. Le rapport a été bien plus simple à réaliser car nous pouvions le faire en simultané sur Google Doc.
En ce qui concerne le code, nous avons créé un maximum de fonctions pour pouvoir alléger un maximum le **main**. Cela nous a permis de mieux structurer nos idées et si un bug avait lieu, cela était plus simple pour nous de le repérer et de le corriger.
Nous voulions également remercier nos professeurs d'informatique qui nous ont aidé et aiguillé durant plusieurs séances, Nathan Crombez notre professeur de TD ainsi que Nicolas Gaud notre professeur de CM.
En ce qui concerne le cahier des charges, nous avons respecté toutes les consignes demandées, que ce soient les fonctions définies par la consigne et leur fonctionnalité comme nous avons l'expliquer tout au long du rapport.
Pour les améliorations, nous aurions pu enlever la dernière ligne du plateau de jeu lorsque celui-ci est plein pour éviter les égalités, améliorer l'interface graphique ou encore optimiser le code pour pouvoir effectuer un puissance N supérieur à 10.